PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA INFRAESTRUTURA PARA CONSISTÊNCIA DOS PROCESSOS DE SOFTWARE BASEADOS NO METAMODELO SPEM 2.0

ELIANA BEATRIZ PEREIRA

Tese apresentada como requisito parcial à obtenção do grau de Doutor, pelo programa de Pós Graduação em Ciência da Computação da Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Ricardo Melo Bastos

Porto Alegre

2011

P436i Pereira, Eliana Beatriz
Uma infraestrutura para consistência dos processos de software baseados no metamodelo SPEM 2.0 / Eliana Beatriz Pereira. – Porto Alegre, 2011.
279 f.

Tese (Doutorado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Ricardo Melo Bastos

1. Informática. 2. Engenharia de Software. 3. Software – Análise de Desempenho. 4. Simulação e Modelagem em Computadores. I. Bastos, Ricardo Melo. II. Título

CDD 005.1

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO

Tese intitulada "Uma Infraestrutura para Consistência dos Processos de Software baseados no Metamodelo SPEM 2.0", apresentada por Eliana Beatriz Pereira, como parte dos requisitos para obtenção do grau de Doutor em Ciência da Computação, Sistemas de Informação, aprovada em 27/06/2011 pela Comissão Examinadora:

Prof. Dr. Ricardo Melo Bastos – Orientador — PPGCC/PUCRS

Prof. Dr. José Palazzo Moreira de Oliveira - — UFRGS

Prof. Dr. Toacy Cavalcante de Oliveira - — COPPE/UFRJ

Prof. Dr. Marcelo Blois Ribeiro – — PPGCC/PUCRS

Homologada em 03./.10./.2012., conforme Ata No. 21..... pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P. 32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

AGRADECIMENTOS

A Deus, pela sabedoria, persistência e oportunidades que tem posto em minha vida.

Aos meus pais, Arnaldo e Flávia, e minhas irmãs, Márcia e Patrícia, pela compreensão do tempo em que estive ausente e por acreditarem em mim e no meu trabalho, mas principalmente por torcerem por esta conquista.

Ao meu orientador, Ricado Bastos, por sua amizade e cumplicidade, fruto de longo tempo de estrada juntos desde meu mestrado, além do apoio e incentivo em todos os momentos que foram importantes para minha vida acadêmica e pessoal. A você serei eternamente grata por tudo e espero que ainda realizemos muitos trabalhos em conjunto.

Aos membros da banca, professores Toacy C. Oliveira, José Palazzo e Marcelo Blois, pela aceitação do convite de participação na avaliação deste trabalho. Em especial um agradecimento ao professor Toacy C. Oliveira pela amizade e pelo seu apoio técnico neste trabalho. Espero que esta parceria continue.

Ao Márcio Brigidi, com quem tive a imensa satisfação de trabalhar, pessoa que aprendi a admirar e tenho muito a agradecer pelo apoio que foi fundamental na realização deste trabalho. Meu sincero agradecimento e reconhecimento por sua amizade e cumplicidade que foram essenciais para mim.

Aos meus bolsistas, Bruno e Leonardo. Obrigada por compartilharem comigo muitas tardes de trabalho na implementação do sSPEM Tool. A participação de vocês foi fundamental para a realização da pesquisa. Considero esta vitória de vocês também!

À minha grande amiga Tanise Novelo, que conheci durante o doutorado. Sua amizade sem dúvida foi um presente. Com certeza essa caminhada se tornou menos difícil pelo seu incomensurável incentivo e companheirismo. Valeu amiga, pelas horas a fio no skype fazendo praticamente tudo juntas!!! TESE, Negrinho, TESE, Compras, Almoços, TESE, Jantas....Te considero uma irmã!

A todos os meus amigos pela paciência e incentivo nesta caminhada. Em especial um agradecimento a Analuisa Monteiro, Mariana Denes, Ana Mansur, Sobral, Marcos Tadeu, Carlos Bragança, Mirela Medeiros, Elisa Cerri, Bruno Pilz e Leonardo Machado. Vocês são especiais para mim!

Ao Convênio Dell/PUCRS pelo apoio financeiro para realização do trabalho e ao SERPRO pela liberação de horas, as quais foram imprescindíveis para realização desta pesquisa.

UMA INFRAESTRUTURA PARA CONSISTÊNCIA DOS PROCESSOS DE SOFTWARE BASEADOS NO METAMODELO SPEM 2.0

RESUMO

O uso de processos de desenvolvimento de software nas organizações de TI tem se tornado cada vez mais comum. Um dos motivos é que a qualidade do produto de software está relacionada com a qualidade do processo utilizado na sua construção. Nesse contexto, o interesse das organizações é estabelecer um ou mais processos de desenvolvimento de software bem definidos; adaptando-os, quando necessário, para atender metas específicas dos projetos de software. Contudo, devido à grande quantidade de elementos e relacionamentos que um processo de desenvolvimento de software possui, as atividades de definição e adaptação de processos são tarefas não triviais. Quando alguns cuidados não são tomados, inconsistências podem ser facilmente introduzidas em um processo de desenvolvimento de software, fato que pode, muitas vezes, ocasionar a geração de um processo inadequado que acarretará em erros durante a execução de um projeto de software. Considerando a necessidade de evitar inconsistências em um processo de desenvolvimento de software, esta pesquisa propõe uma infraestrutura que viabiliza a definição e adaptação dos processos de desenvolvimento de software consistentes baseados no metamodelo SPEM 2.0. A infraestrutura definida é composta por uma extensão ao metamodelo SPEM 2.0, um conjunto de regras de boa-formação para consistência dos processos de desenvolvimento de software e um protótipo de ferramenta que auxilia o uso do metamodelo proposto e das regras de boa-formação.

Palavras chave: consistência, processo de desenvolvimento de software, definição de processos, adaptação de processos, metamodelo SPEM 2.0

AN INFRASTRUCTURE TO CONSISTENCY OF SPEM-BASED SOFTWARE DEVELOPMENT PROCESSES

ABSTRACT

The use of software development processes in the IT organizations has become common. This happens because the quality product is related to the process quality. The main interest of the IT companies is to adopt one or more well-defined software development processes and tailor them when necessary to meet the projects specific needs. However, since the amount of elements and relationships of a software development process is huge, defining and tailoring a software development process are not trivial activities. Inconsistencies may easily be introduced into a software development process when certain precautions are not taken. As a consequence, an inadequate software development process may be created to a software project causing errors during its enactment. Considering the need to avoid inconsistencies in a software development process, this research proposes a consistence infrastructure that enables defining and tailoring consistent software development processes based on SPEM 2.0 metamodel. The proposed infrastructure is composed by an extension to the SPEM 2.0 metamodel, a set of well-formedness rules related to the consistency of the software development processes and a tool prototype that supports automatically the proposed metamodel and well-formedness rules.

Keywords: consistency, software development process, process definition, process tailoring, SPEM 2.0 metamodel

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

APE	agentTool Process Editor
CMMI	Capability Maturity Model Integration
EMF	Eclipse Modeling Framework
EPF	Eclipse Process Framework
FoI	First-Order Logic
MDD	Model Driven Development
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OPEN	Object-oriented Process, Environment and Notation
OPF	OPEN Process Framework
PA	Process Area
PML	Process Modeling Language
RSM	Rational Software Modeler
RUP	Rational Unified Process
SME	Situational Method Engineering
SPEM	Software & Systems Process Engineering Meta-Model Specification
SPI	Software Process Improvement
TI	Tecnologia da Informação
UML	Unified Modeling Language
WBS	Work Breakdown Element

SUMÁRIO

1 INTRODUÇÃO

Ao longo dos últimos anos, o software tem conquistado um papel essencial e crítico em nossa sociedade, uma vez que a dependência por produtos e serviços oferecidos através de sistemas de computador, é cada vez mais comum [Fug00]. Nesse sentido, a indústria do software está passando por um acentuado crescimento, impulsionado pelas exigências do mercado, no sentido de desenvolverem seus produtos de software em prazos e custos determinados, obedecendo a padrões de qualidade.

Para auxiliar as organizações de Tecnologia da Informação - TI a atingirem níveis mais elevados de qualidade em seus produtos de software, satisfazendo prazos e custos, encontram-se os processos de desenvolvimento de software (ou simplesmente processos de software), que são a principal base para melhorar a produtividade das equipes de desenvolvimento, com vistas à qualidade dos produtos de software desenvolvidos [ZAH98]. Com base nesse contexto, o interesse das organizações de TI é definir um ou mais processos de software, adaptando-os, quando necessário, para atender metas específicas dos projetos de software.

Definir um processo de software consiste em escolher um conjunto de elementos tais como tarefas, produtos de trabalho e papéis, combinando e organizando estes elementos em fluxos de trabalho para a construção ou manutenção de um produto de software. Já a adaptação desse processo envolve adicionar, excluir e/ou modificar alguns de seus elementos e relacionamentos com o objetivo de torná-lo mais apropriado para o alcance das metas de um projeto de software específico [Yoo01].

Dada a complexidade de um processo de software, decorrente da sua elevada quantidade de elementos e relacionamentos, ambas as atividades de definição e adaptação de processos não são triviais. Alguns cuidados devem ser tomados durante estas atividades, principalmente para evitar a incidência de inconsistências nos processos de software. De acordo com Harmsen [Har97], uma inconsistência em um processo de software é representada por um conjunto de informações incompletas e/ou incoerentes nos seus elementos e relacionamentos, e, quando não identificada previamente, esta inconsistência é reproduzida na execução dos projetos de software, o que pode ocasionar o seu insucesso.

Na literatura, soluções têm sido propostas para a identificação de inconsistências em processos de software. Em geral, o foco tem sido a definição de um conjunto de

regras de consistência e também a definição de uma atividade específica para a checagem da consistência dos processos de software [Atk07], [Baj07] e [Hsu08]. Além disso, um ponto comum nessas soluções é a indicação do uso de um metamodelo como base para definição e adaptação dos processos de software.

Segundo Perez e Sellers [Per07a], processos de software definidos a partir de um metamodelo geralmente oferecem um alto grau de formalismo e melhor suporte para consistência e customização, uma vez que os conceitos que formam sua base são explicitamente definidos. Além disso, como metamodelos são normalmente representados com diagramas de classe da *Unified Modeling Language* – UML [Omg03], eles já são capazes de representar restrições através das multiplicidades entre suas metaclasses, e, ainda, permitem associar restrições mais complexas através da linguagem natural ou linguagens formais, tais como, *Object Constraint Language* – OCL [War03], *Z* [Mou01], Lógica de Primeira Ordem (em inglês *First-order Logic* – FoL) [Smu95], entre outras.

Existem diversos metamodelos de processo, tais como: *Software & Systems Process Engineering Meta-Model Specification* - SPEM 1.1 [Omg02], *OPEN Process Framework* – OPF [Fir02], entre outros. Atualmente, o metamodelo referência para a definição de processos de software é o SPEM 2.0, que foi publicado pela *Object Management Group* - OMG em 2007 [Omg07a]. Tal metamodelo tem sido utilizando em muitas pesquisas, tais como em [Was06], [Ben07], [Fer10], [Rod10] e [Par11].

O SPEM 2.0 é usado para definir processos de software e também prevê atividades de adaptação, já que ele possui um conjunto de relacionamentos para reuso e variabilidade que permite definir variações de um mesmo processo de software para diferentes projetos. Segundo a especificação do SPEM 2.0, a meta em definir um metamodelo referência é viabilizar a criação de uma grande variedade de processos de software para diferentes culturas que utilizam diferentes níveis de formalismo e modelos de ciclo de vida.

Apesar da importância do supracitado, é necessário ressaltar que embora o metamodelo SPEM 2.0 seja bastante completo e represente um avanço para área de processos de software, este metamodelo não possui regras específicas para as atividades de definição e adaptação de processos consistentes. Além disso, analisando a literatura disponível, constata-se que os estudos que abordam o aspecto da consistência em processos de software só o fazem de forma parcial. Esse é o caso, por exemplo, dos estudos que apresentam regras apenas para a definição de processos e não abordam as atividades de adaptação [Atk07], [Hsu08], [Bao08], [Fer10], [Rod10], ou ainda, dos

estudos que tratam da consistência de apenas alguns aspectos e/ou elementos de um processo de software, como por exemplo, da consistência referente ao sequenciamento de tarefas [Bao08].

Diante disso, constata-se como uma lacuna para área de pesquisa sobre processos de software, a falta de estudos mais completos sobre a consistência desses processos durante suas atividades de definição e adaptação. Entende-se por mais completos estudos que considerem os principais elementos de um processo de software e que definam soluções para o tratamento da consistência em ambas as atividades de definição e adaptação de processos. Segundo a OMG [Omg02] e [Omg07a], os principais elementos de um processo de software são *Atividades, Tarefas*, *Papéis* e *Produtos de Trabalho.*

Baseado no exposto acima, esta tese tem como o foco estudar todos os aspectos de consistência de um processo de software considerando seus principais elementos e relacionamentos nas suas atividades de definição e adaptação. Durante a condução desse estudo, o metamodelo SPEM 2.0 será utilizado como referência e como resultado de pesquisa será apresentada uma infraestrutura que viabiliza a definição e adaptação de processos de sofware consistentes. A infraestrutura definida será composta por uma extensão ao metamodelo SPEM 2.0, um conjunto de regras de boa-formação para consistência dos processos de software e um protótipo de ferramenta que auxilia o uso do metamodelo proposto e das regras de boa-formação. Como resultado adicional da pesquisa, será também apresentado um guia que auxilia a definição e adaptação de processos de software utilizando a infraestrututra apresentada neste trabalho.

As seções seguintes, apresentam um detalhamento sobre os objetivos, etapas de pesquisa e contribuições desta pesquisa.

1.1 Objetivo Geral da Tese

O objetivo geral desta pesquisa é o desenvolvimento de uma infraestrutura que suporte a definição e adaptação de processos de software consistentes baseados no metamodelo SPEM 2.0.

1.1.1 Objetivos Específicos

Para a consecução do objetivo geral proposto pode ser desmembrado nos seguintes objetivos específicos:

- Identificar um conjunto de premissas para a definição de processos de software consistentes;
- Propor uma extensão ao metamodelo SPEM 2.0 para o atendimento das premissas para a contrução de processos de software consistentes. A extensão proposta inclui e/ou altera alguns elementos, bem como alguns relacionamentos do metamodelo SPEM 2.0;
- Propor um conjunto de regras de boa-formação para consistência que serão utilizadas nas atividades de definição e adaptação dos processos de software. Como forma de respeitar as regras de boa-formação definidas na pesquisa será desenvolvida a análise dos impactos sobre os elementos e relacionamentos para os mecanismos de adaptação do metamodelo SPEM 2.0;
- Desenvolver um guia que auxilia o uso da infraestrutura para consistência proposta nesta pesquisa.
- Formalizar as regras de boa-formação para consistência em Lógica de Primeira Ordem;
- Implementar um protótipo de ferramenta que automatize a infraestrutura para consistência proposta nesta pesquisa;
- Avaliar a solução proposta, utilizando-se de cenários simulados que são baseados no processo *OpenUP* [Ope10].

Uma vez apresentado os objetivos geral e específicos deste estudo, introduz-se a questão de pesquisa utilizada:

“Como viabilizar a consistência dos processos de software baseados no metamodelo SPEM 2.0 durante suas atividades de definição e adaptação?”

1.2 Metodologia de Pesquisa

Embora alguns trabalhos relacionados com o objetivo desta pesquisa tenham sido encontrados na revisão teórica desenvolvida, não se tem conhecimento de que o problema apresentado tenha sido tratado da mesma maneira em outros estudos. Desta forma, esta pesquisa se caracteriza como exploratória, sendo o principal método de

pesquisa utilizado, de acordo com a classificação de Oates [Oat06], *projeto e criação* (do inglês *design and creation*).

Segundo Oates [Oat06], a estratégia de pesquisa *projeto e criação* têm como finalidade o desenvolvimento de novos produtos de TI que podem ser construções, modelos, métodos ou instanciações. Para este último caso, considera-se como instanciação um trabalho que demonstra que modelos, métodos, gêneros ou teorias podem ser implementados em um sistema de computador. Salienta-se que, para projetos que seguem a estratégia de *projeto e criação* serem considerados de fato uma pesquisa, eles devem demonstrar, segundo Oates, “qualidades acadêmicas, como análise, discussão, justificação e avaliação crítica” [Oat06].

Nesta pesquisa os novos produtos de TI desenvolvidos envolvem um modelo e um protótipo de ferramenta. O modelo proposto é resultado da extensão proposta ao metamodelo SPEM 2.0 e o protótipo de ferramenta implementa o modelo desenvolvido. A avaliação do modelo foi realizada com o auxílio do protótipo de ferramenta desenvolvido e realizada de forma analítica, ou seja, verificando-se os resultados produzidos pelo modelo em decorrência de algumas informações manipuladas em cenários simulados.

A seção a seguir apresenta as etapas que constituíram esta pesquisa:

1.2.1 Etapas de Pesquisa

Resumidamente, esta tese foi realizada através de quatro etapas, as quais estão representadas pela Figura 1 e são descritas a seguir:

Etapa 1: inicialmente, para obter maior conhecimento sobre o assunto de pesquisa, realizou-se um levantamento bibliográfico e estudo do referencial teórico que permitiu aprofundar os conhecimentos sobre processos de software.

Etapa 2: com base nos resultados da etapa 1, um estudo foi realizado sobre os processos de software *Rational Unified Process* – RUP [Kru00], *Object-oriented Process, Environment and Notation* – OPEN [Fir02] e também sobre o metamodelo SPEM 2.0. A partir disto, um conjunto de premissas relacionadas com a consistência foi identificado para os processos de software que são baseados no metamodelo SPEM 2.0. Com base nas premissas definidas, uma nova análise ao metamodelo SPEM 2.0 foi realizada e como resultado para esta análise, identificou-se a necessidade de uma extensão a esse

metamodelo. Desta forma, ainda nesta etapa foram estudados os mecanismos de extensão para metamodelos e o mecanismo de *merge*[1] da UML 2.0 [Omg07b].

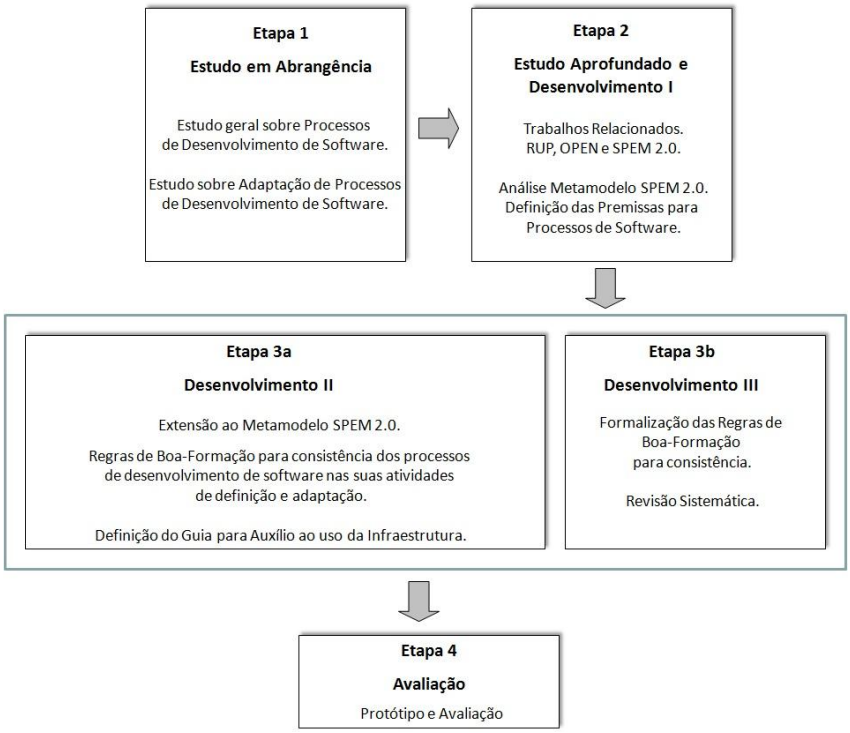

Figura 1 - Etapas da pesquisa

Etapa 3: foi nesta etapa que se desenvolveu grande parte da infraestrutura que viabiliza a consistência dos processos de software que são definidos e adaptados a partir do metamodelo SPEM 2.0. Para fazer isso, uma extensão a esse metamodelo foi proposta envolvendo os seguintes passos: (1) alteração e/ou inclusão de alguns elementos e relacionamentos; (2) inclusão de um conjunto de regras de boa-formação que visam atender as premissas desta pesquisa para consistência dos processos de software e que devem ser utilizadas nas atividades de definição e adaptação de processsos; e (3) alteração das semânticas de alguns mecanismos de adaptação do metamodelo SPEM 2.0 com a proposição de uma análise de impacto para os elementos e relaciomentos desse metamodelo, quando operações de adaptação são conduzidas. Em adição, na etapa 3, todo o conjunto de regras de boa-formação foi formalizada em Lógica de Primeira Ordem e o guia que auxilia o uso da infraestrutura para consistência foi definido. Também foi nesta etapa que foi conduzida uma revisão sistemática sobre o tratamento da consistência dos processos de software para suas atividades de definição e adaptação. O objetivo da revisão foi acompanhar o estado da arte sobre os trabalhos relacionados com esta pesquisa.

[1] Explicado na Seção 2.6.1.3 deste trabalho.

Etapa 4: a etapa final desta pesquisa constituiu-se da construção do protótipo de ferramenta e da avaliação do metamodelo e regras de boa-formação definidos nesta pesquisa. O protótipo desenvolvido oferece suporte automatizado para definição e adaptação de processos de software.

1.3 Contribuições da Tese

As contribuições desta tese para a área de Engenharia de Software estão relacionadas principalmente com o desenvolvimento de uma infraestrutura que suporte a definição e adaptação de processos de software consistentes baseados no metamodelo SPEM 2.0, o qual é tido hoje pela OMG como principal referência na área de processos de software.

A infraestrutura proposta abrange os principais elementos e relacionamentos de um processo de software e aborda aspectos de sua consistência tanto para as atividades de definição quanto para as atividades de adaptação. Desta forma, indica-se uma contribuição para o estado da arte nos estudos sobre a consistência dos processos de software.

Os produtos que integram a infraestrutura para consistência também são considerados como contribuições específicas desta pesquisa e são:

- Extensão ao Metamodelo SPEM 2.0 que inclui e/ou altera alguns elementos, bem como alguns relacionamentos do metamodelo SPEM 2.0. A principal contribuição do metamodelo proposto é a redefinição de alguns relacionamentos do metamodelo SPEM 2.0 que geram inconsistências para os processos de software.
- Conjunto de regras de boa-formação para consistência que são utilizadas nas atividades de definição e adaptação dos processos de software. A principal contribuição das regras é guiar a construção (através das atividades de definição e adaptação) de processos de software consistentes. Pode-se apontar também como uma contribuição a análise de impacto definida para os mecanismos de adaptação do metamodelo SPEM 2.0 que permite identificar todos os elementos e relacionamentos afetados durante as operações de adaptação realizadas sobre um processo de software. A principal contribuição da análise de impacto proposta é garantir que as dependências de um processo de software serão respeitadas durante uma operação de adaptação, evitando assim, que inconsistências sejam geradas no processo resultante.

- Protótipo de ferramenta que implementa: (1) a extensão do metamodelo proposta nesta pesquisa incluindo o conjunto total de regras de boa-formação para consistência de processos (definição e adaptação dos processos); (2) mecanismo de funcionamento do repositório de conteúdos do metamodelo SPEM 2.0; (3) visões gráficas para um processo de software; (5) análise de impacto para os mecanismos de adaptação; e (5) um mecanismo para verificação de processos de software. A principal contribuição do protótipo de ferramenta desenvolvido é auxiliar o uso e avaliação da infraestrutura de consistência proposta nesta pesquisa. Além disso, o suporte automatizado para esta infraestrutura é considerado de alta importância, uma vez que, como mencionado acima, a quantidade e complexidade de informações envolvidas em um processo de software construído a partir da extensão proposta ao metamodelo SPEM 2.0 é elevada.

Além das contribuições específicas dos produtos que integram a infraestrutura de consistência, também são definidos nesta pesquisa um conjunto de premissas para a definição de processos de software consistentes e um guia para definição e adaptação dos processos de software. O guia elaborado especifica um fluxo de atividades a ser seguido para a definição e adaptação de processos utilizando a infraestrututra apresentada neste trabalho. A contribuição mais expressiva deste guia é facilitar o uso da solução aqui proposta, uma vez que devido a grande quantidade de elementos, relacionamentos e regras de boa-formação da extensão definida para o metamodelo SPEM 2.0, seu uso se torna uma tarefa não trivial.

1.4 Estrutura da Tese

Este documento está organizado da seguinte forma: o capítulo 2 apresenta a base teórica da área e descreve o funcionamento do metamodelo SPEM 2.0; o capítulo 3 discorre sobre o aspecto da consistência em processos de software e apresenta o estado da arte sobre este assunto; o capítulo 4 descreve a extensão proposta ao metamodelo SPEM 2.0; o capítulo 5 descreve o guia para definição e adaptação de processos definidos nesta pesquisa; o capítulo 6 apresenta a formalização das regras de boa-formação; o capítulo 7 apresenta a avaliação da solução proposta nesta pesquisa baseado em cenários de uso; e, por fim, no capítulo 8 são apresentadas as considerações finais seguidas das referências bibliográficas.

2 PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE

Este capítulo descreve a introdução à área de estudo e também traz alguns conceitos utilizados no contexto desta pesquisa. Em seguida, um detalhamento do metamodelo SPEM 2.0 é apresentado.

2.1 Introdução

No contexto da computação, um processo é uma tarefa em execução inserida em um dispositivo computacional [Ost87]. No entanto, não existe um consenso entre os autores no tocante a definição de um processo de software. Adicionalmente, existem também diferenças na nomenclatura utilizada para o termo na área de pesquisa, já que alguns autores utilizam como sinônimo para processo de software os termos método ou metodologia.

Segundo Jacobson *et al.* [Jac01], um processo de software é o conjunto completo de atividades necessárias para transformar requisitos de usuários em produtos de software. Como as organizações geralmente consideram o desenvolvimento de um produto de software como um projeto, então, um processo pode ser considerado como uma seqüência de passos que um projeto pode seguir para desempenhar alguma tarefa.

Para [Fug00], um processo de software é um conjunto de atividades e resultados associados que levam ao desenvolvimento de um produto de software. Ainda, segundo Fuggetta [Fug00], o processo de software pode ser definido como um conjunto coerente de políticas, estruturas organizacionais, tecnologias, procedimentos e produtos de trabalho que são necessários para compreender, desenvolver e manter um produto de software.

Com relação aos estudos que utilizam o termo método ou metodologia, apresentam-se [Sel03], [Ser04], [Per05a], [Bur08] e [Kor10]. De acordo com Perez e Sellers [Per05a], uma metodologia é a especificação do processo a ser seguido, dos produtos de trabalho a serem gerados, das pessoas e das ferramentas envolvidas durante um esforço de desenvolvimento de software. Para Sellers [Sel03], uma metodologia (ou método) inclui aspectos de processo e descrições dos produtos de trabalho que serão gerados na construção e/ou manutenção de um produto de software. De acordo com Kornyshova *et al.* [Kor10], por sua vez, uma metodologia é um conjunto de idéias,

abordagens, técnicas e ferramentas usadas por analistas de sistemas para a construção e/ou manutenção de um produto de software.

Através do exposto acima, pode-se observar que, embora existam vários termos e definições na presente área de pesquisa, não existem diferenças significativas para os conceitos apresentados. Assim, deste ponto em diante, assume-se, nesta pesquisa, os termos *processo de software*, *metodologia* e *método* como sinônimos. Ainda no contexto da solução apresentada neste estudo, o termo processo de software será utilizado.

Faz-se necessário também definir, neste ponto, os termos utilizados no presente estudo para os principais elementos de um processo de software. Isso porque, mediante a existência de vários processos de software na literatura, nem sempre o nome dos elementos são os mesmos, sendo comum termos diferentes terem o mesmo significado. Nesta pesquisa, assume-se a utilização dos termos apresentados no metamodelo SPEM, na sua versão 2.0 [Omg07a], que são *Atividade*, *Artefato*, *Papel e Tarefa.* Contudo, como outros processos de software são utilizados em alguns pontos deste trabalho, a Tabela 1 apresenta a correlação entre os termos utilizados nos processos RUP, OPEN e *OpenUP* com os termos do metamodelo SPEM 2.0.

Tabela 1 - Principais terminologias utilizadas nos processo de software

SPEM 2.0	**Artefato**	**Papel**	**Atividade**	**Tarefa**
RUP	Artefato	Papel	Fase / Iteração / *Workflow Detail*	Atividade
OPEN	Produto de Trabalho	Produtor	Atividade / Técnica / Workflow / *Stage*	Tarefa
OpenUP	Produto de Trabalho	Papel	Fase / Iteração / Atividade **/** Tarefa	Tarefa

Durante a descrição dos trabalhos relacionados e definições da literatura apresentadas no texto que segue, correlações entre a nomenclatura original dos estudos e a terminologia utilizada nessa pesquisa, também serão realizadas.

2.2 Processos de Software nas Organizações de TI

Na literatura, muitos estudos indicam que o estabelecimento de um ou mais processos de software nas organizações de TI tem se tornado cada vez mais comum. Um dos motivos disso é que existem fortes indícios que a qualidade do produto de software está relacionada com a qualidade do processo utilizado na sua construção [Yoo01] [Xu05] [Kan08]. Desta forma, o interesse das organizações de TI é estabelecer um ou mais

processos de software bem definidos buscando atender metas específicas de seus projetos de software.

Além do interesse pelo aumento da qualidade dos produtos de software, outro aspecto a ser considerado é que a adoção de um processo de software torna-se ainda mais importante para organizações de TI que desejam implantar um modelo de qualidade como as normas ISO/IEC 15504 [Iso98], ISO/IEC 12007 [Iso95] ou *Capability Maturity Model Integration* (CMMI) [Chr03]. Isso porque tais modelos possuem seus esforços centrados na definição e uso de processos de software e na constante melhoria destes processos [Fug00].

Nesse sentido, estabelecer um processo de software é uma tarefa não trivial [Fug00]. Basicamente, isto envolve as seguintes atividades [Omg07a]: (1) definição de um processo de software (em inglês *process modeling*); (2) adaptação do processo de software (em inglês *process tailoring*); e (3) execução do processo de software (em inglês *enactment*). Tais atividades ocorrem em instantes diferentes e constituem o ciclo de vida básico de um processo de software, conforme mostrado na Figura 2.

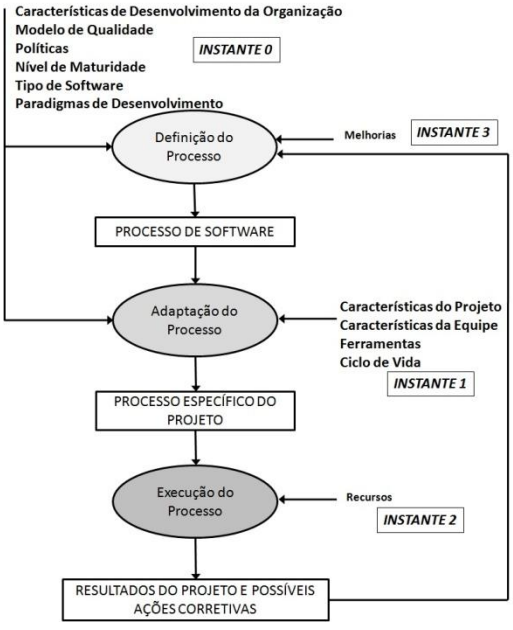

Figura 2 - Ciclo de vida dos processos de software. (Adaptado [MAC00])

Na Figura 2, no instante 0 e 3, em tom mais claro, está a atividade de definição de um processo de software. As entradas para esta atividade, no instante 0, são as características de desenvolvimento da organização, das políticas e de outros fatores tais como se a organização pretende atingir algum nível de maturidade em determinado

modelo de qualidade. A saída para essa atividade é a definição de um ou mais processos de software para a organização.

Em seguida, o instante 1, em tom de cinza intermediário, representa a atividade de adaptação de um processo para um projeto específico. Nesse momento, além das entradas já consideradas para o instante de definição de processos, somam-se as entradas específicas de cada projeto de software, tais como características do projeto, ferramentas disponíveis, características da equipe e do ciclo de vida selecionado para o projeto. Logo, a saída desta atividade é a especificação de um processo adequado ao projeto de software, comumente chamado de processo específico do projeto.

Por fim, em tom mais escuro está o instante 2, que representa a atividade de execução do processo de software. Nesse momento, os recursos são selecionados e associados ao processo específico do projeto. As saídas desta atividade são os resultados do projeto relativos ao processo de software executado (por exemplo, lições aprendidas) e possíveis ações corretivas.

A Figura 2 exibe também que a saída do instante de execução pode ser entrada para uma nova atividade de definição do processo de software, o que caracteriza o instante 3 do ciclo de vida do processo. Isso ocorre, porque melhorias podem ser realizadas, a qualquer momento, em um processo de software. Em verdade, todo processo de software precisa ser monitorado na execução dos projetos de software para que possíveis falhas não ocorram. A importância da constante melhoria de um processo de software pode ser possivelmente notada pela quantidade de trabalhos disponíveis na literatura especializada, a qual é chamada de Melhoria de Processos de Software (em inglês *Software Process Improvement* – SPI).

Na seção seguinte, cada uma das atividades do ciclo de vida de um processo de software será apresentada. Como o foco desta pesquisa concentra-se nas atividades de definição e adaptação de processos de software, estas serão as atividade mais detalhadas. Torna-se importante também ressaltar que os aspectos relacionados com a melhoria dos processos de software não serão tratados nesta pesquisa.

2.3 Definição de Processos de Software

O primeiro passo para a definição de um processo de software é a especificação de suas informações. Isto envolve escolher um conjunto de elementos tais como

atividades, tarefas, produtos de trabalho e papéis, combinando e organizando estes elementos em fluxos de trabalho para a construção ou manutenção de um produto de software [Jac01]. Nesse estágio, deve-se considerar as tecnologias utilizadas pela organização onde o processo será utilizado, os tipos de software desenvolvidos, o domínio de aplicação, o grau de maturidade (ou capacitação) da equipe em engenharia de software, as características próprias da organização e as características dos projetos e das equipes [Hum89], [Mac00] e [Roc01].

Tipicamente, para definir um processo de software as organizações de TI podem se utilizar de processos *off-the-shelf,* tais como RUP e OPEN, os quais estão disponíveis na literatura e especificam as melhores práticas de engenharia de software. Os referidos processos podem ser adotados por completo, ou podem ainda, ser adaptados de acordo com as características da organização. Faz-se possível, também, que uma organização baseie-se em mais de um desses processos mencionados para especificação de seu ou, ainda, que especifique seu processo próprio, através de consultorias ou de um grupo de processos de software interno.

Independente da estratégia escolhida para a definição do processo, além da especificação de suas informações, torna-se necessário também a sua modelagem. Modelar um processo de software consiste em representar as informações deste, utilizando alguma linguagem de modelagem de processos [Rib02]. Uma linguagem de modelagem de processos (em inglês *Process Modeling Language* – PML) tem como objetivo apoiar a representação de diversos conceitos que caracterizam um processo de software [Cug98].

Atualmente, uma grande quantidade de linguagens de modelagem de processos está disponível na literatura, mas, nos últimos anos, em uma tentativa de padronização destas linguagens muitos pesquisadores e profissionais da área têm adotado o uso da UML como PML [Hsu08]. Já em 1999, Franch e Ribó [Fra99] definiram a UML como uma linguagem emergente que havia despertado um grande interesse por parte da indústria e academia e estava se tornando, de fato, um padrão na área da modelagem de processos.

Nesse sentido, a popularização da UML, como linguagem para modelagem de processos, deve-se principalmente ao fato de importantes processos terem sido modelados, utilizando essa linguagem. Esse é o caso, por exemplo, dos processos RUP e OPEN, os quais possuem metamodelos de processos representados com diagramas de classe da UML [Kru00] e [Fir02]. Outro fato que contribuiu para o aumento do uso da UML

como PML foi a definição do metamodelo de processos SPEM [Omg02] e [Omg07a]. Este metamodelo, que teve sua primeira versão publicada em 2002, encontra-se hoje na versão 2.0 e, como já dito anteriormente, é tido como referência para a definição de processos de software.

Uma vez que o metamodelo SPEM 2.0 será utilizado nesta pesquisa para a modelagem de processos de software, a Seção 2.6 abordará alguns conceitos sobre metamodelagem e detalhará o metamodelo SPEM 2.0.

2.3.1 Estruturação dos Processos de Software nas Organizações

Seguindo com o entendimento acerca da atividade de definição dos processos de software, é importante considerar que, torna-se também necessário determinar a forma como isso será feito em uma organização de TI. Tipicamente, organizações podem optar por definir apenas um processo de software (conhecido como processo padrão de desenvolvimento de software) que servirá como base para todos os projetos ou podem ainda seguir a abordagem proposta pela área de pesquisa denominada Engenharia de Método (em inglês *Method Engineering*). Nesta área, os autores estabelecem a criação dos processos de software a partir de um repositório de fragmentos de processo e fragmentos de produto (conforme Seção 2.3.1.2). Cada um destes fragmentos é responsável por descrever somente uma parte do processo de software.

2.3.1.1 Processo Padrão de Desenvolvimento de Software

Um processo padrão de desenvolvimento de software é o processo que deve servir como referência para guiar a execução de todos os projetos de software dentro de uma organização. De acordo com Ginsberg e Quinn [Gin95], este é o meio pelo qual a organização expressa os requisitos que todos os processos de software dos projetos devem atender. Assim sendo, a implantação de um processo padrão apresenta vantagens, como:

- Redução dos problemas relacionados a treinamento, revisões e suporte de ferramentas [Hum89];
- Maior facilidade em medições de processo e qualidade [Hum89];
- Maior facilidade de comunicação entre os membros da equipe [Xu05];
- Melhor desempenho, previsibilidade e confiabilidade dos processos de trabalho [Xu05].

Além das vantagens acima, outro incentivo para definição de um processo padrão é que este é um dos requisitos essenciais para obtenção de alguns níveis de maturidade de importantes modelos de qualidade tais como ISO/IEC 15504 e CMMI. Segundo Borges e Falbo [Bor01], esses modelos definem um processo padrão como um ponto base a partir do qual um processo especializado poderá ser obtido de acordo com as características de um projeto de software específico.

Nesse sentido, um cenário bastante comum nas organizações de TI que utilizam um processo padrão é a utilização de processos já estabelecidos e difundidos na literatura (processos *off-the-shelf*). Como já mencionado anteriormente, essas organizações, muitas vezes adotam tais processos por completo ou fazem adaptações de acordo com suas características de desenvolvimento.

2.3.1.2 Engenharia de Método

A principal abordagem da área de engenharia de método é a decomposição de um processo de software em partes modulares conhecidas na bibliografia especializada, como fragmentos ou *chunks* de método (em alguns artigos chamados de fragmentos ou *chunks* de processo). De acordo com Sellers [Sel03], esses fragmentos ou *chunks* ficam armazenados em um repositório e são disponibilizados para serem selecionados durante a criação de qualquer processo de software.

O termo fragmento de método (ou fragmento de processo) foi introduzido nos estudos da área por Harmsen *et al*. em [Har94]. Os autores definiram um fragmento de método como *uma descrição de um método de engenharia ou qualquer parte coerente deste método*. Um fragmento de método é classificado em dois tipos [Har94] e [Bri96]: os fragmentos de processo e os fragmentos de produto. Os fragmentos de processo descrevem as ações do processo de software e podem ser representados pelas atividades e tarefas. Já os fragmentos de produto representam as estruturas dos produtos de trabalho de um processo de software. *Deliverables*, diagramas e modelos constituem exemplos de fragmentos de produto.

Analogamente aos fragmentos de método, alguns autores, em seus respectivos estudos [Rol96], [Rol98], [Pli98], [Ral99], [Ral01a], [Ral01b], [Ral04] e [Sel08], definiram o termo c*hunk* de método (ou *chunk* de processo). Um *chunk* de método é *uma combinação de uma parte processo (também chamada de guideline) com uma parte produto* [Sel08]. A grande diferença para os fragmentos de método é que um *chunk* de método apresenta

um forte acoplamento entre produto e processo. De acordo com *Sellers et al.* [Sel08], a ligação entre produto e processo do *chunk* de método é uma abstração de todos os tipos de ligação (produção, modificação, etc.) que a parte processo realiza sobre a parte produto de um processo de software.

Usualmente, ambos (fragmentos e *chunks* de métodos) são modelados utilizando um metamodelo. Muitos trabalhos, como [Ral01b], [Per05a], [Hug09] e [Kor10], propõem soluções para esta modelagem, e há também alguns autores que utilizam o metamodelo SPEM 2.0 para modelagem de fragmentos e *chunks* de métodos [Puv09].

Os tópicos de pesquisa em engenharia de método são muitos, entre eles, existem estudos que referenciam como os processos devem ser estruturados em uma organização de TI. Alguns trabalhos [Ral01a], [Ral03], [Mir06] e [Sel08] propõem a definição de um processo de software para cada projeto da organização. A ideia é a de que, através das técnicas para recuperação de fragmentos e *chunks* a partir do repositório, apenas sejam selecionados os fragmentos e *chunks* de método necessários a um projeto.

Nessa abordagem, todo projeto inicia com uma etapa de definição de processo em que os fragmentos ou *chunks* de método são selecionados e organizados para atender às necessidades específicas do projeto em questão [Sel08]. Outros autores propõem uma abordagem similar à já apresentada sobre processo padrão. Para isso, esses pesquisadores definem que um processo base deve ser estabelecido, através dos fragmentos e *chunks,* e tem de estar disponível para ser utilizado em todos os projetos de software.

2.4 Adaptação dos Processos de Software

Não existe um processo de software que possa ser genericamente aplicado [Jal02] e [Sel08]. Nesse sentido, conforme os estudos [Roc01], [Jal02] e [Sel08], o mesmo processo de software não pode servir a qualquer tipo de projeto. Dessa maneira, questões relacionadas ao porte da empresa e à cultura organizacional, aos objetivos de projetos específicos, aos recursos disponíveis, às tecnologias, ao conhecimento e à experiência da equipe impõem características aos processos.

Hoje em dia, é mundialmente aceito que um processo de software deve sofrer adaptações para atender as necessidades específicas no contexto dos projetos [Fit03].

Isso porque cada projeto é único em termos de domínio de negócio, requisitos de cliente, tecnologia, entre outros [Xu05].

Adaptar consiste em excluir, modificar ou adicionar novos elementos e/ou relacionamentos a um processo de software. De acordo com os estudos [GIN95], [ISO95], [Wel95], [Iso98], [Yoo01], [Jal02] essa atividade deve gerar um processo de software específico para o projeto cada vez que for executada. Ainda, de acordo com Kellner [Kel96] e Yoon *et al.* [Yoo01], a atividade de adaptação também pode ser considerada como uma atividade de reuso de processos.

Devido a importância da adaptação dos processos de software muitos estudos estão disponíveis na literatura. Tanto autores que propõem o uso de um processo padrão nas organizações quanto os autores que utilizam a abordagem de criação de processos para cada projeto pela seleção de fragmentos e *chunks* de método são unânimes em estabelecer mecanismos de adaptação para os processos de software, visando atender as necessidades específicas de cada projeto.

Quando um processo padrão é estabelecido, o mecanismo de adaptar um processo é comumente conhecido na literatura como *Process Tailoring* [Yoo01], [Jal02] e [Fit03]. Especificamente, sobre este assunto é trivial encontrar estudos na literatura que apresentam mecanismos de adaptação para processos *off-the-shelf*, tais como as pesquisas de [Han05], [Wes05], [Col06] e [Li09] que se baseiam no processo RUP.

Os estudos que relacionam os mecanismos de adaptação e o uso de fragmentos ou *chunks* de processo fazem parte de uma subárea da engenharia de método chamada engenharia de método situacional (em inglês *Situational Method Engineering* - SME).

O principal objetivo em SME é a construção de métodos modulares que representem uma coleção de fragmentos ou *chunks* de método que se interconectem. Segundo Mirbel e Ralyté [Mir06], graças a essa modularidade, os métodos são mais flexíveis e adaptáveis, pois permitem que fragmentos ou *chunks* de métodos sejam adicionados e/ou removidos para cada projeto de software.

Dentro da subárea da engenharia de método, como já mencionado na seção anterior deste estudo, há autores que defendem o uso de um processo base (processo padrão) como ponto de partida para a criação de um processo específico de projeto [Kar01], [Wis04] e [Baj07]. Nesses estudos, a adaptação de processos é conhecida também como *Process Tailoring* ou ainda como *Method Configuration.*

Com relação a isso, cabe destacar que a adaptação de processos é considerada um mecanismo tão importante que os modelos de qualidade ISO/IEC 15504, ISO/IEC 12007 e CMMI também referenciam essa atividade como um requisito para as organizações de software. Além disso, vários processos de software, tais como o RUP, OPEN, OpenUP, assim como metamodelos de processo tais como, OPF e SPEM 2.0, também referenciam as atividades de adaptação. Basicamente, o que fica estabelecido em tais processos e metamodelos é a possibilidade de exclusão de elementos não requeridos, inclusão de elementos requeridos e modificação de elementos existentes para um projeto em particular.

2.5 Execução dos Processos de Software

A execução de processos de software é a atividade do ciclo de vida de processos de software na qual as tarefas modeladas são realizadas tanto pelos desenvolvedores (quando demandam agentes humanos) quanto automaticamente (quando demandam a invocação de ferramentas autônomas). Portanto, essa atividade envolve questões importantes acerca de planejamento, controle, monitoração, garantia de conformidade com o processo modelado, treinamento, segurança e recuperação do processo [Fei93].

2.6 Metamodelos e Modelos

De acordo com Lee *et al.* [Lee02], um metamodelo é um modelo que serve para modelar um outro modelo conceitual, ou seja, é uma forma de descrever como um modelo deve ser modelado. Dessa forma, um metamodelo também pode ser utilizado para modelar metadados, assim como, por exemplo, configuração de um software ou os metadados dos requisitos.

No contexto da definição e adaptação dos processos de software os metamodelos possuem um papel essencial. Isto porque processos construídos a partir de um metamodelo, geralmente, oferecem um alto grau de formalismo e melhor suporte para consistência e customização, uma vez que os conceitos que formam sua base são explicitamente definidos. Segundo Perez e Sellers [Per05b], essa é a maneira de formalizar as ideias e conceitos subjacentes de um processo de software que são importantes para sua checagem de consistência e, também, para possíveis extensões e/ou modificações do processo.

De acordo com *Hug et al.* [Hug09], um metamodelo torna-se necessário para a definição e adaptação dos processos de software pelo fato destes processos serem basicamente formados por conceitos, regras e relacionamentos. Ainda, um metamodelo de processo é necessário para definição de regras que garantam a integridade entre os elementos e relacionamentos em um processo de software, tanto na atividade de definição quanto durante as atividades de adaptação.

O uso de metamodelos também possibita o suporte automatizado para os processos de software, o que é considerado de alta importância devido à quantidade e complexidade de informações envolvidas em um processo de software.

Nesta pesquisa, o metamodelo SPEM 2.0 é utilizado para as atividades de definição e adaptação de processos de software. As próximas seções deste trabalho descrevem a organização do metamodelo SPEM 2.0 em pacotes, detalham o conteúdo de cada pacote e apresentam os mecanimos de adaptação propostos nesse metamodelo.

2.6.1 SPEM 2.0

Conforme já mencionado anteriormente, o SPEM 2.0 é um metamodelo desenvolvido pelo OMG para a definição de processos de desenvolvimento de software e seus componentes. Este metamodelo é expressso em uma linguagem de meta-modelagem fornecida pelo padrão *MetaObject Facility* - MOF [Omg08] que tem sua origem na UML. A Figura 3 apresenta o uso do MOF 2.0 e da UML 2.0 para a modelagem e definição do SPEM 2.0.

Na figura 3, é possível ver as diferentes camadas de instanciação do formalismo usado na especificação do metamodelo SPEM 2.0. No nível M3, encontra-se o MOF 2.0 que é instanciado pelo nível M2 para a definição dos metamodelos SPEM 2.0 e da linguagem UML 2.0, mesmo nível em que o metamodelo SPEM 2.0 estende o metamodelo da linguagem UML 2.0. Finalmente, no nível M1 está uma instância de M2, mostrando a definição do *Method Library* a partir de dois pontos diferentes. Uma definição através do metamodelo SPEM e outra através da utilização de UML *Profiles* definidos especialmente para esse domínio.

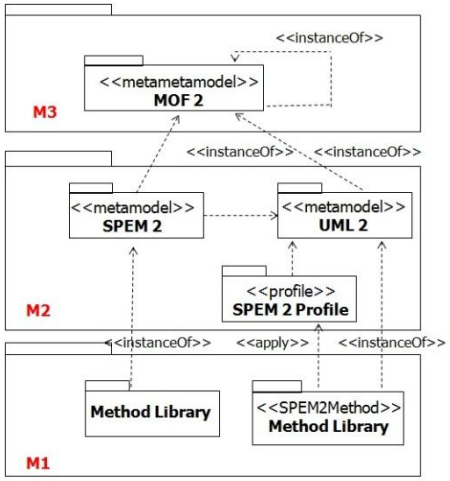

Figura 3 - Camadas de metamodelagem propostas pela OMG

Através da Figura 3 é possível também observar que a definição de qualquer metamodelo de processo de software que se origina do SPEM 2.0 encontrar-se-ia no nível M2, bem como que os modelos de processo derivados a partir desses metamodelos estariam no nível M1. Assim, por exemplo, a inclusão dos metamodelos do processo RUP ou do processo OPEN na Figura 3, os quais possuem elementos que podem ser facilmente relacionados com os elementos do SPEM 2.0, seria realizada no nível M2. Já os modelos desses processos ficariam no nível M1.

2.6.1.1 Estrutura do Processo x Conteúdo

O SPEM 2.0 separa a engenharia dos processos de desenvolvimento de software em dois momentos principais: a criação de um repositório de conteúdo do processo (*Method Content*) e a utilização deste conteúdo (*Process Structure*) em um processo de desenvolvimento de software. *Method content* é tudo aquilo que provê explicações passo-a-passo, descrevendo como os objetivos de um processo de desenvolvimento de software serão alcançados, independente do ciclo de desenvolvimento. Já o *Process Structure* contém os elementos que permitem a definição dos processos de software e também definem os mecanismos que possibilitam o uso dos elementos definidos no *Method Content* nestes processos.

A Figura 4 fornece uma visão de como os conceitos definidos no SPEM 2.0 são posicionados para representar o conteúdo do processo (*Method Content*) e sua utilização (*Process Structure*). O *Method Content* é representado no lado esquerdo da Figura 4 e é formado pelas definições dos *Produtos de Trabalho*, definições dos *Papéis* e definições das *Tarefas*.

No lado direito da Figura 4, encontra-se o *Process Structure* que, como mencionado anteriormente, possibilita a definição dos processos de software, inclusive pela utilização do conteúdo definido no *Method Content*. Por fim, tem-se, ao centro da Figura 4 o elemento *Guidance,* o qual pode estar presente tanto no *Method Content* quanto no *Process Structure*. Esses elementos são os guias, *checklists*, exemplos ou *roadmaps* definidos para os processos de desenvolvimento de software.

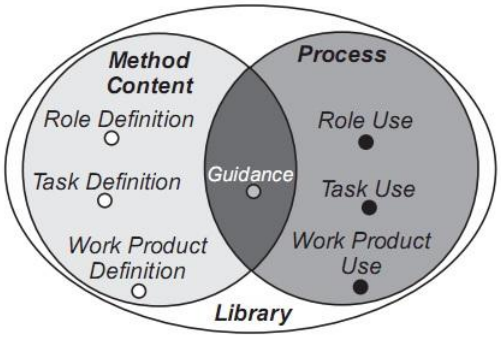

Figura 4 - Divisão entre *Method Content* e *Process Structure*

2.6.1.2 Definição dos Pacotes

O metamodelo SPEM 2.0 é estruturado em sete pacotes principais, conforme mostrado na Figura 5, os quais são compostos aplicando-se o mecanismo package *merge*[2] da UML. Tais pacotes são:

- *Core* - contém as metaclasses e abstrações para a construção da base do metamodelo;
- *Process Structure* - define os conceitos base para modelagem de processos representando sua estrutura estática.
- *Process Behaviour* - pacote que permite uma extensão do metamodelo do SPEM 2.0 para que a execução de um processo possa ser acompanhada.
- *Managed Content* - adiciona conceitos para documentação e descrição textual de um processo.
- *Method Co*ntent - permite que os usuários do SPEM 2.0 criem uma biblioteca com conhecimento reutilizável e independente de processos para uso posterior.
- *Process with Methods* - define novos conceitos e altera outros conceitos já existentes nos pacotes anteriores para integrar processos definidos pelo pacote *Process Structure* com seus conteúdos, definidos pelo pacote *Method Content*.

[2] *Merge* é um relacionamento entre dois pacotes onde o conteúdo do pacote de destino é combinado com o conteúdo do pacote de origem através de especializações e redefinições, quando aplicáveis. [OMG08]

- *Method Plugin* - define os conceitos necessários para criar, gerenciar e manter bibliotecas e processos de software.

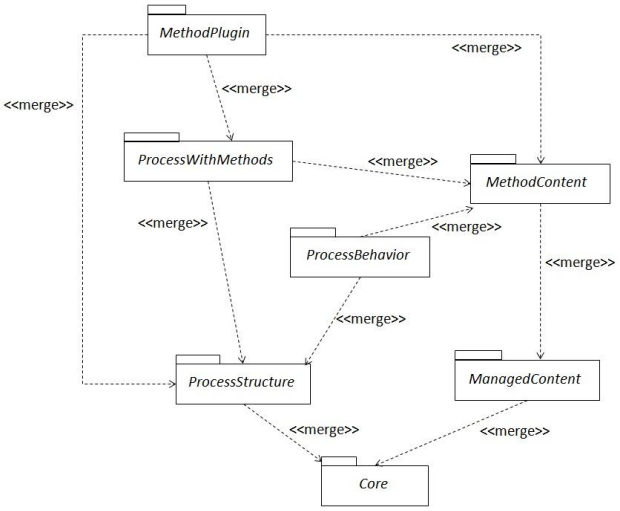

Figura 5 - Estrutura de pacotes do SPEM 2.0

2.6.1.3 Pontos de Conformidade

Segundo sua especificação, o SPEM 2.0 é definido a partir de três pontos de conformidade: SPEM *Complete*, SPEM *Process with Behaviour* and *Content*, e SPEM *Method Content*, descritos a seguir.

SPEM *Complete*

O SPEM *Complete* compreende todos os sete pacotes do metamodelo do SPEM 2.0. Este Ponto de Conformidade é recomendado para àqueles que desejam utilizar todo o metamodelo e suas capacidades. Na Figura 6 é apresentado um espaço de nomes chamado SPEM2-*Complete*, o qual utiliza-se de package *merge* para compor: (1) o LM, um nível de conformidade definido pela UML 2 *Infrastructure Library*; (2) o UML *Profiles*; e (3) os pacotes *Method Plugin*, *Process Behavior*, compondo através de transitividade todos os outros pacotes do SPEM 2.0.

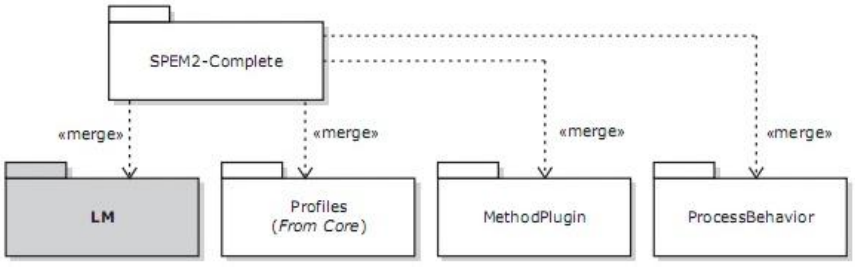

Figura 6 - Ponto de conformidade - *SPEM Complete*

Audiência: Fornecedores em larga escala de ferramentas para bibliotecas de procesos de software e conteúdos.

SPEM *Process with Behavior and Content*

O SPEM *Process with Behavior and Content* compreende quatro pacotes do metamodelo do SPEM 2.0. Na Figura 7 é apresentado um espaço de nomes chamado SPEM2-*Process-Behavior-Content*, o qual utiliza-se de package *merge* para compor: (1) o LM, um nível de conformidade definido pela UML 2 *Infrastructure Library*; e (2) os pacotes *Managed Content* e *Process Behavior* do SPEM 2.0, que por transitividade compõem também os pacotes *Process Structure* e *Core*.

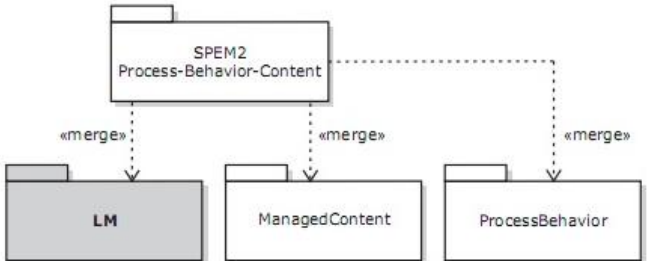

Figura 7 - Ponto de conformidade - *SPEM with Behavior and Content*

Audiência: Fornecedores de ferramentas com foco em compatibilidade com o SPEM 1.0 e modelagem.

SPEM *Method Content*

O SPEM *Method Content* compreende três dos pacotes do metamodelo do SPEM 2.0. Na Figura 8 é apresentado um espaço de nomes chamado SPEM2-*Method-Content*, o qual utiliza-se de package merge para compor: (1) o LM, um nível de conformidade definido pela UML 2 *Infrastructure Library*; e (2) o pacote *Method Content*, compondo através de transitividade os pacotes *Managed Content* e *Core* do SPEM 2.0.

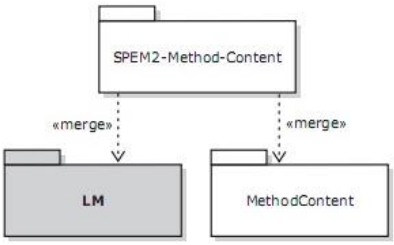

Figura 8 - Ponto de conformidade - *SPEM Method Content*

Audiência: Fornecedores de bases de conhecimento organizacional e responsáveis por documentação.

2.6.1.4 Funcionamento e Principais Classes dos Pacotes SPEM 2.0

Pacote Core

O pacote *Core* é o pacote núcleo do SPEM 2.0. Nesse pacote, as metaclasses e abstrações que constroem a base para todos os outros pacotes do metamodelo são definidas.

Basicamente, o pacote *Core* define duas capacidades para o SPEM 2.0:

- A habilidade para usuários criarem qualificações para diferentes metaclasses do SPEM 2.0 distinguindo elas com diferentes '*kinds'*.
- Um conjunto de metaclasses abstratas para expressar o trabalho nos processos de software. Todas as metaclasses do metamodelo SPEM 2.0 que derivam das metaclasses de trabalho definidas no *Core* tem como objetivo mapear os modelos de comportamento do processo.

As metaclasses definidas no pacote *Core* tem como super classes as metaclasses definidas na UML 2.0 e serão exibidas nas Figuras 9 e 10.

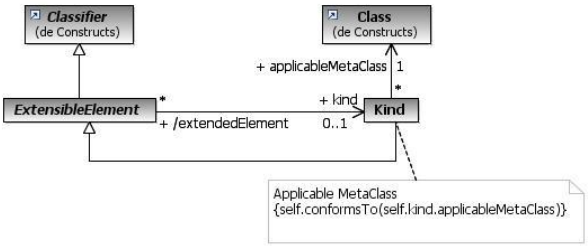

Figura 9 – Metaclasses *ExtensibleElement* e *Kind* definidas no pacote *Core*

A metaclasse *ExtensibleElement,* vista na Figura 9*,* é uma generalização abstrata que representa qualquer metaclasse do metamodelo SPEM 2.0 que pode ter um valor de '*kind'* atribuído, expressando, assim, uma qualificação definida pelo usuário. Cada metaclasse do metamodelo SPEM 2.0 que permite utilizar esta qualificação deriva direta ou indiretamente da metaclasse *ExtensibleElement*.

A metaclasse *kind* que também é mostrada na Figura 9*,* é um especialização da metaclasse *ExtensibleElement.* Esta metaclasse é usada para qualificar outras intâncias de metaclasse do metamodelo SPEM 2.0 com tipos definidos pelo usuário.

Prosseguindo com e explanação sobre as metaclasses do pacote *Core,* têm-se as mostradas na Figura 10. Inicialmente, a metaclasse *ParameterDirectionKind* é de enumeração que representa parâmetros de entrada (*in*), saída (*out*) e também de entrada

e saída (*inout*) para as instâncias das metaclasses e subclasses de *WorkDefinition*. Esses parâmetros são definidos através do atributo *direction* definido para a classe *WorkDefinitionParameter* que é uma generalização abstrata para elementos do processo e representa os parâmetros para as metaclasses e subclasses de *WorkDefinition*.

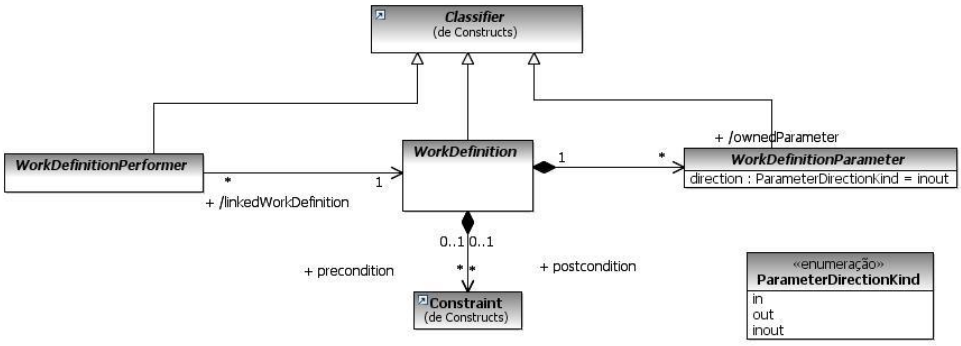

Figura 10 – Metaclasses do pacote *Core*

A metaclasse *WorkDefinition* é uma metaclasse abstrata que generaliza todas as definições de trabalho no metamodelo SPEM 2.0. Nos pacotes *ProcessStructure* e *MethodContent*, respectivamente, essa metaclasse é especializada em atividades e tarefas. No pacote *Core*, a metaclasse *WorkDefinition* pode conter pré e pós condições que são representadas pela metaclasse *Constraint* da UML 2.0. A metaclasse *WorkDefinition* também pode possuir 0 ou vários parâmetros representados pela metaclasse *ParameterDirectionKind* e também pode ser associada a 0 ou várias metaclasses *WorkDefinitionPerformer*.

Por fim, a metaclasse *WorkDefinitionPerformer* é uma metaclasse abstrata que representa o relacionamento de um executor de trabalho para uma metaclasse *WorkDefinition.* Essa metaclasse será especializada em outros pacotes do metamodelo SPEM 2.0 em diferentes tipos de relacionamentos. Ela irá ser especializada, por exemplo, no pacote *Process Structure* em *ProcessPerformer*.

Pacote Process Structure

O pacote *Process Structure* é o pacote que define a base para todos os processos de software no metamodelo SPEM 2.0. Nesse pacote, processos são representados por uma estrutura de *Work Breakdown Element* – WBS que permite o aninhamento de atividades dentro de outras atividades ou, ainda, o aninhamento de outros elementos dentro de uma atividade. Desse modo, os elementos que podem ser

aninhados em atividades são papéis, produtos de trabalho e também várias metaclasses que representam relacionamentos. A Figura 11 e 12 exibem, respectivamente, a taxonomia de metaclasses do pacote *Process Structure* e as principais associações definidas nele.

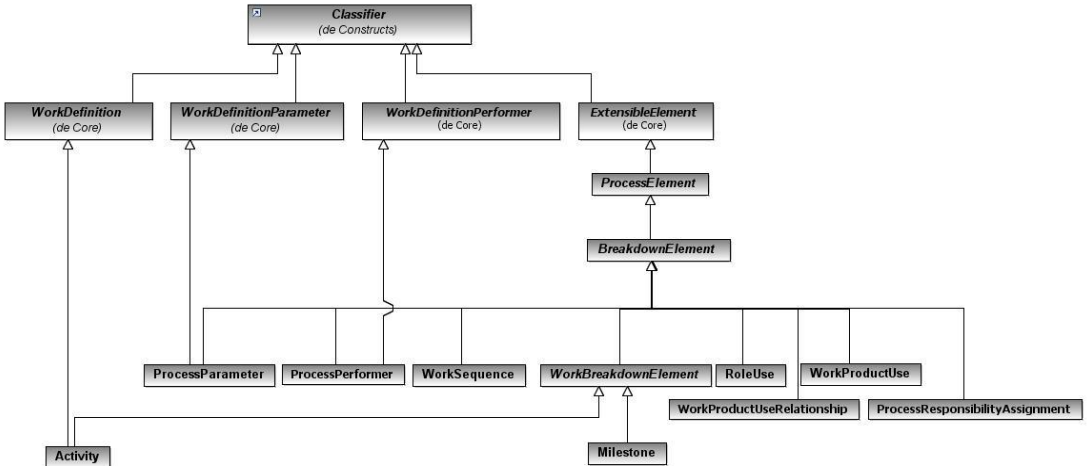

Figura 11 – Taxonomida das metaclasses do pacote *Process Structure*

Na Figura 12 as metaclasses *Activity*, *ProcessPerformer*, *ProcessParameter* e *BreakdownElement* especializam, respectivamente, as metaclasses *WorkDefinition*, *WorkDefinitionPerformer*, *WorkDefinitionParameter* e *ExtensibleElement* do pacote *Core*.

As metaclasses *Process Element*, *BreakdownElement* e *WorkBreakdownElement* são metaclasses abstratas. Especificamente, as metaclasses *BreakdownElement* e *WorkBreakdownElement* é que permitem a representação de um processo através de uma estrutura WBS. A ideia é que a metaclasse *Activity* seja utilizada para instanciar os elementos que representam unidades de trabalho nos processos tais como as atividades e também utilizadas para instanciar os elementos do processo que organizam as unidades de trabalho em definições de tempo tais como as fases, as iterações e o próprio processo. Já os outros elementos deste pacote que são *RoleUse*, *WorkProductUse*, *WorkProductUseRelationship*, *WorkSequence*, *Milestone*, *ProcessPerformer* e *ProcessResponsabilityAssignment* podem ser instanciados dentro das atividades para representar as outras informações do processo de software.

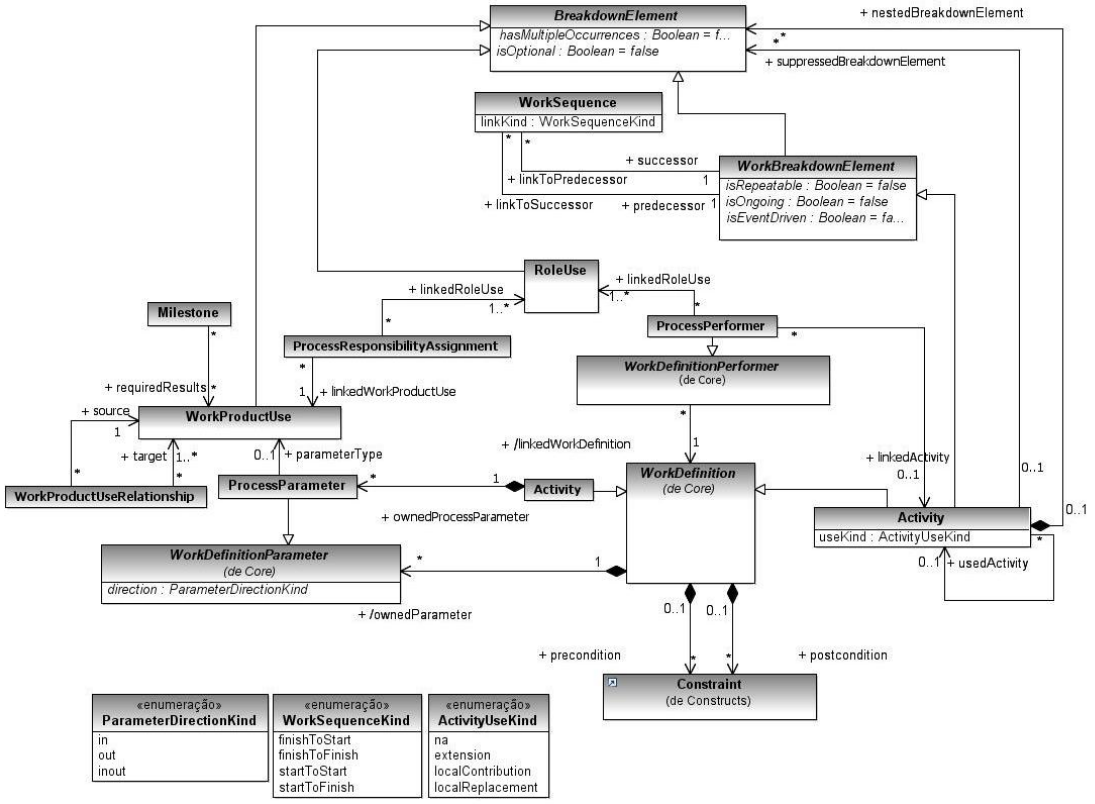

Figura 12 – Metaclasses e associações do pacote *Process Structure*

As metaclasses *Activity*, *RoleUse*, *WorkProductUse* e *Milestone* representam elementos do processo de software. Já as metaclasses *WorkProductUseRelationship*, *ProcessPerformer*, *ProcessResponsabilityAssignment, ProcessParameter* e *WorkSequence* representam os relacionamentos entre esses elementos. *WorkProductUseRelationship* estabelece relações entre *WorkProductUses*; *ProcessPerformer* estabelece a relação entre as atividades e os papéis no processo de software; *ProcessResponsabilityAssignment* estabelece a relação de responsabilidade entre os papéis e os produtos de trabalho; *ProcessParameter* estabelece através do atributo *direction* e da metaclasse de enumeração *ParameterDirectionKind* os parâmetros de entrada e/ou saída para as atividades em termos de produtos de trabalho.

Por fim, a metaclasse *WorkSequence* permite estabelecer sequenciamento entre as atividades utilizando o atributo *linkKind* e a metaclasse de enumeração *WorkSequenceKind*. As relações de sequenciamento permitem estabelecer relações de dependência entre as atividades em que uma atividade depende do início ou fim de outra(s) atividade(s) para poder iniciar ou terminar.

Outros relacionamentos, não são representados por metaclasses, existem no pacote *ProcessStructure*. Estes relacionamentos são o auto-relacionamento *usedActivity*

da metaclasse *Activity* e as relações *suppressedBreakdownElement* e *nestedBreakdownElement* entre as metaclasses *Activity* e *BreakdownElement*. A relação *nestedBreakdownElement* é a que permite o aninhamento de elementos dentro de uma atividade. Já as relações *usedActivity* e *suppressedBreakdownElement* permitem, respectivamente, um mecanismo de reuso e supressão de elementos do processo. Essas relações serão detalhadas na seção 2.6.1.5 deste estudo, pois, embora possam ser usadas na atividade de definição de processos, são também consideradas como mecanismos de adaptação do metamodelo SPEM 2.0.

Pacote Managed Content

O pacote *Managed Content* define os conceitos fundamentais para gerenciar as descrições textuais para os processos (elementos do *ProcessStructure*) e elementos do *Method Content* no SPEM 2.0. Esse pacote introduz a metaclasse abstrata *DescribableElement* que, através do mecanismo de *merge*, serve como uma super classe para elementos de processo definidos no pacote *Process Structure* e também para as metaclasses do pacote *Method Content*. O elemento *DescribableElement* é composto de uma metaclasse *ContentDescription* que permite relacionar descrições textuais para os elementos. A Figura 13 mostra as metaclasses e relações do pacote *Managed Content.*

Na Figura 13, é possível verificar que a metaclasse *DescribableElement* é uma especialização da metaclasse *ExtensibleElement* do pacote *Core*. A metaclasse *DescribableElement* possui uma relação de composição com a metaclasse *ContentDescription* que possui vários atributos relacionados com descrição textual, bem como relação de composição com uma metaclasse denominada *Section*. Isso é o que permite que todas as especializações da metaclasse *DescribableElement* possuam descrições textuais e também que se utilizem do conceito de *Seção*. Na Figura 13, observa-se também que a metaclasse *ProcessElement* (definida no pacote *ProcessStrucure*) aparece como uma especialização da metaclasse *DescribableElement*. Através dessa relação é que todos os elementos do pacote *ProcessStructure* podem possuir descrições textuais.

Existem também as metaclasses *Guidance*, *Category* e *Metric* que especializam a metaclasse *DescribableElement*. A metaclasse *Guidance,* além de ser uma especialização de *DescribableElement,* possui relação com essa metaclasse. Isso permite que guias contendo informação adicional possam ser associados a qualquer elemento *DescribableElement.* As metaclasses *Category* e *Metric*, igualmente a metaclasse *Guidance*, também possuem relação com a metaclasse *DescribableElement.* Tais

metaclasses podem ser usadas, respectivamente, para categorizar os elementos *DescribableElement* e para associar uma ou mais restrições que fornecem medidas para qualquer elemento *DescribableElement.*

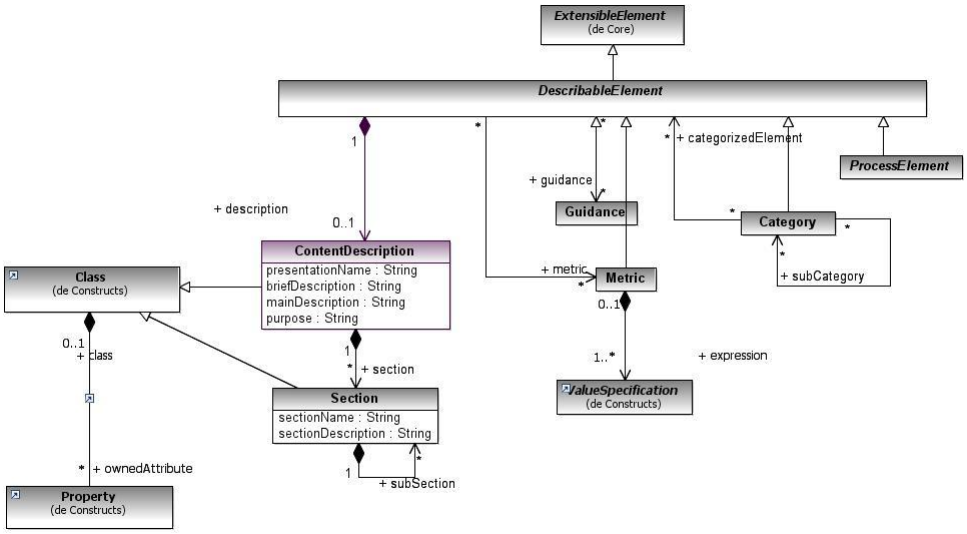

Figura 13 – Metaclasses e associações do pacote *Managed Content*

Pacote Process Behavior

O metamodelo SPEM 2.0 não fornece conceitos para modelagem da execução de processos de software. De acordo com a especificação desse metamodelo, ele apenas define a habilidade para que implementadores escolham uma abordagem de modelagem de comportamento que melhor atendam suas necessidades. Partindo do contexto que esse pacote não é detalhado no metamodelo SPEM 2.0 e que esta pesquisa não considera aspectos de execução de processos, o pacote *Process Behavior* não é descrito neste trabalho em termos de suas classes e relacionamentos.

Pacote Method Content

O pacote *Method Content* define os elementos que formam uma base de conteúdo para ser utilizada em qualquer processo de software. Os principais elementos desse pacote são *Papéis*, *Tarefas* e *Produtos de Trabalho*. As Figuras 14 e 15 exibem, respectivamente, a taxonomia dos elementos do pacote *Method Content* e como as principais metaclasses desse pacote estão relacionadas.

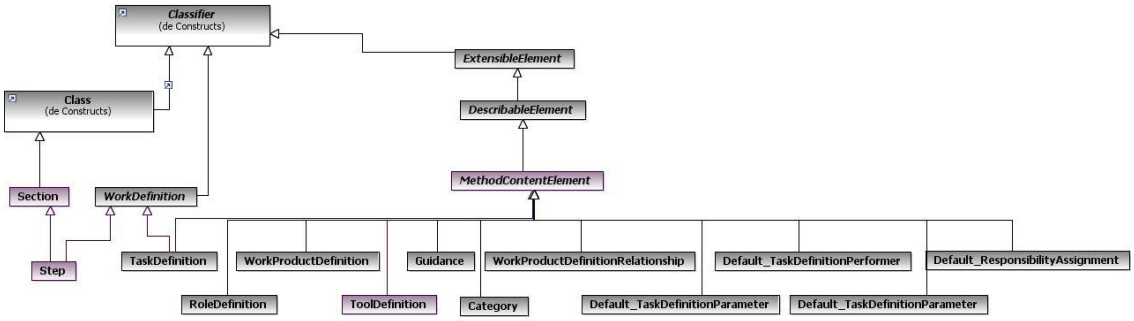

Figura 14 – Taxonomida das metaclasses do pacote *Method Content*

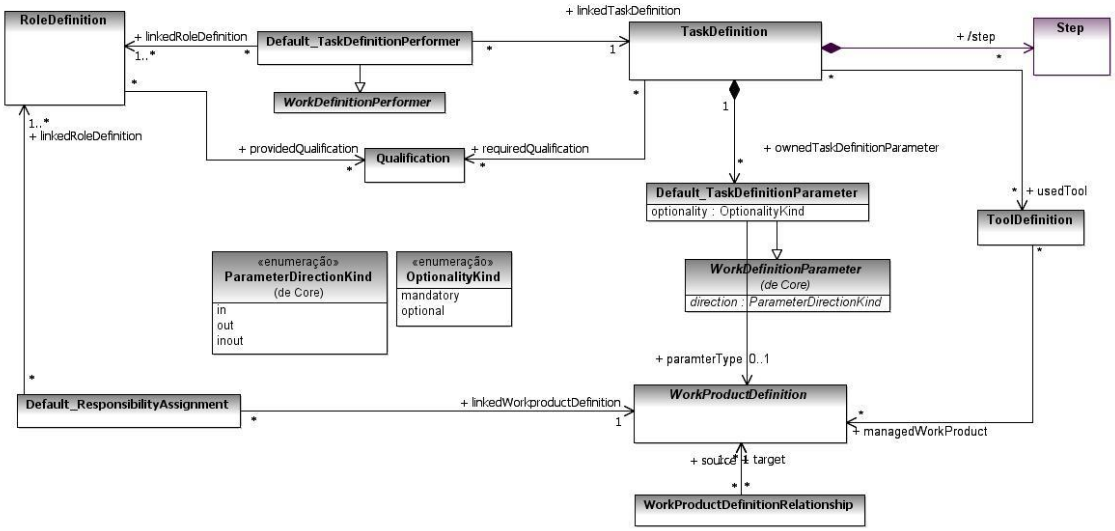

Figura 15 – Metaclasses e associações do pacote *Method Content*

Na Figura 14 especificamente, observa-se que todos os elementos definidos nesse pacote são especializações da metaclasse *MethodContentElement* que, por sua vez, especializa a metaclasse *DescribableElement*. Os elementos introduzidos pelo mecanismo de *merge* neste pacote são *Step*, *TaskDefinition*, *RoleDefinition*, *WorkProductDefinition*, *ToolDefinition*, *Qualification*, *WorkProductDefinitionRelationship*, *Default_TaskDefinitionPerformer*, *Default_TaskDefinitionParameter* e *Default_ResponsabilityAssignment*.

Já a Figura 15 exibe como os elementos do pacote *Method Content* se relacionam. Basicamente, a metaclasse *TaskDefinition* pode ser considerada como elemento central do pacote. Essa metaclasse possui relação com a metaclasse *RoleDefinition* através das metaclasses *Default_TaskDefinitionPerformer* e *Qualification*; relação com a metaclasse *WorkProductDefinition* através da metaclasse *Default_TaskDefinitionParameter*; relação com a metaclasse *ToolDefinition* e relação de composição com a metaclasse *Step*. Os outros relacionamentos desse pacote são o auto-

relacionamento para a metaclasse *WorkProductDefinition* representado pela metaclasse *WorkProductDefinitionRelationship* e o relacionamento entre a metaclasse *WorkProductDefinition* e *RoleDefinition* estabelecido pela metaclasse *Default_ResponsabilityAssignment.*

Pacote Process with Methods

O pacote *Process with Methods* liga os conceitos definidos no pacote *Process Structure* com os conceitos definidos no pacote *Method Content*. Assim, é possível que processos de software sejam montados, utilizando os conceitos pré-definidos no *Method Content.* A figura 16 mostra a taxonomia das metaclasses do pacote *Process with Methods.* Nessa Figura, é possível observar que as novas metaclasses incluídas neste pacote são *TeamProfile*, *CompositeRole, MethodContentUse* e *TaskUse*. Além disso, as metaclasses *RoleUse* e *WorkProductUse* que já haviam sido definidas no pacote *Process Structure* são, agora, especializações da nova metaclasse *MethodContentUse*.

As principais relações das metaclasses no pacote *Process with Methods* são mostradas na Figura 17. Nesta Figura, novamente percebe-se que o elemento tarefa é o elemento central. A metaclasse *TaskUse* assume praticamente as mesmas relações daquelas descritas no pacote *MethodContent*. A grande diferença nesse pacote é que a nova metaclasse *TaskUse* assume relações com elementos definidos no pacote *Process Structure*. Além disso, relacionamentos novos são estabelecidos no pacote *Process with Methods.* Esses relacionamentos ligam os elementos *TaskUse*, *WorkProductUse* e *RoleUse*, respectivamente, com os elementos *TaskDefinition*, *WorkProductDefinition* e *RoleDefinition* do pacote *Method Content*.

Através das relações acima entre as metaclasses do tipo *Use* e *Definition* é que um processo reutiliza os elementos definidos no *Method Content*. A idéia do metamodelo SPEM 2.0 é que um repositório de conteúdo seja montado através do pacote *Method Content* e que as estruturas dos processos de software (tal como o fluxo de atividades e tarefas) sejam estabelecidas com os conceitos definidos no *Process Structure.* A junção desses conceitos é realizada justamente no pacote *Process with Methods,* o qual, através do mecanismo de *merge*, possui ligações com ambos os pacotes *Method Content* e *Process Structure* (conforme Figura 5).

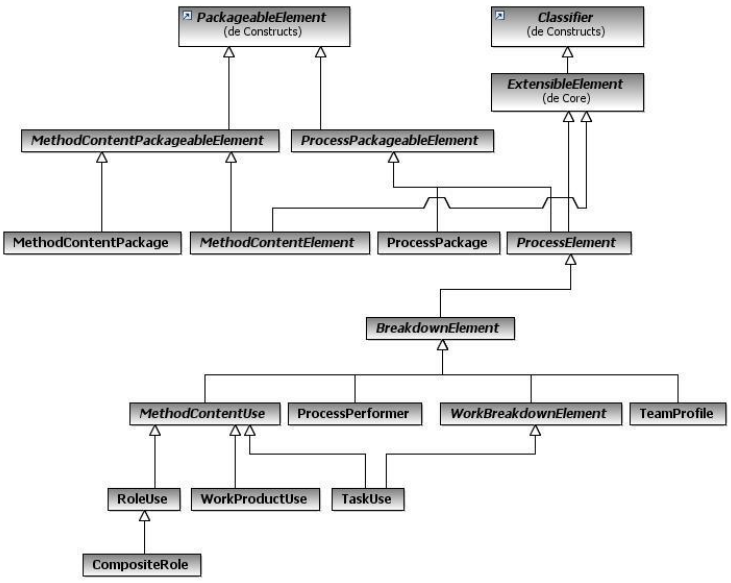

Figura 16 – Taxonomida das metaclasses do pacote *Process with Methods*

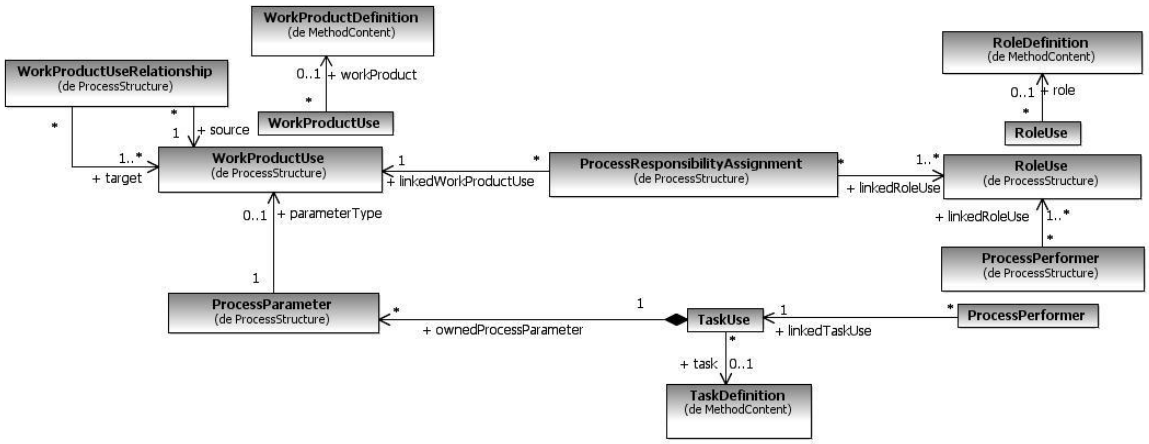

Figura 17 – Metaclasses e associações do pacote *Process with Methods*

Pacote Method Plugin

O pacote *Method Plugin* define a capacidade para gerenciar bibliotecas de *Method Content* e Processos. Esse pacote endereça o interesse de estabelecer grandes bibliotecas por definir *Method Plugins* and *Method Configurations*. Adicionalmente, o pacote *Method Plugin* define mecanismos de variabilidade e extensibilidade para o *Method Content* e Processos. Tais mecanismos são considerados pelo metamodelo SPEM 2.0 como mecanismos de adaptação, os quais permitem que processos sejam criados sobre demanda ou adaptados de acordo com o contexto dos projetos de software.

As novas metaclasses incluídas no pacote *Method Plugin* são mostradas na Figura 18 que exibe a taxonomia de metaclasses para esse pacote. Observa-se ainda que novas metaclasses são incluídas nesse metamodelo e também que há novas definições

de especialização. As principais novas metaclasses são as metaclasses *MethodConfiguration*, *MethodPlugin*, *MethodLibrary* e *VariabilityElement* e estão envolvidas na definição de bibliotecas e também com o mecanismo de adaptação proposto nesse pacote,

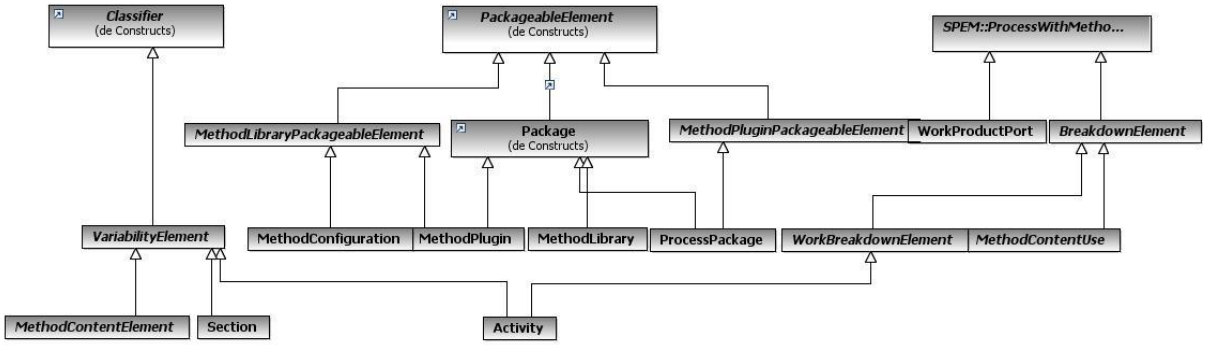

Figura 18 – Taxonomida das metaclasses do pacote *Method Plugin*

As relações entre as metaclasses *MethodConfiguration*, *MethodPlugin* e *MethodLibrary* podem ser vistas no diagrama de classes exibido na Figura 19. Nessa Figura, observa-se que uma biblioteca representada pela metaclasse *MethodLibrary* é uma composição da metaclasse *MethodPlugin* e da metaclasse *MethodConfiguration*.

Um *Method Plugin* reprenta um *container* físico para *Content* (definido com metaclasses do pacote *Method Content*) e *Process Packages* (definido com metaclasses do pacote *Process Structure,* usando ou não as metaclasses do pacote *Method Content*). Na Figura 19, a metaclasse *MethodPlugin* é uma composição das metaclasses *ProcessPackage* e *MethodContentPackage*, as quais, respectivamente, guardam as metaclasses do pacote *Process Structure* e do pacote *Method Content*.

Um *Method Configuration* é uma coleção de *Method Plugins* selecionada e seus subconjuntos de *MethodContentPackages* e *ProcessPackages*. Esta metaclasse, *MethodConfiguration*, é utilizada no SPEM 2.0 para definir subconjuntos lógicos dentro de uma biblioteca (*MethodLibrary*).

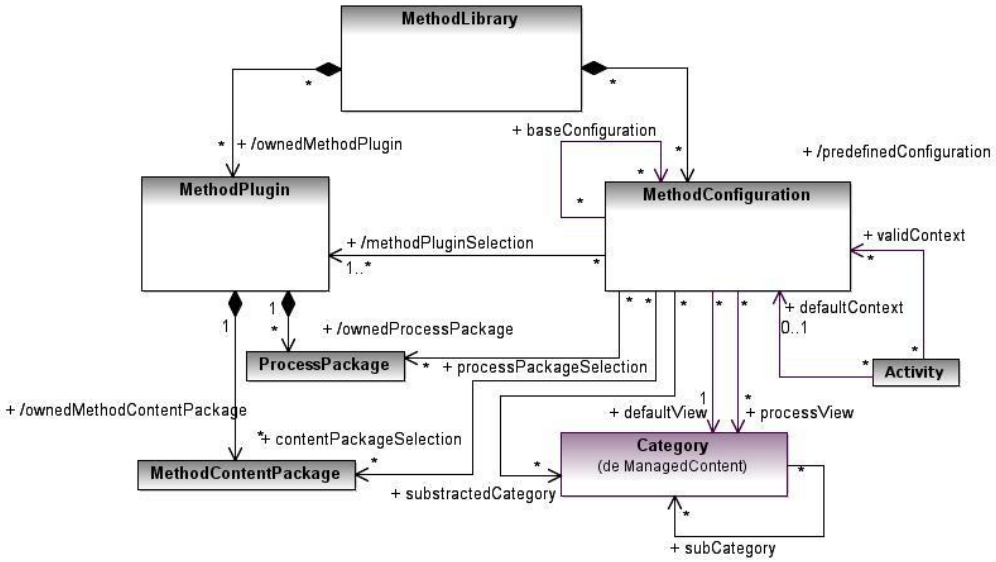

Figura 19 – Associações entre as metaclasses *Method Library, Method Plugin* e *Method Configuration*

A Figura 20 exibe a definição da metaclasse *VariabilityElement* no pacote *MethodPlugin*. Essa metaclasse e suas relações são considerados como mecanismos de adaptação do metamodelo SPEM 2.0 e, dessa forma, serão detalhados na próxima seção deste trabalho.

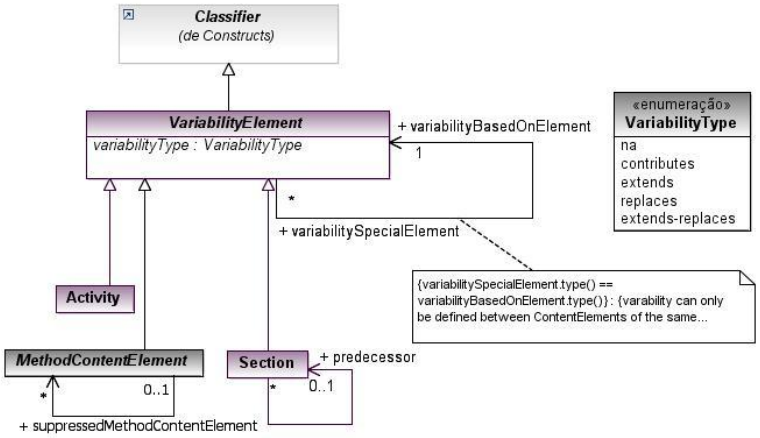

Figura 20 – Metaclasse *Variability* definida no pacote *Method Plugin*

2.6.1.5 Adaptação de Processos no SPEM 2.0

A adaptação de processos de desenvolvimento de software é tratada em dois pacotes do metamodelo SPEM 2.0. No pacote *Process Structure,* isso é feito, através dos seguintes relacionamentos: *usedActivity* e *suppressedBreakdonwElement*. Já no pacote *Method Plugin* isso é definido pela metaclasse *VariabilityElement* e seus relacionamentos.

Inicialmente, no pacote *Process Structure* o auto-relacionamento definido para o elemento *Activity* chamado *usedActivity* permite o reuso do conteúdo definido para uma atividade em outra atividade. Dessa forma, torna-se possível herdar a estrutura definida para uma atividade em termos de seus elementos aninhados em uma segunda atividade. O metamodelo SPEM 2.0 define alguns tipos de herança para o relacionamento usedActivity, os quais são estabelecidos pelo atributo *useKind* da metaclasse *Activity* e pela metaclasse de enumeração *ActivityUseKind*. Esses tipos de herança são:

- *na*: este é o valor *default* do atributo *useKind* de todas as atividades. Esse valor é usado para atividades que não instanciam a relação *usedActivity*.
- *extension*: este mecanismo permite reutilizar, dinamicamente, as subestruturas (elementos aninhados pela relação de composição) de uma atividade em outras atividades. Dessa forma, a atividade que é associada à outra, pela relação *usedActivity* e possui o valor para o atributo *useKind* igual a *extension* herda todo conteúdo dessa atividade.
- *localContribution*: este mecanismo deve ser sempre utilizado em conjunto com o mecanismo de extensão (*extension*). Ele permite que adições locais (contribuições) sejam feitas em atividades que estão sendo herdadas através do mecanismo de extensão. Uma atividade A poderia, por exemplo, herdar toda estrutura da atividade B pelo mecanismo de extensão (*extension)*. Contudo, poderia ser necessário fazer adições locais (contribuições) para a atividade A, através da relação *localContribution*. Para fazer isso, devem existir subatividades dentro das atividades A e B. Na subatividade de A (por exemplo, A.1), devem ser definidas as contribuições locais e, nesse momento, ser apontado para subatividade de B (por exemplo B.1). O resultado em A.1 será todo conteúdo de B.1 somado aos elementos pré-definidos em A.1.
- *localReplacement:* este mecanismo deve ser sempre utilizado em conjunto com o mecanismo de extensão (*extension*). Ele permite que substituições locais sejam feitas em atividades que estão sendo herdadas, através do mecanismo de extensão. Uma atividade A poderia, por exemplo, herdar toda estrutura da atividade B pelo mecanismo de *extension*. Contudo, poderia ser necessário fazer substituições locais para partes da atividade A, através da relação *localReplacement*. Para fazer isso, devem existir subatividades dentro das atividades A e B. Na subatividade de A (por exemplo, A.1) deve ser definido o conteúdo local que vai substituir o conteúdo sendo herdado e, nesse momento, ser apontado para subatividade de B (por exemplo B.1). O resultado em A.1

será seu próprio conteúdo, já que tal conteúdo foi definido para substituir o conteúdo de B.1.

Além dos mecanismos acima, a relação *supressedBreakdownElement* também é definida entre as metaclasses *Activity* e *BreakdownElement*. De acordo com a especificação do metamodelo SPEM 2.0, essa relação também deve ser sempre utilizada em conjunto com o mecanismo de extensão (*extension*). Ela permite suprimir elementos de uma atividade após o mecanismo de extensão (*extension*) ter sido realizado. Após uma atividade A, por exemplo, ter herdado todo conteúdo da atividade B, fazendo ou não contribuições e/ou substituições locais, é possível ainda que elementos sejam suprimidos da atividade A.

Para facilitar o entendimento dos mecanismos explicados acima, a Figura 21 mostra um exemplo. Nesta figura é mostrado que a atividade *Process 2* estende a atividade *Process 1* realizando algumas contribuições, substituições e supressão de elementos. O resultado, após a interpretação das relações é mostrado na parte direita da Figura 21.

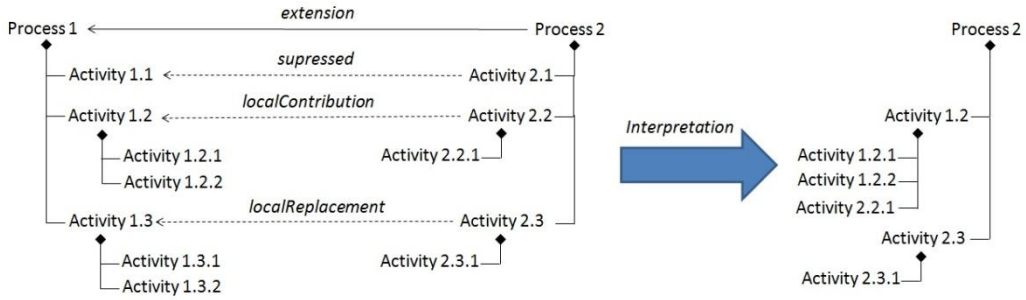

Figura 21 – Exemplo de aplicação dos relacionamentos *usedActivity* e *supressedBreakdownElement*

No pacote *Method Plugin* a adaptação dos processos é definido através de mecanismos de variabilidade. De acordo com a especificação do metamodelo SPEM 2.0, estes mecanismos fornecem flexibilidade para definir diferentes variantes do conteúdo dos processos e dos próprios processos de software.

Como já mostrado na Figura 20 a metaclasse *VariabilityElement* possui o atributo *variabilityType* que pode assumir os valores definidos na metaclasse de enumeração *VariabilityType*. A metaclasse *VariabilityElement* é uma metaclasse abstrata e os elementos que especializam esta metaclasse e se utilizam do mecanismo de variabilidade são: *Activity* e *MethodContentElement* (ou seja, todos elementos do pacote *Method Content*). O auto-relacionamento definido na metaclasse *VariabilityElement* e uma

restrição em OCL definem que um elemento só poderá variar com base em outro elemento do mesmo tipo. Um *WorkProductDefinition*, por exemplo, só poderá se associar com um dos valores de *VariabilityType* com outro *WorkProductDefinition*. Os tipos de variabilidade do *VariabilityType* são os seguintes:

- *contributes*: este mecanismo permite que contribuições sejam realizadas em um elemento do repositório ou uma atividade do processo sem modificar seu conteúdo existente. As propriedades que utilizam o mecanismo de contribuição são: valores de atributos e instâncias de associação. O efeito, após a interpretação do mecanismo *contributes,* é que o elemento base é substituído com uma visão variante aumentada do elemento que combina os valores dos atributos e as instâncias de associação. A forma como essa combinação é realizada depende do tipo do atributo ou da associação (veja Tabela 2). Por exemplo, atributos do tipo *String* de um elemento são concatenados (como por exemplo, contribuir com a descrição de uma atividade). Ainda, elementos adicionais são acrescidos em uma associação 0..* (como por exemplo, adicionar um elemento do tipo *Guidance* em uma Tarefa ou em uma Atividade). A Tabela 2 fornece a lista das regras para combinação dos atributos e associações.

Tabela 2 - Regras para o mecanismo *contributes*

Valores de Atributos	Valores do tipo *String* do elemento variável serão concatenados com os valores do tipo *String* do elemento base. Valores de qualquer outro tipo do elemento variável, tais como *Integer*, *Data*, URL são ignorados. Os atributos identificadores, como por exemplo o nome do elemento, estão isentos destas regras e não serão modificados.
Associações - 0..1 (*Outcoming*)	Uma instância da associação 0..1 do elemento base é mantida no elemento variável e a associação do elemento variável é ignorada. Se o elemento base não contém uma instância para a associação e o elemento variável contém, então o elemento resultante terá a associação do elemento variável.
Associações - 0..* (*Outcoming*)	Instâncias da associação 0..* do elemento variável são adicionadas as já existentes no elemento base. Se ambos os elementos (base e variável) se referem ao mesmo objeto, então, apenas uma instância da associação irá permanecer.
Associações - 0..1 (*Incoming*)	Uma instância da associação 0..1 do elemento base é mantida no elemento variável e a associação do elemento variável é ignorada. Se o elemento base não contém uma instância para a associação e o elemento variável contém, então o elemento resultante terá a associação do elemento variável.
Associações - 0..* (*Incoming*)	Instâncias da associação 0..* do elemento variável são adicionadas as já existentes no elemento base. Se ambos os elementos (base e variável) se referem ao mesmo objeto, então, apenas uma instância da associação irá permanecer.

- *replaces*: esse mecanismo permite que elementos do repositório ou o conteúdo de uma atividade do processo sejam substituídos. As propriedades que utilizam o mecanismo de substituição são: valores de atributos e instâncias de associação. Por exemplo, o mecanismo permite que seja alterado o papel responsável por um produto de trabalho ou ainda que seja modificada a descrição de uma tarefa. A Tabela 3 fornece a lista das regras para combinação dos atributos e associações.

Tabela 3 - Regras para o mecanismo *replaces*

Valores de Atributos	Valores do elemento base são substituídos com os valores do elemento variável incluindo os valores que não foram atribuídos. Esta regra também é aplicada aos atributos identificadores. Isto quer dizer que depois da substituição o objeto resultante tem o Id do elemento variável e não o do elemento base.
Associações - 0..1 (*Outcoming*)	Uma instância da associação 0..1 do elemento base é substituído com a instância da associação do elemento variável. Se o elemento variável não contém uma instância para a associação então o elemento resultante também não terá a associação.
Associações - 0..* (*Outcoming*)	Instâncias da associação 0..* do elemento base são substituídas pelas instâncias da associação do elemento variável. Se o elemento variável não contém uma instância para alguma associação 0..* então o elemento resultante também não terá esta associação.
Associações - 0..1 (*Incoming*)	Uma instância da associação 0..1 do elemento base é substituído com a instância da associação do elemento variável. Se o elemento variável não contém uma instância para a associação então o elemento resultante herdará o valor do elemento base.
Associações - 0..* (*Incoming*)	Instâncias da associação 0..* do elemento variável são adicionadas as já existentes no elemento base. Se ambos os elementos (base e variável) se referem ao mesmo objeto, então, apenas uma instância da associação irá permanecer.

- *extends*: este mecanismo permite reutilizar elementos do repositório ou o conteúdo de uma atividade do processo, proporcionando uma espécie de herança. As propriedades que utilizam o mecanismo de extensão são: valores de atributos e instâncias de associação. O resultado desse mecanismo é que o elemento que estende outro elemento tem as mesmas propriedades que o elemento estendido, mas pode sobrepor os valores destas propriedades com seus próprios valores, o que é uma variante do conteúdo já definido. Por exemplo, o mecanismo de extensão permite que uma versão especial de um Registro de Revisão genérico seja definida para um tipo específico de revisão. A Tabela 4 fornece a lista das regras para combinação dos atributos e associações.

Tabela 4 - Regras para o mecanismo *extends*

Valores de Atributos	Valores do elemento base são herdados e usados para popular os valores do elemento variável. Se atributos do elemento variável já estão populados, os valores herdados são ignorados. Os atributos

	identificadores são isentos desta regra e não serão modificados.
Associações - 0..1 (*Outcoming*)	Uma instância da associação 0..1 do elemento base é herdada para a instância da associação do elemento variável. Se o elemento variável define sua própria instância da associação então o valor herdado da associação é ignorado.
Associações - 0..* (*Outcoming*)	Instâncias da associação 0..* do elemento base são herdadas para a instância da associação do elemento variável. Se o elemento variável define suas próprias instâncias para a associação então os valores herdados da associação são ignorados.
Associações - 0..1 (*Incoming*)	Instâncias da associação 0..* do elemento base não são herdadas para a instância da associação do elemento variável.
Associações - 0..* (*Incoming*)	Uma instância da associação 0..1 do elemento base não é herdada para a instância da associação do elemento variável.

- *extends-replaces*: este mecanismo combina os efeitos do mecanismo de extensão e substituição. As propriedades que utilizam o mecanismo de extensão-substituição são: valores de atributos e instâncias de associação. A diferença desse mecanismo para o mecanismo de substituição é que o mecanismo de substituição, conforme seu nome sugere, substitui completamente todos os atributos e as instâncias de associação do elemento original com novos valores e as novas instâncias de associação, removendo, assim, todos os valores e as instâncias de associação do elemento original que não sofreram substituição. Já esse mecanismo permite substituir apenas partes do elemento original, alterando somente alguns valores de atributos e instâncias de associação. No mecanismo extensão-substituição os valores de atributos e as instâncias de associação que não foram substituídos permanecem com os valores do elemento original. A Tabela 5 fornece a lista das regras para combinação dos atributos e associações.

Tabela 5 - Regras para o mecanismo *extends-replaces*

Valores de Atributos	Valores do elemento base são herdados para o elemento variável somente nos casos onde não são definidos valores para o elemento variável. Essa regra também é aplicada aos atributos identificadores. Isto quer dizer que depois da substituição o objeto resultante tem o Id do elemento variável e não o do elemento base.
Associações - 0..1 (*Outcoming*)	Uma instância da associação 0..1 do elemento base é herdada para a instância da associação do elemento variável. Se o elemento variável define sua própria instância da associação então o valor herdado da associação é ignorado.
Associações - 0..* (*Outcoming*)	Instâncias da associação 0..* do elemento base são herdadas para a instância da associação do elemento variável. Se o elemento variável define suas próprias instâncias para a associação então os valores herdados da associação são ignorados.
Associações - 0..1 (*Incoming*)	Uma instância da associação 0..1 do elemento base é substituído com a instância da associação do elemento variável. Se o elemento

	variável não contém uma instância para a associação então o elemento resultante herdará o valor do elemento base.
Associações - 0..* (*Incoming*)	Instâncias da associação 0..* do elemento variável são adicionadas as já existentes no elemento base. Se ambos os elementos (base e variável) se referem ao mesmo objeto, então, apenas uma instância da associação irá permanecer.

Para entender o funcionamento do mecanismo *Variability* nos elementos do *Method Content* e em instâncias do elemento de processo *Activity,* a seguir alguns exemplos serão apresentados.

Os primeiros exemplos mostrados na Figura 22 exibem os tipos de variabilidade e*xtends* e c*ontributes,* sendo utilizados na metaclasse *TaskDefinition* do pacote *Method Content*. No lado esquerdo da Figura 22, a instância *B* de *TaskDefinition* aponta para a instância *A* de *TaskDefinition,* utilizando o valor c*ontributes* no seu atributo *variabilityType*. Já no lado direito da Figura 22, a mesma instância *B* de *TaskDefinition* aponta para a mesma instância *A* de *TaskDefinition,* contudo, neste exemplo, o valor do atributo *variabilityType* da instância B é igual a *extends.*

Baseado nas semânticas e regras explicadas acima para os tipos de variabilidade e*xtends* e c*ontributes*, o resultado da interpretação dos exemplos mostrados na Figura 22 é exibido na Figura 23.

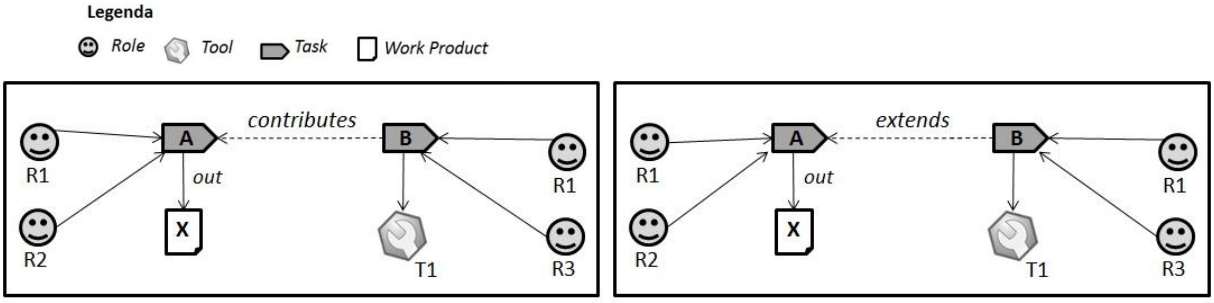

Figura 22 – Exemplos de aplicação do mecanismo *Variability* em elementos do *Method Content*

Observa-se na Figura 23 que a única diferença entre os resultados mostrados para a interpretação das variabilidades e*xtens* e c*ontributes* é a instância *R2* de *RoleDefinition*, que aparece apenas para o resultado do c*ontributes*. Isso ocorreu porque, de acordo com a semântica e regras deste tipo de variabilidade, todos os relacionamentos da instância (*incoming* e *outcoming*) que sofre o apontamento (nesse caso, a instância *A*) e são do tipo 0..* deverão ser somadas às relações do elemento que realiza o apontamento (nesse caso, a instância *B*). Especificamente, considerando o resultado do tipo de variabilidade e*xtends,* a instância *R2* de *RoleDefinition* não foi mantida no

resultado, pois, de acordo com a semântica e regras deste tipo de variabilidade, todos os relacionamentos (*incoming*) da instância que sofre o apontamento (nesse caso, a instância *A*) e são do tipo 0..* não deverão ser herdadas.

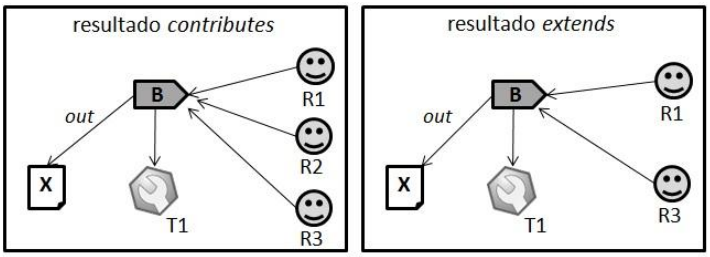

Figura 23 – Resultados da aplicação do mecanismo *Variability* em elementos do *Method Content*

O último exemplo para o mecanismo de variabilidade é mostrado no lado esquerdo da Figura 24 e é realizado sobre instâncias da metaclasse *Activity* do pacote *Process Structure*. Nesse exemplo, a instância *Activity 2* de *Activity* aponta para a instância *Activity 1* de *Activity* utilizando o valor *Extends* no seu atributo *variabilityType*.

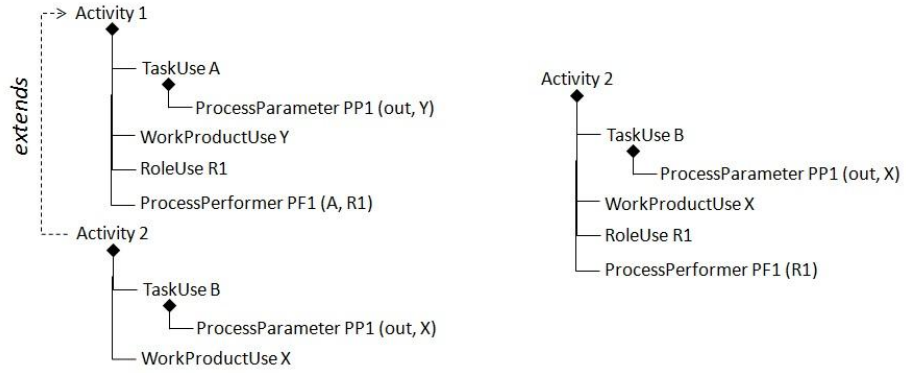

Figura 24 – Exemplos de aplicação do mecanismo *Variability* no elemento *Activity*

O resultado após a interpretação do exemplo acima é mostrado na Figura 24, no lado direito. Os elementos resultantes na atividade *Activity 2* mostram que somente as instâncias *R2* de *RoleDefinition* e *PF1* de *ProcessPerformer* foram herdadas para essa atividade. Isto ocorreu porque, como explicado acima, quando o tipo de variabilidade e*xtends* é utilizado, todos os relacionamentos (*outcoming*) da instância que sofre o apontamento (nesse caso, a instância *Activity 1*) e são do tipo 0..* não deverão ser herdadas se a instância que realiza o apontamento (nesse caso, a instância *Activity 2*) já define suas próprias instâncias do relacionamento. Baseado nesse contexto, uma vez que a instância *Activity 2* de *Activity* já possui relações com instâncias de *TaskUse* e *InternalUse,* estes elementos não foram herdados para a *Activity 1*.

Para finalizar o entendimento sobre a aplicação dos mecanismos *usedActivity*, *supressedBreakdownElement* e *Variability* é importante apenas considerar que, embora estes mecanismos sejam utilizados na atividade de adaptação de um processo de software, eles podem também ser utilizados na sua atividade de definição.

2.7 Considerações Finais

Este capítulo apresentou os conceitos referentes a processos de software objetivando demonstrar a importância de seu uso para as organizações de TI. Inicialmente, foi apresentado o ciclo de vida básico para um processo de software e descritas em detalhe as suas atividades de definição e adaptação. Em seguida, foi realizada uma descrição sobre o metamodelo SPEM 2.0 que é utilizado por esta pesquisa para a definição e adaptação dos processos de software.

Após o entendimento sobre o ciclo de vida básico dos processos de software e do funcionamento do metamodelo SPEM 2.0, o capítulo seguinte desta tese discorre sobre o aspecto da consistência nos processos de software.

3 CONSISTÊNCIA X PROCESSOS DE SOFTWARE

Neste capítulo, será descrito o aspecto da consistência nos processos de software durante suas atividades de definição e adaptação. Apresentar-se-ão também apresentados os resultados de uma revisão sistemática, conduzida nesta pesquisa sobre como é realizado o tratamento da consistência em estudos que apresentam soluções para definição e adaptação dos processos de software.

3.1 Introdução

O desenvolvimento de sistemas tem sido prejudicado por problemas de inconsistência desde seu início [Luc09]. Esse é um dos motivos pelo qual a verificação de inconsistências envolvendo diversos tipos de informação tem sido objeto de estudos em muitas áreas da engenharia de software.

Uma das primeiras áreas que identificaram problemas relacionados com o gerenciamento e a manutenção de inconsistências foi a de banco de dados [Cod70]. Nesta área, as primeiras inconsistências detectadas foram, na sua maioria, causadas por redundância de informações, resultantes de erros em projetos de banco de dados.

Outra área que explora, há bastante tempo, o aspecto da consistência é a conhecida na literatura como Verificação de Modelos (do inglês *Model Checking*). Nesta área, o interesse é a checagem de consistência entre modelos UML e é consenso, entre seus autores, que, devido à existência dos múltiplos modelos dessa linguagem, várias inconsistências sejam introduzidas em uma especificação de software [Eng01], [Wag03], [Luc05], [Egy07a], [Egy07b], [Lal08]. Sendo assim, tais inconsistências podem se constituir em uma fonte de numerosos erros em um produto de software, dificultando sua manutenção [Mus05].

Uma prova da importância do tema consistência entre modelos UML é a quantidade de trabalhos encontrados na literatura sobre o assunto. Em uma recente revisão sistemática conduzida por [Luc09], foram analisados 44 artigos sobre este assunto, fato que levou os autores desse estudo a concluírem que esta é uma área de pesquisa altamente ativa e promissora. Outro importante fator que prova a relevância do tema consistência entre modelos UML são as ferramentas comerciais e não comerciais que implementam funcionalidades de verificação de modelos. Esse é caso das ferramentas comerciais *IBM Rational Software Architecture* [Nor09] e *IBM Software*

Modeler [Ibm09] e do *plugin* do eclipse *Eclipse Modeling Framework – EMF* [Emf07], que é uma ferramenta não comercial.

Uma terceira área da engenharia de software que trata de aspectos de consistência é a modelagem de processos de negócio e *workflows.* Neste campo de pesquisa, os autores apresentam vários estudos, propondo soluções para garantir a consistência do *workflow* que modela um processo de negócio [Mar00], [Li03] e [Zho05]. Isso porque, de acordo com [Zho05], um modelo de *workflow* que contém inconsistências pode levar à falha na execução de um processo de negócio.

Especificamente na área dos processos de software, a consistência também possui devida importância tanto nas atividades de definição e adaptação quanto na sua atividade de execução. Logo, considerando que esta pesquisa tem foco no tratamento da consistência dos processos de software nas atividades de definição e adaptação, as próximas seções deste trabalho apresentarão, respectivamente, uma descrição sobre este assunto e o resultado de uma revisão sistemática conduzida nessa área.

3.2 Consistência em Processos de Software

Independente da estratégia escolhida por uma organização de TI para definição e adaptação de seus processos de software, é importante compreender a complexidade associada a essas atividades. Isso porque, normalmente, um processo de software possui dezenas, às vezes centenas de elementos interconectados por relacionamentos e qualquer inconsistência na definição desse processo poderá ter impacto negativo sobre a sua execução em um projeto de software.

Assim, a consistência de um processo de software é definida como a exigência de que suas atividades, tarefas, produtos de trabalho, ferramentas, papéis e relações não contenham qualquer contradição e sejam, portanto, coerentes entre si [Har97]. De acordo com Atkinson *et al.* [Atk07], erros são frequentemente introduzidos na definição de um processo de software, por seus modeladores, e, geralmente, estão relacionados com a sintaxe ou erros tipográficos, o que leva a várias inconsistências no processo resultante.

Um modelador pode, por exemplo, cometer um erro simples ao conectar um produto de trabalho que ainda não foi produzido no processo de software como entrada para uma determinada tarefa (denominado em [Atk07] como *ação*). Ao fazer isso, o modelador pode quebrar uma dependência do processo, pois, se a tarefa em questão depende da entrada faltante, ela não pode ser executada, o que causa uma falha para a

execução do processo de software. Outros exemplos de inconsistências incluídas na atividade de definição dos processos de software são encontrados nos estudos de [Baj07], [Dai07], [Hus08] e [Bao08].

Para Bajec *et al.* [Baj07], os problemas de consistência nos processos de software estão basicamente relacionados com a incompletude das suas informações. Nesse sentido, os autores referenciam a importância do uso de um metamodelo para a definição de processos que estabeleça regras mínimas de consistência, através dos seus atributos e suas associações entre metaclasses. Esses tipos de regras devem, por exemplo, definir a quantidade mínima e máxima de papéis, produtos de trabalho e ferramentas que devem estar associados tarefas em um processo de software. Ainda, de acordo com Bajec *et al.* [Baj07], outras regras, as quais não podem ser expressas através de um metamodelo, devem ser definidas, utilizando outro tipo de linguagem como, por exemplo, a natural.

As inconsistências relacionadas com a incompletude de informações em um processo de software também são referenciadas por [Hus08]. Analogamente ao estudo de Bajec *et al.* [Baj06], o trabalho apresentado por Hsueh *et al.* também referencia o uso de um metamodelo de processo que permita a definição de regras de validação para cada um de seus elementos.

Outros importantes tipos de inconsistências citados por alguns estudos na literatura [Dai07], [Bao08], os quais também podem ocorrer durante a definição dos processos de software, são aquelas relacionadas com o sequenciamento definido para atividades ou tarefas. De acordo com Bao [Bao08], um conjunto de regras deve ser estabelecido para os elementos de um processo de software, de forma a garantir sua correta execução em um projeto de software. Essas regras, basicamente, devem definir requisitos afim de que o ponto inicial e final das atividades ou tarefas seja estabelecido, assim como devem estabelecer os requisitos para que as atividades ou tarefas sejam sequenciadas de forma a não apresentarem nenhuma situação que represente um *deadlock*[3] durante a execução do processo de software.

Analisando o exposto acima, é possível observar que é comum entre os autores da área de pesquisa sobre definição de processos, o uso de regras para checagem da consistência de um processo de software. Alguns destes autores enfatizam também uma etapa específica, durante a definição dos processos de software, para checagem da

[3] Um *deadlock* se configura como uma situação onde duas ou mais atividades estão esperando uma pela outra para iniciar ou terminar formando assim uma cadeia circular.

consistência desses processos [Atk07], [Baj07] e [Hsu08]. Atkinson *et al.* em [Atk07], por exemplo, destacam a importância da checagem de um processo de software antes de sua execução devido à habilidade que esse mecanismo fornece para detectar e eliminar inconsistências em um processo antes que elas se tornem erros e comprometam o sucesso de um projeto de software. Hsueh *et al.* em [Hsu08] compartilham, em seu artigo, da mesma ideia de [Atk07] e ainda relatam que a verificação de processos é reconhecida como um elemento essencial para melhoria de um processo de software.

Faz-se necessário citar que inconsistências podem ser introduzidas também durante a adaptação dos processos, devido a falta de mecanismos apropriados para conduzir esse tipo de atividade. Para Yoon *et al.* [Yoo01], um mecanismo apropriado para adaptação de processos é fornecido por um conjunto de regras que guiem sistematicamente as operações de adaptação e preservem as relações de dependência entre os elementos de um processo de software. De acordo com *Park et al.* [Par11], todo mecanismo de adaptação deve fornecer uma análise de impacto sobre os elementos e relacionamentos afetados durante qualquer mudança em um processo de software. Para esses autores, essa é a única maneira de garantir que o responsável por uma mudança possa manter a consistência do processo original.

Outros autores, os quais apresentam soluções especificamente para a atividade de adaptação de processos, também relacionam o uso de regras para essas atividades [Wel95] e [Rui09] e suportam a análise de impacto para mudanças em processos de software [Wel95], [Yoo01] e [Rui09]. Percebe-se que, assim como na atividade de definição de processos, o uso de regras também é o principal mecanismo utilizado para garantir a consistência de um processo de software durante suas atividades de adaptação.

Desse modo, visando descrever as regras e o funcionamento dos trabalhos citados acima e de outros estudos relacionados, assim como objetivando a identificação das lacunas de pesquisa na área de definição e adaptação de processos de software consistentes, a seguir, apresenta-se o resultado da revisão sistemática conduzida sobre esse assunto nesta pesquisa.

3.3 Revisão Sistemática sobre Consistência em Processos de Software

Após vários estudos preliminares na literatura para identificação de pesquisas relacionadas foi realizada, nesta pesquisa, uma revisão sistemática sobre o tratamento do aspecto da consistência nos processos de software. De acordo com kitchenham [Kit04], uma revisão sistemática permite identificar, avaliar e interpretar toda a literatura relevante para uma determinada questão de pesquisa, tópico ou fenômeno de interesse.

As três principais fases de uma revisão sistemática são [Kit04]: planejamento, condução e relatório da revisão. A fase de planejamento envolve a definição do protocolo da revisão sistemática, que apresenta o propósito da revisão e os procedimentos que serão adotados. Durante a condução da revisão, é feita a busca por trabalhos relevantes para o objeto de estudo, a seleção das publicações e a extração dos dados de cada uma das publicações selecionadas. Na fase de relatório da revisão, as conclusões obtidas devem ser descritas em relatórios ou artigos. Exemplos de revisões sistemáticas na literatura podem ser encontrados em [Ped07], [Cal08] e [LucC09].

Inicialmente, para realizar esta revisão, o protocolo (Apêndice A) definido partiu da comtemplação da seguinte questão de pesquisa:

“Quais são as abordagens que apresentam soluções relacionadas com o aspecto de consistência em um processo de desenvolvimento de software nas suas atividades de definição e/ou adaptação?”

A partir da definição desta questão de pesquisa, das “*strings”* e fontes de busca utilizadas que estão descritas no protocolo, a revisão sistemática foi executada. A seleção inicial dos artigos retornou um total de 212 artigos. A partir deste resultado os seguintes procedimentos foram realizados na revisão:

- foram excluídos os artigos que não estavam no formato completo ou que não puderam ser baixados a partir dos motores de busca.
- foram lidos o título e o resumo (*abstract*) de cada artigo para verificar se o artigo era aprovado para revisão.
- os artigos remanescentes (aprovados) foram selecionados para leitura integral.

No primeiro procedimento (exclusão de artigos não completos), foram excluídos 33 artigos. Dos 179 artigos selecionados para a segunda etapa, 135 foram excluídos após a leitura de título e resumo. Dessa forma, 44 artigos foram lidos de forma integral, sendo

que apenas 10 atenderam aos critérios para extração de informações (o que correspondeu a 4,7% do conjunto inicial de artigos).

Além dos 10 artigos selecionados na execução da revisão sistemática outros 3 artigos foram incluídos para extração de informações. Segundo [Oli07], a inclusão de artigos é possível em revisões sistemáticas e isso é comum em várias áreas do conhecimento, tal como, por exemplo, a Medicina. A Tabela 6 lista os títulos e as referências dos artigos usados nessa revisão sistemática para extração de informações. Nas últimas linhas, em destaque, estão os artigos que não foram obtidos com os critérios estabelecidos por essa revisão.

Tabela 6 – Referências da revisão sistemática

Id	Fonte	Referência	Título
1	ACM	[Oje09]	agentTool Process Editor: Supporting the Design of Tailored Agent-based Processes
2	ACM	[Boe06]	Applying the Value/Petri Process to ERP Software Development in China
3	IEEE	[Was06]	Deriving Project-Specific Processes from Process Line Architecture with Commonality and Variability
4	IEEE	[Dai07]	Tailoring Software Evolution Process
5	ScienceDirect	[Hsu08]	Applying UML and Software Simulation for Process Definition, Verification and Validation
6	ScienceDirect	[Atk07]	Tool Support for Iterative Software Process Modeling
7	ScienceDirect	[Par11]	An Approach to Analyzing the Software Process Change Impact using Process Slicing and Simulation
8	ScienceDirect	[Baj07]	Practice-Driven Approach for Creating Project-Specific Software Development Methods
9	ScienceDirect	[Fer10]	A Technique for Defining Agent-Oriented Engineering Processes with Tool Support
10	SCOPUS	[Rod10]	Defining Software Process Model Constraints with Rules Using Owl and Swrl
11	IEEE	[Yoo01]	Tailoring and Verifying Software Process
12	IEEE	[Bao08]	A Study of Rationality Test Rules for Software Process Model
13	IEEE	[Rui09]	A Software Process Tailoring Approach Using a Unified Lifecycle Template

Para avaliar o conjunto de artigos relacionados na tabela acima o seguinte conjunto de critérios para identificação de informações foi utilizado. Mais especificamente, as informações buscadas foram (ver Tabelas 7 e 8):

- Linguagem para Modelagem de Processos: indica se a solução usa ou propõe alguma linguagem para modelagem de processos (por exemplo, metamodelo baseado na UML, PML, etc.)
- Regras e/ou Restrições para Consistência na Definição de Processos: indica se a solução define algum tipo de mecanismo baseado em regras ou restrições para consistência dos processos durante suas atividades de definição;

- Define Operações para Adaptação: indica se a solução propõe operações para adaptação de processos (por exemplo, inclusão de atividades, exclusão de produtos de trabalho, etc.);
- Regras e/ou Restrições para Consistência na Adaptação de Processos: indica se a solução define algum tipo de mecanismo baseado em regras ou restrições para consistência dos processos durante suas atividades de adaptação;
- Forma de Avaliação: indica como a solução foi avaliada;
- Ferramenta: indica se a solução usa alguma ferramenta ou protótipo;
- Verificação de Processos: indica se a solução propõe algum mecanismo de verificação ou checagem de consistência nos processos de software nas atividades de definição e/ou adaptação;
- Formalismo para Regras: indica se a solução utiliza algum tipo de formalismo nas regras e/ou restrições de consistência nas atividades de definição e/ou adaptação.

Os itens marcados com um asterisco (*) nas Tabelas 7 e 8 indicam que a característica em questão é tratada de forma superficial ou não muito clara no artigo, quer dizer, sem apresentar diretamente uma solução para a característica em questão (contudo, ainda importante para o presente estudo).

Tabela 7 – Primeira parte dos resultados da revisão sistemática

Referência	Linguagem para Modelagem de Processos	Regras e/ou Restrições para Consistência na Definição de Processos	Define Operações para Adaptação	Regras e/ou Restrições para Consistência na Adaptação de Processos
[Baj07]	Sim – Metamodelo em UML	Sim	Sim*	Sim
[Atk07]	Sim – PML	Sim	Não	Não
[Hsu08]	Sim – Metamodelo em UML	Sim	Não	Não
[Was06]	Sim* - Metamodelo SPEM	Sim	Sim	Sim
[Fer10]	Sim – Metamodelo SPEM	Sim*	Não	Não
[Oje09]	Sim – Metamodelo OPF	Não	Sim*	Sim*
[Boe06]	Sim – Rede de Petri	Não	Sim*	Não
[Par11]	Sim – Metamodelo SPEM	Não	Sim*	Sim*
[Dai07]	Sim* - Rede de Petri	Não	Sim	Sim
[Rod10]	Sim – Metamodelo SPEM	Sim*	Não	Não
[Yoo01]	Sim – Proposto pelos Autores	Sim*	Sim	Sim
[Bao08]	Sim – Diagrama de Atividade, Rede de Petri e Diagrama de Fluxo de Dados	Sim	Não	Não
[Rui09]	Sim – Metamodelo baseado no RUP	Não	Sim	Sim

Tabela 8 – Segunda parte dos resultados da revisão sistemática

Referência	Forma de Avaliação	Ferramenta	Verificação de Processos	Formalismo para Regras
[Baj07]	Estudo de Caso	Sim	Sim	Não
[Atk07]	Exemplo de Uso	Sim	Sim	Sim – Pseudo Código
[Hsu08]	Experimento	Sim	Sim	Sim - OCL
[Was06]	Exemplo de Uso	Não	Não	Não
[Fer10]	Exemplo de Uso	Sim	Não	Não
[Oje09]	Exemplo de Uso	Sim	Sim	Não
[Boe06]	Estudo de Caso	Sim	Sim	Sim – Rede de Petri
[Par11]	Estudo de Caso	Sim	Não	Não
[Dai07]	-	Não	Não	Sim – Pseudo Código
[Rod10]	Exemplo de Uso	Não	Sim	Sim – OWL e SWRL
[Yoo01]	Exemplo de Uso	Sim	Sim	Não
[Bao08]	-	Não	Sim*	Não
[Rui09]	Exemplo de Uso	Sim	Não	Não

Dessa maneira, na sequência, uma análise qualitativa dos artigos selecionados a partir da revisão sistemática será descrita. Para essa descrição os artigos serão organizados em dois grupos: (1) artigos com foco no tratamento da consistência na atividade de definição dos processos de software; e (2) artigos com foco no tratamento da consistência na atividade de adaptação dos processos de software. Os estudos de [Yoo01], [Was06] e [Baj07] que, de alguma forma, relacionam o aspecto da consistência em ambas as atividades de definição e adaptação, aparecerão nos dois grupos conforme o aspecto tratado.

3.3.1 Consistência na Atividade de Definição dos Processos de Software

Considerando o primeiro critério de extração de informações que é sobre o uso ou definição de linguagem para modelagem de processos, constata-se, a partir das Tabelas 7 e 8, que todos os artigos descrevem algum tipo de solução relacionada com esse aspecto. Dos 13 artigos analisados, apenas [Was06] e [Dai07], os marcados com * não apresentam com clareza como é realizada a modelagem de processos.

No caso específico do estudo apresentado em [Was06], os autores trabalham com linhas de processo e diagrama de *features,* não dando ênfase a linguagem de modelagem dos processos. Contudo, em alguns pontos, os referidos autores relacionam o uso do metamodelo SPEM 2.0. Em [Dai07], os autores apenas referenciam o uso de uma linguagem denominada EPMN que seria baseada em Redes de Petri. Contudo, nenhum detalhamento maior sobre a linguagem ou forma de modelagem é apresentado.

Ainda com relação a linguagem de modelagem de processos, observa-se que 9 dos 13 artigos utilizam a linguagem UML para descrever os processos de software, o que corrobora o fato dessa linguagem ser um padrão na área de modelagem de processos. Estes são os casos dos trabalhos que propõem metamodelos próprios ([Baj07], [Hsu08]); utilizam o metamodelo SPEM ([Was06], [Fer10], [Rod10], [Par11]); utilizam o metamodelo RUP ([Rui09]); utilizam diagrama de atividades da UML ([Was06] e [Bao08]); e utilizam o metamodelo OPF ([Oje09]). As 3 soluções restantes, que não foram citadas ainda, trabalham com Rede de Petri ([Boe06]), linguagem de modelagem que os autores chamam de PML ([Atk07]); e linguagem própria ([Yoo01]), a qual é baseada em módulos de processo formados por tarefas e produtos de trabalho (denominado em [Yoo01], respectivamente, como *atividade* e *artefato*). Em [Bao08], além do diagrama de atividades da UML, são também utilizados para modelagem dos processos Rede de Petri e diagrama de fluxo de dados.

Prosseguindo com a avaliação dos critérios estabelecidos nesta revisão, verificou-se que 8 dos 13 artigos abordam, em suas soluções, algum tipo de regra ou restrição para a consistência de um processo de software durante suas atividades de definição. Inicialmente, em relação ao estudo apresentado em [Yoo01], marcado com * na Tabela 7, verificou-se que seus autores não exploram especificamente regras para consistência na definição de processos. Contudo, em meio à descrição desse trabalho, foi possível constatar uma regra de consistência para a atividade de definição. Essa regra estabele que produtos de trabalho só podem ser produzidos por uma tarefa em um processo de software.

Em [Was06] os autores explicam como montar uma arquitetura de linhas de processo, utilizando diagramas de *features*. Nessa arquitetura, são estabelecidos pontos de variabilidade (em inglês *Variability*), que são pontos de variação do processo, e as variantes de processo. Os pontos de variação são as tarefas (denominado em [Was06] como *atividade*) que podem ser adaptadas de acordo com as características do processo e as variantes do processo são as tarefas candidatas que são aplicadas nos pontos de variação.

Nesse estudo em questão, os autores mostram a necessidade de estabelecer todos os relacionamentos entre todos os elementos do processo, incluindo os relacionamentos para cada variante de processo. Eles dão ênfase à definição consistente dos relacionamentos de precedência entre as tarefas para que, no momento que uma variante de processo seja selecionada, o fluxo do processo possa ser estabelecido

também de forma consistente. Além disso, é possível também definir pontos opcionais no processo e, dessa forma, os autores exploram como devem ser estabelecidas as relações de precedência envolvendo os elementos opcionais do processo, uma vez que eles poderão ser excluídos em um novo processo. O objetivo é, mais uma vez, não violar o fluxo de tarefas do processo gerado.

Os trabalhos de [Baj07], [Atk07] e [Bao08] são os mais completos em estabelecerem regras para consistência de processos na atividade da definição. Em [Baj07] os autores definem um conjunto de regras e classificam as mesmas em regras de restrição (*constraint*) e regras fato.

Relacionadas com aspecto de consistência estão as regras de restrição, as quais são subdivididas em: regras de fluxo, regras de estrutura, regras de consistência e regras de completude. Inicialmente, as regras de fluxo definem condições que precisam ser atendidas para que uma transição possa ser executada. Essas regras equivalem as condições de guarda do diagrama de atividades da UML. Uma condição de guarda controla qual transição, de um conjunto de transições alternativas, ocorre após a conclusão da atividade. As regras de estrutura são análogas as regras de fluxo, pois também estabelecem condições para que um fato ocorra dentro do processo. Segundo Bajec *et al.* [Baj07], a diferença das regras de estrutura para as regras de fluxo são que as regras de estrutura restringem qualquer elemento do processo e não somente tarefas (denominada em [Baj07] como *atividade*) (como é o caso das regras de fluxo). Uma regra de estrutura pode, por exemplo, definir, que quando uma determinada tarefa for executada em um período superior a um mês, ela deverá ser realizada utilizando determinada ferramenta.

Ambas as regras de fluxo e regras de estrutura são regras que consideram as características de projeto e estão associadas à execução do processo em um projeto de software. Por fim, as regras de completude e consistência são as regras que mais se aproximam do tema desta pesquisa. As regras de completude são aquelas relacionadas com as multiplicidades UML e estabelecem que todas as associações entre as metaclasses de um metamodelo devem ser devidamente respeitadas na modelagem de um processo. Já as regras de consistência são aquelas que não podem ser expressas através das multiplicitadades UML e que definem as dependências entre os elementos de um processo de software. Como um exemplo para as regras de consistência, o autor cita a dependência que pode ocorrer entre dois produtos de trabalho (quando um produto de

trabalho depende de um ou mais produtos de trabalho para ser produzido e/ou modificado) ou ainda a de um produto de trabalho com sua tarefa de produção.

Atkinson *et al.* em [Atk07] estabelecem duas principais categorias de erros que geram inconsistências em um processo de software: erros locais em tarefas e erros que envolvem várias tarefas. A primeira categoria estabelece quatro regras: (1) *Empty* – esta regra estabelece que tarefas não podem ser vazias, ou seja, não possuir parâmetros nem de entrada e nem de saída (em termos de produtos de trabalho); (2) *Black Holes*: esta regra estabelece que tarefas não podem ter somente parâmetros de entrada (em termos de produtos de trabalho); (3) *Miracle*: esta regra estabelece que tarefas não podem ter somente parâmetros de saída (em termos de produtos de trabalho); e (4) *Transformation*: esta regra estabelece que todos os parâmetros de saída (em termos de produtos de trabalho) de uma tarefa devem também estar conectados como seus parâmetros de entrada . No caso desta última, os autores definem que poderá ser usado um qualificador para indicar que o produto de trabalho está sendo produzido na tarefa. Nesse caso, a regra não indicaria o erro.

O segundo conjunto de regras proposto por [Atk07] é relacionado com as dependências que pode haver entre os elementos de um processo de software. Basicamente, as regras propostas aqui verificam se um produto de trabalho que é consumido em uma tarefa foi produzido anteriormente e se existe caminho entre sua tarefa de produção e as tarefas que consomem este artefato. Outra regra estabelecida é verificar se os produtos de trabalho que são produzidos em um processo são consumidos em algum ponto deste processo. Segundo Atkinson *et al.* [Atk07], isso só não gera uma inconsistência em se tratando da ação final.

No estudo descrito em [Bao08], são apresentadas somente regras relacionadas com consistência do sequenciamento entre atividades. Os autores definem regras de sequenciamento para diagramas de atividade, Redes de Petri e diagrama de fluxo de dados. Regras são definidas para essas três estruturas, pois, segundo os autores qualquer uma delas pode ser utilizada para modelagem de um processo. Com o intuito de entender melhor as regras definidas neste trabalho, abaixo, seguem as regras estabelecidas para os diagramas de atividade:

- Diagramas de atividade devem possuir somente um nodo inicial;
- Diagramas de atividade devem possuir um ou mais nodos finais;
- Todas as atividades devem ter caminho originado a partir do nodo inicial;

- Todas as atividades devem ter caminho de chegada no nodo final;
- O nodo inicial não pode ser destino (sucessor) para nenhuma outra atividade;
- O nodo final não pode ser origem (predecessor) para nenhuma outra atividade;
- Cada *disjoin* deve ter uma ou mais saídas e somente uma entrada;
- Cada *join* deve ter uma ou mais entradas e somente uma saída;
- Não devem existir situações de sequenciamento no diagrama que represente *deadlocks* durante a execução de um processo de software;

Hsueh *et al.* [Hsu08] definem um mecanismo de definição, verificação e simulação de processos e estabelecem várias regras para a etapa de verificação de consistência dos processos de software. Contudo, os referidos autores estão bastante interessados em sua pesquisa em verificar o quanto um processo atende às áreas de processo (em inglês *process área* – PA) do CMMI. Assim, a maioria das regras, as quais são descritas em *Object Constraint Language* – OCL, é relacionada com esse aspecto.

Especificamente relacionado ao tema desta pesquisa, que não trata de modelos de maturidade, os autores estabelecem apenas as regras associadas a respeitar as multiplicidades UML entre as metaclasses de um metamodelo de processo. Embora esse trabalho não estabeleça muitas regras de consistência ele é considerado relacionado, pois seus autores dão um enfoque muito grande a atividade de checagem de processos baseado em regras.

Em [Fer10], os autores utilizam o ambiente do *plugin Eclipse Modeling Framework* – EMF para definição de processos e não definem explicitamente regras de consistência. Contudo, conforme já citado antes, a ferramenta EMF possui uma funcionalidade de checagem de modelos UML.

Dessa forma, é considerado que o estudo proposto por [Fer10] também apresenta algumas regras de consistência. Com relação à checagem feita pelo EMF, isso é feito basicamente sobre as multiplicidades e atributos definidos em um modelo de classes UML. Por fim, em [Rod10] os autores propõem representar os processos modelados a partir do SPEM 2.0, usando ontologias (OWL e SWRL). Segundo os referidos autores, o objetivo é incluir capacidades de checagem de consistência no processo. Contudo, o artigo em questão se dedica quase que exclusivamente a mostrar a tradução do SPEM 2.0 para a linguagem OWL e a definir regras muito simples, que estão relacionadas ao tempo de execução dos processos. Tais autores definem, por exemplo, que a mesma pessoa não poderia estar alocada 100% a duas tarefas que ocorrem ao mesmo tempo. Relacionado aos trabalhos de [Fer10] e [Rod10], considera-se também que, embora eles

não estabeleçam muitas regras de consistência, são relacionados a esta pesquisa visto que dão enfoque à atividade de checagem de processos.

3.3.1.1 Conclusões e Lacunas Identificadas para a Atividade de Definição de Processos

Considerando a atividade de definição de processos verificou-se que todos os estudos analisados propõem ou se utilizam de uma linguagem para modelagem de processos.

No que concerne às regras e/ou restrições de consistência para a atividade de definição de processos, identificou-se, pelo nível de aprofundamento do assunto nos artigos selecionados para esta revisão sistemática, que esta é uma área de pesquisa que merece maior atenção. Concluiu-se isto, inicialmente, porque três dos nove artigos encontrados sobre esse assunto não definem de forma clara, suas regras para consistência. Em seus estudos [Yoo01], [Fer10] e [Rod10] citam, por exemplo, a importância da consistência para os processos de software na atividade de definição, contudo não deixam claro como isto é realizado. Especificamente, no caso de [Yoo01] a justificativa se dá pelo fato desse trabalho ter maior foco na atividade de adaptação dos processos, o que não é o caso dos estudos de [Fer10] e [Rod10], os quais possuem foco em consistência na atividade de definição dos processos.

Analisando as regras apresentadas nos seis artigos restantes, constatou-se que muitos autores estabelecem poucas regras em seus estudos e deixam vários aspectos de consistência sem tratamento. No caso específico dos trabalhos apresentados em [Baj07] e [Hsu08], os autores focam apenas nas regras que podem ser expressas através das informações de multiplicidade UML entre as metaclasses de seus metamodelos. Em [Hsu08], uma regra adicional considerando a dependência entre produtos de trabalho também é mostrada e descrita em linguagem natural. Contudo, em ambos os trabalhos, não é citado referências sobre regras de sequenciamento das atividades ou tarefas de um processo ou regras mais complexas envolvendo produtos de trabalho, papéis e até mesmo as tarefas.

Dessa forma, seria possível, por exemplo, nos processos gerados com base na solução proposta pelos estudos apontados acima incluir como parâmetro de entrada para uma tarefa um produto de trabalho que pelo sequenciamento estabelecido ainda não foi produzido no processo, ou ainda, um produto de trabalho que foi produzido anteriormente,

mas que, pelo sequenciamento estabelecido, não possui nenhuma ligação (não possui caminho) com a tarefa que o consome. De acordo com Atkinson *et al.* [Atk07], ambas as situações acima representam inconsistências para um processo de software.

Outros trabalhos que também tem foco em aspectos mais específicos para consistência de um processo de software são os estudos de [Was06] e [Bao08]. Os autores desses estudos apresentam regras específicas sobre sequenciamento entre atividades e tarefas em um processo de software, deixando em aberto todos os outros aspectos e elementos envolvidos em tal processo.

O trabalho apresentado por *Atiknson et al.* em [Atk07] é considerado o que tem a solução mais completa em relação a definição de regras de consistência para a atividade de definição dos processos de software. Ainda assim, cabe enfatizar que esses autores também não abordam vários aspectos de consistência em um processo e não consideram o elemento de processo *papel* em suas regras. Seria possível, por exemplo, em um processo definido a partir dessa solução a definição de um sequenciamento entre atividades ou tarefas totalmente inconsistente pela inclusão de situações que representem *deadlocks* durante a execução de um processo de software. Seria possível ainda definir tarefas que não fossem executadas por nenhum papel.

A partir das conclusões acima, identifica-se como principal lacuna para a área de pesquisa específica sobre as regras de consistência na atividade de definição a falta de estudos mais completos sobre o assunto. Entende-se como mais completos estudos que abordem todos os aspectos de consistência para os principais elementos de um processo de software, que de acordo com [Omg02] e [Omg07a] são as *Atividades,* as *Tarefas*, os *Papéis* e os *Produtos de Trabalho*.

3.3.2 Consistência na Atividade de Adaptação dos Processos de Software

Quanto à atividade de adaptação de processos, obteve-se como resultado que 8 dos 13 artigos analisados abordam, em suas soluções, algum aspecto relacionado com adaptação de processos. Nota-se que a quantidade de artigos que abordam este aspecto é grande, fato que indica a importância da atividade de adaptação no ciclo de vida dos processos de software.

Já com relação ao primeiro critério analisado, observa-se nas Tabelas 7 e 8 que todos os estudos definem operações de adaptação de processos. Inicialmente, com o trabalho apresentado em [Yoo01] verifica-se que *Yoon et al.* consideram operações de

adaptação somente sobre tarefas e produtos de trabalho do processo. Isto é feito no estudo citado, porque esses são os únicos elementos considerados pelos autores para a definição dos processos. Porém, referente às operações de adaptação, os mesmos autores relacionam operações de exclusão, adição, união e divisão de tarefas e produtos de trabalho. Tais pesquisadores apenas restringem a união de produtos de trabalho, pois, segundo eles, um produto de trabalho não pode ter duas tarefas de produção e, no caso de dois produtos de trabalho serem acoplados sem a união de suas tarefas produtoras, eles passariam a ser produzidos em ambas as tarefas, o que violaria uma de suas regras de consistência para definição de processos.

Em [Was06], por sua vez, a definição de um processo é realizada com base em uma linha de processos. Dessa forma, os autores definem os pontos de variação e variantes em um diagrama de *features*. Isso caracteriza uma operação de substituição, pois as variantes podem variar de um processo para outro. Além disso, como nos diagramas de *features* pontos podem ser definidos como opcionais considera-se que a operação de exclusão é permitida.

Contudo, como no estudo acima não são explicitados quais os elementos que fazem parte de um processo, então, não é possível identificar quais são os elementos que realmente podem sofrer operações de adaptação. Através dos exemplos do artigo em discussão foi possível identificar apenas as operações de exclusão em tarefas e produtos de trabalho. Não existem referências, por exemplo, aos papéis envolvidos no processo.

O conjunto de operações proposto em [Rui09] é bastante completo e permite a exclusão, inclusão, substituição, modificação e união de tarefas, produtos de trabalho e papéis do processo RUP. Em [Dai07], o conjunto de operações também é mais completo, uma vez que os autores desse estudo permitem a exclusão, adição, união e divisão de tarefas (denominado em [Dai07] como *atividade*).

Prosseguindo com a avaliação sobre o critério relacionado com as operações de adaptação, encontram-se os estudos que não definem suas operações explicitamente e são marcados com * na Tabela 7. Tais estudos são [Boe06], [Baj07], [Oje09] e [Par11]. Começando pelo trabalho de [Boe06], identifica-se que seus autores não descrevem atividades de adaptação e operações de forma evidente. Esses autores apenas citam. em seu trabalho. que tarefas (denominada em [Boe06] como *atividade*) podem ser excluídas de um processo e, dessa forma, considera-se que a operação de exclusão de tareafas é permitida nesse estudo. Em [Baj07] os autores definem que um processo base (padrão), o

qual é formado por fragmentos de método, tem muitos destes fragmentos opcionais para um projeto específico. Dessa maneira, analogamente ao trabalho de [Boe06], considera-se que a exclusão de elementos é permitida em [Baj07]. Contudo, como os autores do referido trabalho não detalham quais elementos de um processo formam os fragmentos de método e nem definem sua granularidade, não é possível saber exatamente quais elementos podem sofrer operações de adaptação.

Analogamente ao funcionamento da operação de exclusão proposta em [Baj07], apresenta-se a solução definida em [Oje09]. Neste estudo, Ojeda *et al.* também definem a operação de exclusão por considerarem fragmentos de método opcionais. A única diferença em relação a [Baj07] é que os autores do estudo apresentado em [Oje09] não relacionam o uso de um processo base (padrão) em sua solução. Eles definem que um repositório de fragmentos de método desconexos deve estar disponível e que todo processo específico de um projeto deve ser montado pela seleção de tais fragmentos. Além disso, esses autores também consideram que novos fragmentos de método podem ser incluídos pra um projeto em específico.

Por fim, analisando o estudo de [Par11], verifica-se que seus autores não definem claramente operações de adaptação em seu trabalho, uma vez que este não tem como foco a adaptação de processos. Sendo assim, os autores, apenas citam, durante o texto que, elementos tais como papéis, produtos de trabalho e tarefas (denominados em [Par11], respectivamente, como *papéis*, *produtos de trabalho* e *atividades*) podem ser adicionados, excluídos ou modificados em um processo. Isto é referido no artigo porque os autores afirmam que estas são as possíveis alterações que um processo pode sofrer.

Com relação às regras ou restrições de consistência para adaptação de processos, foi constatado que apenas em [Boe06] não existe a definição de regras para controlar as operações de adaptação. Outro resultado da análise sobre esse aspecto é que alguns autores propõem muitas operações de adaptação, contudo, no momento de apresentar as regras que guiam tais operações, não o fazem de forma completa, apresentando regras somente para parte das operações propostas. Nesse contexto, os trabalhos mais completos são os de [Yoo01] e [Dai07], nos quais há definição de regras de consistência para todas as operações propostas em seus estudos.

Em [Yoo01] as regras de consistência são definidas para as operações de inclusão, exclusão, união e divisão de tarefas e produtos de trabalho. As regras mais simples nesse estudo são definidas para a operação de inclusão de tarefas e produtos de trabalho, uma vez que elas apenas definem que, quando um destes elementos é incluído,

o mesmo deve ser devidamente relacionado com os outros elementos (também tarefas e produtos de trabalho) do processo. Resumidamente, para as operações de divisão de tarefas e produtos de trabalho, define-se, respectivamente, que os produtos de trabalho resultantes da divisão devem ser conectados nas mesmas tarefas que o produto de trabalho original era conectado, bem como que as novas subtarefas resultantes da divisão também devem ser conectadas com os mesmos produtos de trabalho que a tarefa original possuía associações. Para esta última regra, fica estabelecido em [Yoo01] que o responsável pela operação de divisão de tarefas deve verificar, após a execução da operação, se algum produto de trabalho não está sendo produzido por duas tarefas no processo resultante. Isso não deve ser permitido, pois viola a única regra de consistência para definição dos processos de software deste estudo (conforme Seção 3.3.1).

Prosseguindo com as regras de consistência definidas em [Yoo01], têm-se as regras para as operações de união de tarefas. Nesta operação, é definido que, quando duas tarefas são unidas, todas as suas entradas e saídas em termos de produtos de trabalho devem ser conectadas na tarefa resultante. Por fim, as regras para a operação de exclusão de tarefas definem que, quando uma tarefa é excluída de um processo de software, os produtos de trabalho gerados por esta tarefa serão também excluídos. Nesse contexto, quando um produto de trabalho é excluído, deve-se eliminar a associação dele com todas as tarefas remanescentes no processo. Em caso de alguma das tarefas afetadas ser a tarefa produtora do produto de trabalho excluído, ela também deverá ser excluída do processo. Por fim, caso existam produtos de trabalho no processo que dependem (relação de dependência) do produto de trabalho excluído, eles também deverão ser eliminados do processo.

Já o estudo de [Dai07] possui regras de consistência totalmente focadas no sequenciamento das tarefas de um processo de software. Como dito anteriormente, os autores propõem as operações de exclusão, adição, união e divisão de tarefas. Assim, esses autores apresentam regras para qualquer uma dessas operações, considerando que as tarefas estejam em sequência, sejam paralelas ou estejam envolvidas em um sequenciamento com decisão ou iteração (tarefas que são executadas repetidamente). Um exemplo da regra destes autores pode ser o seguinte: quando uma tarefa envolvida em uma estrutura de decisão é apagada, deve-se verificar se ainda existe a necessidade de manter tal estrutura ou definir apenas uma estrutura de sequência entre as atividades remanescentes. Analogamente, se a tarefa apagada é paralela a apenas uma outra tarefa

no processo, então, é necessário apagar a estrutura de paralelismo, criando apenas uma estrutura de sequência simples entre as tarefas remanescentes.

Prosseguindo com a avaliação, encontra-se o estudo de [Rui09], o qual propõe um conjunto bastante completo de operações: exclusão, inclusão, substituição, modificação e união de tarefas, produtos de trabalho e papéis. Contudo, nesse estudo, não são descritas regras para todas as operações propostas. Os autores somente apresentam as regras para adição e exclusão dos elementos referidos, as quais são definidas exatamente da mesma forma que as regras já apresentadas para o trabalho de [Yoo01].

No trabalho de [Baj07] os autores consideram que as regras de restrição (já descritas na Seção 3.3.1) utilizadas para modelagem de processos devem ser respeitadas durante a adaptação dos processos. Segundo esses autores, quando os fragmentos de processo opcionais não são selecionados para um projeto, é preciso que essa seleção não viole nenhuma regra de restrição.

Em [Oje09] e [Par11], analogamente ao critério de operações de adaptação, o critério das regras de consistência para adaptação de processos é definido de forma superficial. Especificamente em [Oje09], é explorado amplamente a definição de processos específicos para projetos, realizada a partir de um repositório de processos, o que é feito a partir da utilização de uma ferramenta denominada *agentTool Process Editor* – APE.

Nesta ferramenta, que é um *plugin* do eclipse e tem integração com o *Eclipse Process Framework* – EMF, existe um *framework* de validação para consistência dos processos gerados. Contudo, as regras implementadas dentro do *framework* de validação não são exploradas. A única regra explorada no estudo e mostrada como exemplo no *framework* de validação é a que checa se as entradas de uma tarefa já foram produzidas no processo. Segundo os autores desse estudo, tais entradas são definidas como as pré-condições para que uma tarefa possa ser executada. Ainda relacionado ao estudo de [OJE09], embora não citado pelos autores, considera-se que as regras de multiplicidades e atributos definidos em um modelo de classes UML também são verificadas. Isso porque a ferramenta APE, analogamente à ferramenta apresentada em [Fer10], é um *plugin* do eclipse baseado no EMF.

Em [Par11] constatou-se que não existem regras explícitas de consistência para adaptação de processos. O que os autores apresentam, nesse estudo, é como montar modelos de dependência dos elementos a partir de um processo. A ideia é a de que tais

modelos permitam realizar análise de impacto, quando um elemento for adicionado, excluído ou modificado.

Os modelos de dependência propostos por Park *et al.* são: (1) Modelo de dependências entre o fluxo das tarefas que mapeia as relações de precedência das tarefas em um processo; (2) Modelo de produtos de trabalho no processo que mapeia informações sobre composição, agregação e dependências entre produtos de trabalho em um processo de software; (3) Modelo de dependência entre papéis que mapeia as dependências entre papéis por suas competências. Isto quer dizer que, se papéis tem as mesmas competências, eles estão relacionadas no modelo de dependência de papéis. Tal assertiva serve para indicar que papéis relacionados podem desempenhar as mesmas tarefas (tarefa possui informação de competência necessária); (4) Modelo de dependência de atribuição que mapeia qual papel desempenha qual tarefa em um processo; e (5) Modelo de dependência *In/Out* que mapeia quais tarefas consomem e produzem cada produto de trabalho do processo.

Embora, como dito anteriormente, os modelos acima permitam mapear os impactos, quando o processo é alterado, não existe, em [Par11], uma descrição sobre as regras e/ou procedimentos a serem tomados durante uma alteração. Alguns indícios de regras são extraídos a partir de exemplos apresentados no estudo. Por exemplo, é definido em [Par11] que quando um papel é excluído de um processo e no modelo de dependência entre papéis existe um ou mais papéis relacionados o papel excluído, não existe impacto no processo que sofreu a modificação. Contudo, se não existem papéis relacionados ao papel excluído, então, uma ou mais tarefas poderão ser afetadas, pois tais tarefas podem não possuir mais relação com nenhum papel no processo e, desse modo, também necessitarão serem excluídas. Por conseqüência, um ou mais produtos de trabalho do processo igualmente podem ser afetados e causar novas exclusões. O que se observa, no estudo de [Par11], é que a grande contribuição desse trabalho está na riqueza da análise de impacto permitida pelos modelos de dependência propostos.

Por fim, o estudo de [Was06] define regras de consistência para adaptação de processos de forma um pouco diferente. Isso porque, como dito em momento anterior, esse estudo utiliza a ideia de linhas de processos para geração de um processo específico. Dessa forma, durante uma operação de substituição (utilizando-se das variantes de processo) ou exclusão (que será feita nos pontos opcionais) de elementos, várias regras são definidas para que os relacionamentos estabelecidos no diagrama de *features* sejam respeitados. O que se observa em [Was06] é que o foco de grande parte

das regras é o sequenciamento das tarefas em um processo. Contudo, como esse estudo é específico sobre linhas de processo, consideram-se suas regras aplicáveis a trabalhos que utilizem esse tipo de solução, o que não é caso, por exemplo, desta pesquisa.

3.3.2.1 Conclusões e Lacunas Identificadas para a Atividade de Adaptação de Processos

Como visto, podem existir diferentes formas para a adaptação dos processos de software e um mecanismo pode se mostrar mais completo que outro. Além disso, alguns autores preconizam o uso de um processo padrão, como é casos dos estudos de [Yoo01] e [Baj07], e outros definem que processos devem ser estabelecidos para cada projeto pela seleção de fragmentos de método, a partir de um repositório, como é o caso do estudo apresentado em [Oje09].

Especificamente sobre as operações de adaptação, identifica-se na análise dos estudos encontrados nesta revisão sistemática, que não existe um consenso na literatura analisada sobre quais as operações devem ser permitidas durante a adaptação dos processos. Alguns estudos propõem um conjunto maior de operações, como é o caso de [Yoo01], [DaiI07] e [Rui09], contudo, seus autores não justificam qual seria a motivação ou situação em que determinadas operações seriam necessárias em um projeto de software. Nesse contexto, as únicas operações que apresentam um consenso nos estudos são as de adição e exclusão de elementos e relacionamentos de um processo.

Com relação às regras para consistência que guiam as operações de adaptação, conclui-se que os trabalhos apresentam regras bem completas para suas operações, destacando-se os trabalhos apresentados em [Yoo01] e [RuiI09]. Observa-se ainda que as regras de consistência para adaptação têm como principal objetivo manter a consistência estabelecida durante a definição dos processos. Isto é dito, pois embora alguns estudos não tenham foco nas atividades de definição de processos, seus autores citam que o objetivo das regras utilizadas nas atividades de adaptação é manter a consistência do processo original nos processos adaptados.

Mesmo que exista, como dito acima, a preocupação nos trabalhos selecionados com a consistência dos processos durante as atividades de adaptação, observa-se um problema em alguns estudos que definem regras para consistência. O problema é relacionado com a falta de referências a importantes elementos do processo como, por exemplo, os trabalhos de [Yoo1], [Boe06], [Was06] e [Dai07] que não possuem referências ao elemento *papel* em seus trabalhos. Dessa forma, durante as operações de

adaptação e uso das regras para consistência, esse elemento não possui nenhum tratamento. Nos casos específicos de [Boe06] e [Dai07], também não existem referências ao elemento de processo *produto de trabalho*.

Como base contexto acima identificam-se, algumas lacunas específicas sobre ao tratamento da consistência durante a adaptação dos processos de software. Tais lacunas são:

- necessidade de maior investigação sobre quais operações de adaptação devem ser permitidas quando um processo de software é adaptado;
- especificamente sobre as regras de consistência, necessidade de definição de regras de consistência mais completas que envolvam os principais elementos e relações de um processo de software. Conforme já explicado na Seção 3.3.1.1 deste trabalho os principais elementos de um processo de software são *Atividades*, *Tarefas*, *Papéis* e *Produtos de Trabalho*.

3.4 Premissas para Consistência dos Processos de Software

Através da revisão sistemática conduzida nesta pesquisa, bem como dos estudos preliminares na literatura, do estudo aprofundado na especificação do metamodelo SPEM e dos processos de software OPEN, RUP e *OpenUp*, identificou-se um conjunto de premissas que são apliváveis em qualquer processo de software para seus principais elementos que são *Produto de Trabalho*, *Atividades, Tarefas* e *Papel.*

A Tabela 9 apresenta as premissas identificadas e descritas no contexto do metamodelo SPEM 2.0 que é utilizado como referência nesta pesquisa. Na Tabela 9, as premissas são listadas juntamente com as referências de onde elas foram extraídas. Adicionalmente, para facilitar a identificação de cada premissa no restante deste trabalho, um identificador foi criado para todas elas e é também relacionado na Tabela 9. Torna-se também importante considerar que as referências listadas não são as únicas a mencionar individualmente cada premissa, porém, para a grande maioria delas, pelo menos uma referência está presente. Nas últimas linhas da Tabela 9, se apresentam algumas premissas que não estão associadas com nenhuma referência da literatura por terem sido definidas especificamente nesta pesquisa. A justificativa para tais definições é descrita a seguir juntamente com a explicação de cada uma das premissas.

Tabela 9 – Premissas para os processos de software descritas no contexto do metamodelo SPEM 2.0

Identificador	Premissa	Principais Referências
P #1	Projetos de software podem consumir produtos de trabalho externos ao projeto.	[Omg07a]
P #2	Produtos de trabalho externos são consumidos e/ou modificados em um projeto de software.	[OmgG07a]
P #3	Todo produto de trabalho que não é externo a um projeto de software deve ser produzido no próprio projeto de software.	[Yoo01] [Atk07]
P #4	Produtos de trabalho podem assumir relações de agregação com outros produtos de trabalho em um processo de software.	[KruU00] [Omg07a] [Par11]
P #5	Produtos de trabalho podem assumir relações de dependência com outros produtos de trabalho em um processo de software.	[Yoo01] [Baj07] [Omg07a] [Par11]
P #6	Todo produto de trabalho deverá possuir um responsável associado a sua produção.	[KruU00] [Fir02]
P #7	Toda tarefa de um projeto de software deve ser desempenhada por um papel e deve produzir um resultado de valor observável em termos de produto de trabalho.	[Kru00] [Atk07] [Hsu08]
P #8	Todas as dependências (em termos de produtos de trabalho) de um produto de trabalho devem estar disponíveis no momento da sua produção.	[Yoo01]
P #9	Todas as entradas de uma tarefa devem estar disponíveis no momento de sua execução.	[Atk07]
P #10	O seqüenciamento das atividades ou tarefas não deverá apresentar situações que representem *deadlocks* para a execução de um processo de software.	[Was06] [Dai07] [Bao08]
P #11	O processo de software deverá possuir um nodo inicial e um nodo final.	[Dai07] [Bao08]
P #12	Todo sequenciamento entre as atividades ou tarefas deverá ter seu começo no nodo inicial e seu término no nodo final do processo de software.	[Bao08]
P #13	Quando atividades compostas por outras atividades e/ou tarefas definem seus parâmetros de entrada e/ou saída em termos de produtos de trabalho, estes parâmetros devem ser compatíveis com os parâmetros de entrada e/ou saída definidos nas suas atividades e tarefas internas.	[Kru00] [Rui09]
P #14	Todo papel deverá participar na execução de pelo menos uma tarefa.	[Kru00] [Fir02]
P #15	Atividades, tarefas, produtos de trabalho e papéis podem ser opcionais em um projeto de software.	[Was06] [Omg07a]
P #16	Todo produto de trabalho obrigatório deverá estar associado a pelo menos uma tarefa obrigatória e um papel obrigatório.	-
P #17	Todo papel obrigatório deverá estar associado a execução de pelo menos uma tarefa obrigatória.	-
P #18	Toda tarefa obrigatória deverá estar associada a pelo menos um produto de trabalho obrigatório e um papel obrigatório para sua execução.	-
P #19	Toda atividade opcional não deverá possuir como uma de suas partes nenhum elemento de processo obrigatório.	-
P #20	Toda atividade obrigatória deverá possuir como uma de suas partes ao menos uma tarefa obrigatória.	-
P #21	Instâncias de produtos de trabalho e papéis que possuem nomes iguais não devem ser criados em um processo de software.	-
P #22	Instâncias de relacionamentos que representem mais de uma vez a mesma relação entre dois elementos de processo não devem ser criados.	-

As premissas P#1 e P#2, relacionadas a definição de produtos de trabalho externos a um projeto de software foram definidas com o objetivo de: (1) contemplar os produtos de trabalho de projetos de manutenção de software que em sua grande maioria estão prontos necessitando apenas de atualizações; e (2) contemplar produtos de trabalho padrões definidos nas organizações para serem somente consumidos nos projetos. Neste último caso especificamente, cita-se como exemplo um plano de medições e análise que pode ser definido de forma corporativa e pode ser apenas consumido como uma referência em todos os projetos de software de uma organização [Omg07a].

A premissa P#3 estabelece que todos os produtos de trabalho do tipo interno deverão ser produzidos em pelo menos um ponto no ciclo de vida de um projeto de software. Esta premissa é relacionada em vários estudos da literatura, tais como em [Yoo01] e [Atk07], e também nos processos de software OPEN, RUP e *OpenUP*.

As premissas P#4 e P#5 dizem respeito aos tipos de relacionamentos que os produtos de trabalho podem assumir com outros produtos de trabalho em um projeto de software. Especificamente a premissa P#4 considera que produtos de trabalho podem ser compostos (assumindo relações de agregação simples e agregação composta) por outros produtos de trabalho. Vários exemplos de produtos de trabalho com este tipo de relação podem ser encontrados em ambos os processos RUP e OPEN [Kru00], [Fir02].

Já a premissa P#5 estabelece que produtos de trabalho podem assumir relações de dependência com outros produtos de trabalho ao longo ciclo de vida de um processo de software. Particularmente durante a definição dos processos de software, relacionamentos de dependência são necessários para construção do fluxo de produção dos produtos de trabalho e definição das entradas das tarefas [Atk07]. Durante as atividades de adaptação, este tipo de relacionamento assume um papel essencial, pois é preciso conhecê-los para que inconsistências não sejam geradas no processo resultante. Em operações de exclusão de elementos, por exemplo, torna-se necessário conhecer a relação de dependência entre produtos de trabalho para que seja possível impedir que determinados produtos de trabalho sejam excluídos do processo enquanto outros produtos de trabalho que dependem do produto de trabalho excluído fizerem parte do mesmo [Yoo01].

A premissa P#6 relaciona-se com o fato que todo produto de trabalho possui um papel responsável que deve participar de sua tareafa de produção. No caso específico

dos processos RUP e OPEN, identificou-se que todos os produtos de trabalho destes processos tem a participação de seus papéis responsáveis em suas tarefas de produção. Desta forma, esta premissa foi criada e se aplica a qualquer processo de software que prevê a relação de responsabilidade entre papéis e produtos de trabalho.

A definição da premissa P#7 é baseada no fato de que tarefas em um processo de software deverão produzir um resultado, criando ou modificando um ou mais produtos de trabalho. Essa definição é facilmente encontrada na literatura [Atk07] e [Hsu08], bem como nos processos de software, tais como o RUP e o OPEN. Além de produzir ou modificar um ou mais produtos de trabalho, essa premissa também estabelece que uma tarefa é sempre desempenhada por pelo menos um papel.

As premissas P#8 e P#9 estão relacionadas com a disponibilidade de produtos de trabalho para execução das tarefas em um processo de software. Especificamente a premissa P#8 define que todas as dependências (em termos de produtos de trabalho) de um produto de trabalho devem estar disponíveis neste processo no momento de sua produção. Isso quer dizer que todo relacionamento de dependência entre produtos de trabalho estabelece que um produto de trabalho depende de um ou mais produtos de trabalho para ser produzido em um processo de software [Yoo01], [Par11].

A premissa P#9 define que, quando um produto de trabalho é utilizado em um processo de software, ele deve estar disponível. Essa premissa é específica para produtos de trabalho do tipo interno e implica que quando um produto de trabalho é consumido em um processo de software é necessário que esse produto de trabalho esteja associado a pelo menos uma atividade de produção (ou seja, sua produção já deve ser sido iniciada no processo de software) e que exista pelo menos um caminho entre essa tarefa e as tarefas que consomem o produto de trabalho [Atk07].

As premissas P#10, P#11 e P#12 relacionam-se com o sequenciamento entre as atividades ou tarefas do metamodelo SPEM 2.0. Elas definem as condições que um sequenciamento estabelecido, entre as atividades ou tarefas em um processo de software, deverá atender para ser consistente.

Em vários processos de software, incluindo aqueles definidos a partir do metamodelo SPEM 2.0, atividades podem ser compostas por outras atividades e/ou tarefas. Para esses casos, torna-se necessário garantir a consistência dos produtos de trabalho definidos como entrada e saída para estas atividades e tarefas. Em outras palavras, é necessário verificar se as entradas e saídas definidas para uma atividade estão definidas igualmente em pelo menos uma de suas tarefas internas. Não é possível,

por exemplo, no caso específico do RUP que uma atividade (no RUP um *workflow detail*) possua um parâmetro de saída indicando a produção de um produto de trabalho sendo que suas tarefas (no RUP as tarefas) internas não produzem tal produto de trabalho em nenhum ponto da execução desta atividade. Baseado neste contexto a premissa P#13 estabelece que quando uma atividade é composta por outras atividades e/ou tarefas e estabelece entradas e saídas em termos de produtos de trabalho, estas mesmas entradas e saídas, devem obrigatoriamente, estar definidas em uma de suas tarefas internas.

A premissa P#14 é básica e define que um papel deverá participar na execução de pelo menos uma tarefa. Essa definição é comum na literatura e pode ser encontrada explicitamente em [Kru00] e [Fir02].

As premissas P#15, P#16, P#17, P#18, P#19 e P#20 estão relacionadas com os aspectos de opcionalidade dos elementos em um processo de software. Especificamente a premissa P#15 estabelece que elementos tais como papel, atividade, tarefa e produto de trabalho podem ser opcionais em projetos de software. Esta premissa é comumente encontrada na literatura sobre adaptação de processos. Isso porque vários dos elementos opcionais em um processo poderão ser excluídos em um procedimento de adaptação.

As premissas P#16, P#17, P#18, P#19 e P#20 estabelecem como os elementos obrigatórios deverão estar relacionados em um processo de software e, ainda, como uma atividade opcional deverá ser definida. Não é possível, por exemplo, estabelecer uma tarefa obrigatória em um processo de software que é executada somente por papéis definidos como opcionais. Isso poderia facilmente gerar um inconsistência durante uma atividade de adaptação do processo em questão para um projeto específico, uma vez que os papéis opcionais poderiam ser excluídos deixando a tarefa obrigatória (que não poderia ser excluída) sem executores. Outro exemplo que poderia gerar inconsistências durante as atividades de adaptação seria a definição de uma atividade opcional que é composta por elementos de processo obrigatórios. Em um procedimento de adaptação não seria possível excluir a atividade em questão (que é definida como opcional), uma vez que esta atividade contém elementos que são obrigatórios e, portanto, não podem ser excluídos em nenhum projeto de software. Embora as premissas P#16, P#17, P#18, P#19 e P#20 não estejam relacionadas com nenhuma referência da literatura analisada, suas definições foram necessárias nesta pesquisa para garantir que as informações sobre as opcionalidades dos elementos em um processo de software, que é definido a partir do metamodelo SPEM 2.0, sejam consistentes (de acordo com premissa P#15).

Por fim, as premissas P#21 e P#22 estabelecem que alguns elementos e relacionamentos não devem ser definidos mais de um vez em um processo de software. Por exemplo, não devem ser definidos dois ou mais papéis denominados como *Tester,* ou ainda, dois ou mais produtos de trabalho denominados como *Vision* em um processo de software*.* Especificamente considerando os relacionamentos, define-se, por exemplo, que não devem existir dois relacionamentos no mesmo processo que estabeleçam o papel *Analyst* como responsável pelo produto de trabalho *Use Case*. Observa-se na Tabela 9 que as premissas P#21 e P#22 não estão ligadas a nenhuma referência da literatura analisada, uma vez que não foi possível encontrar referências explícitas nos estudos sobre as informações relacionadas com tais premissas.

3.5 Considerações Finais

Este capítulo identificou as principais lacunas de pesquisa sobre o tratamento da consistência durante a definição e adaptação dos processos de software e também definiu um conjunto de premissas relacionadas com o aspecto de consistência que devem ser atendidas em qualquer processo de software. A seguir, com base no conteúdo apresentado até este momento, a solução proposta nesta pesquisa para o tratamento da consistência dos processos de software começa a ser apresentada. No próximo capítulo, uma extensão ao metamodelo SPEM 2.0 e um conjunto de regras de boa-formação para consistência serão descritos, visando o atendimento das premissas apresentadas nesse capítulo.

4 INFRAESTRUTURA PARA CONSISTÊNCIA DOS PROCESSOS DE SOFTWARE

Este capítulo apresenta uma extensão ao metamodelo SPEM 2.0, que é nomeada como sSPEM 2.0 (*con**S**istent SPEM 2.0),* e um conjunto de regras de boa-formação relacionado com o aspecto da consistência dos processos de software. Ambos, o metamodelo sSPEM 2.0 e o conjunto de regras de boa-formação, são partes da infraestrutura para consistência que está sendo definida nesta pesquisa e visam atender as premissas para consistência dos processos de software, as quais já foram apresentadas no capítulo anterior deste trabalho.

O metamodelo sSPEM 2.0 inclui um conjunto de elementos e relacionamentos, bem como realiza alterações no metamodelo original SPEM 2.0. Esse metamodelo será utilizado para condução de todas atividades de definição e adaptação de processos nesta pesquisa. Já as regras de boa-formação são estabelecidas com base nos elementos e relacionamentos do metamodelo sSPEM 2.0 e estabelecem as condições que devem ser verdadeiras (premissas) para todos os processos de software em qualquer ponto do seu ciclo de vida, não se alterando sobre qualquer conjunto de transformações. No contexto desta pesquisa, transformações em um processo de software são consideradas como as modificações que um processo pode sofrer, decorrente das atividades de adaptação para projetos em específico.

Nesta pesquisa, foram definidas 50 regras de boa-formação. A ideia é que estas regras sejam utilizadas para especificar os elementos e relacionamentos de um processo de software de maneira coerente e, assim, evite que possíveis inconsistências estejam presentes nos projetos de software.

A seção seguinte identifica quais os elementos e relacionamentos que foram tratados na extensão realizada ao metamodelo SPEM 2.0. As outras seções deste capítulo descrevem, respectivamente, o metamodelo sSPEM 2.0, as regras de boa-formação e também os mecanismos de adaptação do metamodelo sSPEM 2.0.

4.1 Identificação do Escopo

As alterações realizadas no metamodelo original SPEM 2.0 e todo o conjunto de regras de boa-formação criado, foram definidos utilizando os elementos de processo *Atividade*, *Tarefa*, *Produto de Trabalho* e *Papel* e todos os relacionamentos existentes

entre estes elementos nos pacotes *Process Structure*, *Process with Methods* e *Method Plugin* do metamodelo original SPEM 2.0. Adicionalmente, algumas alterações e regras, as quais foram consideradas pertinentes, foram também definidas para estes elementos e relacionamentos no repositório de conteúdo, o qual é representado pelo pacote *Method Content.*

A motivação para escolha dos elementos acima se deu em função deles serem considerados pela OMG, em [Omg02] e [Omg07a], como os principais elementos de um processo de software. Tal fato se comprova ao analisar-se processos como o RUP e OPEN, onde com os elementos *Atividade*, *Tarefa*, *Produto de Trabalho* e *Papel* é possível modelar quase que a totalidade dos elementos desses processos. No caso específico do RUP, por exemplo, os únicos elementos adicionais seriam o elemento ferramenta (em inglês *tool*) e o elemento disciplina (em inglês *discipline*), os quais, no SPEM 2.0, são definidos apenas no pacote *Method Content.*

Com relação aos pacotes do metamodelo original SPEM 2.0, que somam um total de sete pacotes, considerou-se, nesta pesquisa, que cinco deles serão tratados, pois, além dos quatro pacotes citados anteriormente, tem-se o pacote *Core* que define a base para todos os demais pacotes do metamodelo original SPEM 2.0 e que, através do mecanismo de *merge,* tem seus elementos presentes nos outros pacotes desse metamodelo. Não serão tratados nesta pesquisa os pacotes *Managed Content* e *Process Behavior,* tendo em vista que: (1) o pacote *Process Behavior* não é implementado pelo metamodelo original SPEM 2.0 (conforme explicado na Seção *2.6.1.4*); e (2) o pacote *Managed Content* que não possui nenhum dos elementos tratados nesta pesquisa. Este pacote define apenas conceitos para gerenciar as descrições textuais para os processos.

4.2 Camadas de Metamodelagem

O metamodelo sPEM 2.0 foi incluído nas camadas de metamodelagem propostas pela OMG, conforme mostrado na Figura 25. Observa-se nesta figura que o esse metamodelo é definido na camada M2 como uma instância do MOF 2.0 definido na camada M3. O metamodelo sSPEM 2.0 se relaciona por meio de um *extends* com o metamodelo SPEM 2.0, caracterizando-se como uma extensão deste metamodelo.

As últimas camadas (M1 e M0) mostram, respectivamente, os modelos gerados a partir dos metamodelos definidos na camada M2 e as instâncias de tais modelos. Destaca-se que a grande diferença entre os modelos gerados a partir do metamodelo

SPEM 2.0 e do metamodelo sSPEM 2.0 reside no fato de que os modelos gerados a partir do sSPEM 2.0 poderão ser verificados através das regras de boa-formação para consistência.

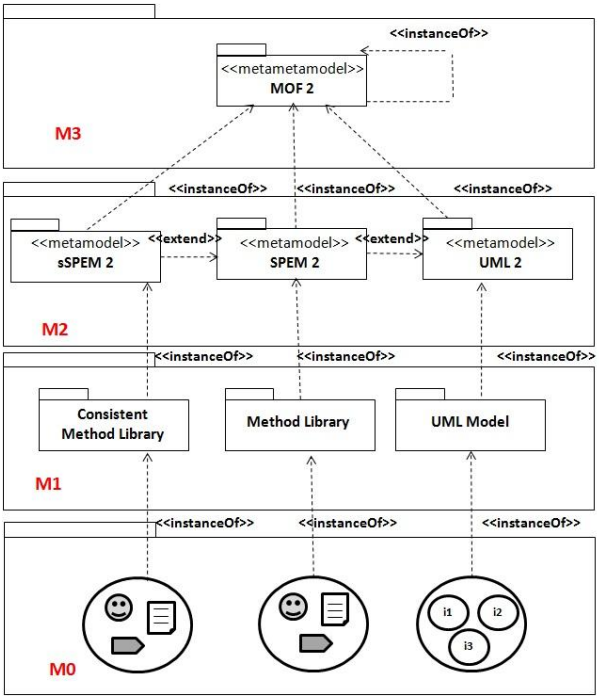

Figura 25 – Organização do metamodelo sSPEM 2.0 nas camadas de metamodelagem propostas pela OMG

4.3 Extensão do Metamodelo SPEM 2.0 – sSPEM 2.0

Estender um metamodelo baseado na UML, como é caso do metamodelo SPEM 2.0, pode ser realizado através de duas estratégias conhecidas como *lightweight built-in extension* e *heavyweight extensibility mechanism* [Omg03]. A primeira estratégia é utilizada, quando não é necessário modificar diretamente os elementos e relações de um metamodelo. Tal estratégia é dita uma extensão do tipo conservativa e se utiliza de um mecanismo chamado *profiles mechanism* que se baseia apenas em *stereotypes* e *tagged values*.

A extensão *heavyweight* é utilizada em um metamodelo baseado na UML quando existe a necessidade de inclusão de novas metaclasses e de outros metaconstrutores, ou ainda, quando é necessário remover ou alterar metaclasses existentes [Omg06]. Esse tipo de extensão é considerada não conservativa e, por este fato, impossibilita a troca de dados entre ferramentas que implementam o metamodelo padrão da UML. Geralmente, uma extensão desse tipo deve ser acompanhada de ferramentas que suportem as

alterações feitas. A principal vantagem da utilização do tipo de extensão *heavyweight* é que ela permite a criação de uma semântica muito mais rica do que um UML *Profile*. Além disso, as restrições podem ser mais elaboradas, permitindo a inserção de regras em diversas linguagens [War03].

Nesta pesquisa, o tipo de extensão utilizada foi a *heavyweight*. Essa escolha foi feita porque, na extensão proposta, um conjunto de elementos, relacionamentos e regras de boa-formação foram incluídos ao metamodelo original SPEM 2.0 e, adicionalmente, alguns de seus elementos e relacionamentos existentes foram modificados.

A seguir, é apresentada uma descrição detalhada das alterações e inclusões realizadas em cada pacote do metamodelo original SPEM 2.0. Primeiramente, para cada pacote, são descritas as inclusões e modificações realizadas diretamente sobre suas metaclasses e relacionamentos; e, em seguida, o conjunto de regras de boa-formação descrito em linguagem natural é exposto. Para facilitar o entendimento e justificar as alterações realizadas nos pacotes do metamodelo original SPEM 2.0, cada modificação apresentada será associada a uma ou mais premissas para consistência (conforme Seção 3.4) e, também, as regras de boa-formação definidas nesta pesquisa. Além disso, durante a descrição das regras de boa-formação, para cada regra, será apresentada sua semântica juntamente com um identificador único que é utilizado no restante deste estudo. Para as regras mais complexas, também serão mostrados exemplos e explicações adicionais.

Todavia, antes de apresentar o metamodelo sSPEM 2.0, como forma de facilitar o conteúdo que será descrito, a Tabela 10 mostra a relação entre os conceitos utilizados neste trabalho para os elementos de um processo de software e as metaclasses do metamodelo sSPEM 2.0.

Tabela 10 – Relação dos Elementos de Processo e das Metaclasses do Metamodelo SPEM 2.0

Elemento	**Metaclasse**	**Pacote**
Tarefa	*TaskUse*	*Process with Methods*
	TaskDefinition	*Method Content*
Atividade	*Activity*	*Process Structure* *Process with Methods*
Produto de Trabalho	*WorkProductUse*	*Process Structure* *Process with Methods*
	WorkProductDefinition	*Method Content*
Papel	*RoleUse*	*Process Structure* *Process with Methods*
	RoleDefinition	*Method Content*

Na sequência, serão descritos os pacotes do metamodelo sSPEM 2.0:

4.3.1 Pacote *Process Structure*

Esta seção relaciona as alterações e regras de boa-formação definidas especificamente para o pacote *Process Structure*. Como forma de auxiliar no entendimento do conteúdo apresentado, a Tabela 11 relaciona cada alteração e as regras de boa-formação incluídas no pacote *Process Structure* com as premissas definidas nesta pesquisa.

Na Tabela 11, especificamente nas últimas linhas, várias regras de boa-formação, não estão relacionadas com nenhuma alteração do metamodelo. Isso ocorre, pois em alguns casos, para o atendimento de algumas premissas, não são necessárias mudanças sobre metaclasses e relações do metamodelo, mas sim a inclusão de regras que definam algumas condições de instanciação destes elementos em um processo de software. Na Tabela 11, pode-se observar, também, que existe uma modificação que não é relacionada com nenhuma premissa e com nenhuma regra de boa-formação. Isso ocorre, pois especificamente esta alteração foi realizada para permitir a criação de um procedimento de análise de impacto para os mecanimos de adaptação do metamodelo sSPEM 2.0, que serão descritos a seguir, na Seção 4.4 deste capítulo.

Tabela 11 – Relação de alterações e regras de boa-formação incluídas no pacote *Process Structure*

Alteração	**Regra(s) de Boa-Formação**	**Premissa(s)**
Inclusão das Metaclasses *InternalUse* e *ExternalUse.*	Regra #1, Regra #2, Regra #3	P #1 P #2 P #3
Inclusão do atributo *relationType* para metaclasse *WorkProductUseRelationship* e da metaclasse de enumeração *RelationType*.	Regra #5, Regra #7, Regra #49	P #4 P #5
Redefinição do relacionamento entre as metaclasses *ProcessResponsabilityAssignment* e *WorkProductUse*.	Regra #8	P #6
Redefinição do relacionamento entre as metaclasses *WorkProductUse* e *ProcessParameter*.	Regra #4	P #7
Inclusão do atributo *specialNode* para a metaclasse *WorkBreakdownElement* e da metaclasse de enumeração *WorkBreakdownElementKind*.	Regra #12, Regra #13, Regra #14, Regra #15, Regra #16, Regra #17, Regra #18, Regra #41, Regra #42, Regra #43, Regra #44, Regra #45	P #10 P #11 P #12
Inclusão do atributo *optionality* para a metaclasse *ProcessParameter* e da metaclasse de enumeração *OptionalityKind*.	**-**	**-**
-	Regra #25, Regra #29	P #15 P #19
-	Regra #33, Regra #34, Regra #36, Regra #37,	P #21 P#22

	Regra #38, Regra #39	

4.3.1.1 Alterações sobre Metaclasses e Relações do Pacote Process Structure

O primeiro conjunto de alterações realizadas no pacote *Process Structure* está relacionada com os elementos *WorkProductUse* e *WorkProductUseRelationship*, conforme pode ser visto na Figura 26 (indicado pela Letra A). Especificamente, a alteração realizada diretamente sobre o elemento *WorkProductUse* envolve a inclusão das metaclasses *InternalUse* e *ExternalUse* como especializações dessa metaclasse. Como consequência, a metaclasse *WorkProductUse* passsa a ser abstrata no pacote *Process Structure*.

O objetivo de incluir as metaclasses *InternalUse* e *ExternalUse* foi diferenciar quando um produto de trabalho é produzido ou somente consumido e/ou modificado em um processo de software. Nesse sentido, a metaclasse *InternalUse* deve ser utilizada toda vez que um produto de trabalho é obrigatoriamente produzido em um determinado processo de software e a metaclasse *ExternalUse* deve ser utilizada para representar produtos de trabalho que são apenas consumidos e/ou modificados em tal processo.

A modificação realizada sobre o elemento *WorkProductUseRelationship* envolve a inclusão do atributo *relationType* (indicado pela Letra B na Figura 26). Ainda relacionado a este elemento, foi incluída uma metaclasse do tipo enumeração chamada *WorkProductRelationshipKind* que define os valores válidos (*dependência*, *composição* e *agregação*) para o atributo *relationType,* conforme mostrado na Figura 26 (indicado pela Letra C).

Tanto o atributo *relationType* quanto a metaclasse de enumeração *WorkProductRelationshipKind* foram definidas no pacote *Process Structure* para estabelecer quais são os tipos de relacionamentos que os produtos de trabalho podem assumir entre si. Isso foi feito, porque, originalmente, na especificação do metamodelo SPEM 2.0, já há a consideração de que produtos de trabalho podem assumir relações entre si através da classe *WorkProductUseRelationship,* contudo, como os tipos de relações não são predefinidas, elas não possuem nenhum tipo de regra de boa-formação relacionada com consistência.

Nesta pesquisa, os tipos de relações permitidas entre produtos de trabalho foram definidas com base na literatura e são os de *dependência*, *agregação* e *composição*. A relação de dependência é citada por vários estudos, tais como [Yoo01], [Baj07], [Omg07a], [Par11], e estabelece que um produto de trabalho (*source*) depende de um ou

mais produtos de trabalho (*target*) para ser produzido em um processo de software. Já as relações de agregação e composição possuem o mesmo significado de relações da UML e estabelecem que um produto de trabalho (*source*) pode ser composto por um ou mais produtos de trabalho (*target*) em um processo de software. A diferença entre as relações de agregação e composição é que, assim como na UML, uma relação de composição estabelece uma relação mais forte entre os produtos de trabalho. Nesse tipo de relação, o(s) produto(s) de trabalho que representa(m) a(s) parte(s) na relação não faz(em) sentido sem o produto de trabalho que representa o todo desta relação.

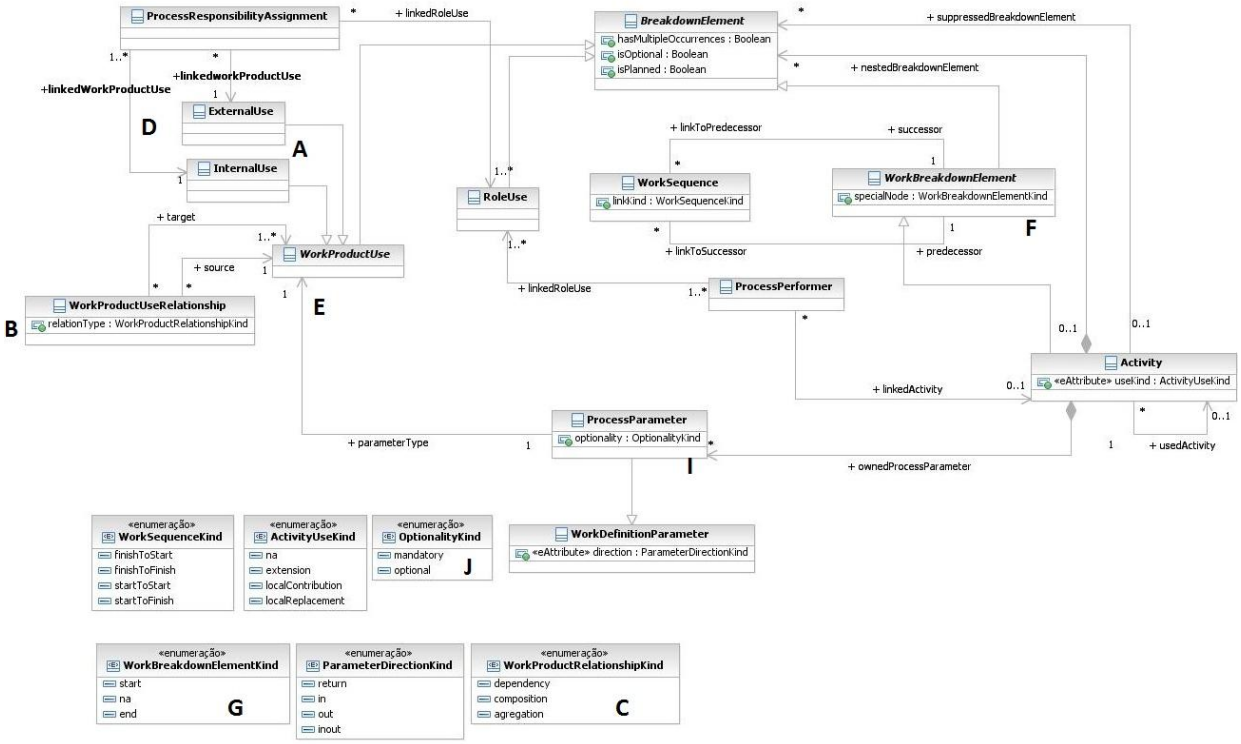

Figura 26 – Metaclasses e relações incluídas e/ou alteradas no pacote *Process Structure*

Especificamente, no contexto dos processos de software, analisando os estudos de [Kru00], [Omg07a], [Par11], os quais citam as relações de agregação e composição, verificou-se que, em uma relação de composição o produto de trabalho que representa o todo também não faz sentido sem o(s) produto(s) de trabalho que representa(m) sua(s) parte(s). Nesse sentido, mudanças também foram feitas sobre alguns relacionamentos e multiplicidades do pacote *Process Structure,* sendo a primeira delas realizada no relacionamento entre as metaclasses *WorkProductUse* e *ProcessResponsabilityAssignment.*

Originalmente, no metamodelo SPEM 2.0, é estabelecido que um produto de trabalho pode estar associado a zero ou vários papéis responsáveis em um processo de

software, o que é representado com a multiplicidade 0..* no relacionamento entre as metaclasses *WorkProductUse* e *ProcessResponsabilityAssignment*. Mas, neste trabalho, como a premissa #P6 define que todo produto de trabalho que é produzido em um processo de software deve possuir ao menos um papel responsável por tal produção, o relacionamento da metaclasse *ProcessResponsabilityAssignment,* que antes era com a metaclasse *WorkProductUse,* foi redefinido, passando a ser estabelecido no pacote *Process Structure* com as metaclasses *ExternalUse* e *InternalUse* (conforme indicado pela Letra D na Figura 26).

Dessa forma, a multiplicidade estabelecida para tais relações foram, respectivamente, 0..*, que estabelece que produtos de trabalho do tipo *ExternalUse* podem ou não ter papéis responsáveis em um processo de software, e 1..*, que estabelece que todo produto de trabalho do tipo *InternalUse* deverá estar associado a responsabilidade de ao menos um papel nos processos de software.

Outra modificação realizada sobre multiplicidade dos relacionamentos estabelecidos no pacote *Process Structure* foi feita na relação estabelecida entre as metaclasses *ProcessParameter* e *WorkProductUse*. Originalmente, no pacote *Process Structure* do metamodelo original SPEM 2.0 uma instância da metaclasse *ProcessParameter* pode estar associada a zero ou um produto de trabalho. Porém, na solução proposta nesta pesquisa, toda instância da metaclasse *ProcessParameter* deve estar associada a exatamente um produto de trabalho. Dessa forma, a multiplicidade da relação entre *ProcessParameter* e *WorkProductUse* foi alterada de 0..1 para exatamente 1, conforme indicado pela Letra E da Figura 26.

As últimas alterações realizadas sobre as metaclasses e relações do pacote *Process Strucute* são a inclusão de alguns atributos para metaclasses existentes e inclusão de algumas metaclasses de enumeração. As primeiras inclusões são o atributo *specialNode* para a metaclasse abstrata *WorkBreakdownElement,* e a inclusão da metaclasse de enumeração, também chamada *WorkBreakdownElementKind* com os valores *none*, *start* e *end*, conforme pode ser visto na Figura 26 pelas Letras F e G, respectivamente. Tanto o novo atributo quanto a nova metaclasse de enumeração permitem a definição de atividades que representem o ponto inicial e final de um processo de software, utilizando-se, para isso, respectivamente, dos valores *start* e *end* da metaclasse *WorkBreakdownElementKind*. Já o valor *none* da metaclasse *WorkBreakdownElementKind* deve ser utilizado para todas as outras atividades do processo de software.

Por fim, como mostrado pelas Letras I e J da Figura 26, incluiu-se um atributo chamado *optionality* também para a metaclasse abstrata *ProcessParameter* juntamente com uma metaclasse de enumeração chamada *OptionalityKind* que define os valores válidos (*mandatory* e *optional)* para esse atributo. O motivo para inclusão do atributo *optionality* e da metaclasse *OptionalityKind* é diferenciar quando um parâmetro de entrada e/ou saída em uma atividade é obrigatório ou opcional para sua execução. Essa informação é essencial durante as operações de adaptação, uma vez que toda atividade que possui parâmetros obrigatórios depende deles para ser executada. Assim, caso um desses parâmetros seja excluído, a atividade a qual pertece também deverá ser eliminada.

4.3.1.2 Regras de Boa-Formação do Pacote Process Structure

Esta seção apresenta o conjunto de regras de boa-formação para consistência definido no pacote *Process Structure.* Neste, encontra-se o maior número de regras de boa-formação contemplando grande parte das premissas para consistência de processos propostas nesta pesquisa (conforme Seção 3.4).

As primeiras regras de boa-formação relacionam-se com as metaclasses *ExternalUse* e *InternalUse* e tem como objetivo garantir que os produtos de trabalho do tipo interno serão produzidos em um processo de software e que os produtos de trabalho do tipo externo serão apenas consumidos e/ou modificados em tal processo. Tais regras são:

Regra #1 - Os produtos de trabalho externos (ExternalUse) não podem ser produzidos em um processo de software.

Semântica: Nenhuma instância da metaclasse *ExternalUse* pode possuir relação com instâncias da metaclasse *ProcessParameter* que possuem o valor do atributo *direction* igual a “*out*”.

Regra #2 - Os produtos de trabalho externos (ExternalUse) devem ser consumidos e/ou modificados em um processo de software.

Semântica*:* Toda instância da metaclasse *ExternalUse* deve ter pelo menos uma relação com uma instância da metaclasse *ProcessParameter* que tenha o valor do atributo *direction* igual a “*in*” ou “*inout*”.

Regra #3 - Os produtos de trabalho internos (InternalUse) devem ser produzidos em um processo de software.

Semântica*:* Toda instância da metaclasse *InternalUse* deve ter pelo menos uma relação com uma instância da metaclasse *ProcessParameter* que tenha o valor do atributo *direction* igual a "*out*".

As próximas regras de boa-formação são relacionadas com as metaclasses *InternalUse, ExternalUse* e *WorkProductRelationshipUse* e são específicas sobre as relações de *dependência*, *agregação* e *composição* que os produtos de trabalho do tipo externo ou interno podem assumir em um processo de software. A definição de tais regras foi realizada para respeitar as propriedades de reflexividade, simetria e transitividade das relações *dependência*, *agregação* e *composição*.

Considerando a propriedade de reflexividade, diz-se que todas as relações de *dependência*, *agregação* e *composição* entre produtos de trabalho são irreflexivas, ou seja, para todas essas relações, produtos de trabalho não poderão estar relacionados consigo mesmo. Considerando a relação de composição, por exemplo, produtos de trabalho não podem ser definidos como composições de si próprios. O mesmo vale para as relações de dependência e agregação.

Com relação à transitividade, define-se que todas as relações de *dependência*, *agregação* e *composição* entre produtos de trabalho são transitivas, ou seja, elas se transmitem em cadeia. Por exemplo, dado que um produto de trabalho A é composto de um produto de trabalho B e B é composto de um produto de trabalho C, então A também é composto de C. O mesmo se aplica para as relações de dependência e agregação.

Por fim, relacionado a propriedade de simetria, estabelece-se que todas as relações de *dependência*, *agregação* e *composição* entre produtos de trabalho são assimétricas. Por exemplo, se o produto de trabalho A é composto pelo produto de trabalho B, então o produto de trabalho B não é composto pelo produto de trabalho A. O mesmo é válido para as relações de *dependência* e *agregação*.

Além das propriedades acima, uma regra específica também foi criada para a relação de *composição* entre produtos de trabalho. Tal regra define que um produto de trabalho (interno e externo) não pode estar envolvido como uma "parte" em mais de um relacionamento de composição. Isso respeita a definição dessa relação estabelecida na literatura sobre processos de software e também na linguagem UML.

Baseado no contexto acima, as seguintes regras de boa-formação foram definidas:

Regra #5 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode ser o "todo" em um relacionamento de composição se uma de suas "partes" já representa o seu "todo" em outro relacionamento de composição ou representa o seu "todo" pela transitividade da relação de composição.

Semântica: Dado que existe uma instância da metaclasse *WorkProductUseRelationship 1* com o seguintes valores de atributos: *relationType* igual a *composition*; *source* associado com a instância da metaclasse *InternalUse A*; e *target* associado com as instâncias da metaclasse *ExternalUse B* e *C*, não pode existir outra instância da metaclasse *WorkProductUseRelationship* com os seguintes valores de atributos: *relationType* igual a *composition*; *source* associado com a instância da metaclasse *ExternalUse B* ou *C*; e *target* associado com a instância da metaclasse *InternalUse A*.

Regra #5 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode ser o "todo" em um relacionamento de agregação se uma de suas "partes" já representa o seu "todo" em outro relacionamento de agregação ou representa o seu "todo" pela transitividade da relação de agregação.

Semântica: Dado que existe uma instância da metaclasse *WorkProductUseRelationship 1* com o seguintes valores de atributos: *relationType* igual a *agreggation*; *source* associado com a instância da metaclasse *InternalUse A*; e *target* associado com as instâncias da metaclasse *ExternalUse B* e *C*, não pode existir outra instância da metaclasse *WorkProductUseRelationship* com os seguintes valores de atributos: *relationType* igual a *agreggation*; *source* associado com a instância da metaclasse *ExternalUse B* ou *C*; e *target* associado com a instância da metaclasse *InternalUse A*.

Regra #5 - ***Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode depender de um produto de trabalho externo ou interno (ExternalUse ou InternalUse) que já é seu dependente.***

Semântica: Dado que existe uma instância da metaclasse *WorkProductUseRelationship 1* com o seguintes valores de atributos: *relationType* igual a *dependency*; *source* associado com a instância da metaclasse *InternalUse A*; e *target* associado com as instâncias da metaclasse *ExternalUse B* e *C*, não pode existir outra instância da metaclasse *WorkProductUseRelationship* com os seguintes valores de atributos:

relationType igual a *dependency*; *source* associado com a instância da metaclasse *ExternalUse B* ou *C*; e *target* associado com a instância da metaclasse *InternalUse A*.

Regra #49 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode representar o "todo" e a "parte" em um relacionamento de composição.

Semântica*:* Uma instância da metaclasse *WorkProductUseRelationship* que possui o valor "*composition*" para o atributo *relationType* não pode possuir a mesma instância da metaclasse *ExternalUse* ou *InternalUse* associada nos seus atributos *source* e *target*.

Regra #49 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode representar o "todo" e a "parte" em um relacionamento de agregação.

Semântica: Uma instância da metaclasse *WorkProductUseRelationship* que possui o valor "*agregation*" para o atributo *relationType* não pode possuir a mesma instância da metaclasse *ExternalUse* ou *InternalUse* associada nos seus atributos *source* e *target*.

Regra #49 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode depender de si próprio.

Semântica: Uma instância da metaclasse *WorkProductUseRelationship* que possui o valor "*dependency*" para o atributo *relationType* não pode possuir a mesma instância da metaclasse *ExternalUse* ou *InternalUse* associada nos seus atributos *source* e *target*.

Regra #7 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode ser "parte" em mais de um relacionamento de composição.

Semântica: Não podem existir duas ou mais instâncias da metaclasse *WorkProductUseRelationship* onde o valor do atributo *relationType* é igual a "*composition*" e o valor do atributo *target* dessas instâncias possuem a mesma instância da metaclasse *ExternalUse* ou da metaclasse *InternalUse*.

Dando sequência à definição das regras de boa-formação ainda relacionadas com os produtos de trabalho (internos e externos) e seus relacionamentos, definiram-se duas novas regras que estabelecem, respectivamente, que toda instância da metaclasse *ProcessParameter* deverá estar associada a um produto de trabalho do tipo interno ou externo e que todo produto de trabalho do tipo interno deverá possuir uma associação com a metaclasse *ProcessResponsabilityAssignment*. Essa última regra torna-se necessária para que todo produto de trabalho interno, o qual é obrigatoriamente produzido em um processo de software, esteja relacionado com ao menos um papel responsável por sua produção.

Nesse sentido, faz-se necessário considerar que ambas as regras citadas acima, as quais serão apresentadas a seguir, já estão expressas no metamodelo (ver Figura 26) com as alterações realizadas sobre a multiplicidade das relações entre os elementos *WorkProductUse* e *ProcessParameter* e, também, entre os elementos *ExternalUse* e *ProcessResponsabilityAssignment*.

Regra #4 – Uma parâmetro de entrada e/ou saída (ProcessParameter) de uma atividade deve estar sempre associado a exatamente um produto de trabalho do tipo interno ou externo (InternalUse ou ExternalUse).

Semântica: Toda instância da metaclasse *ProcessParameter* deve possuir uma instância da metaclasse *ExternalUse* ou da metaclasse *InternalUse* associada a seu atributo *parameterType*.

Regra #8 – Um produto de trabalho do tipo interno (InternalUse) deve possuir ao menos um relacionamento com a metaclasse ProcessResponsabilityAssignment.

Semântica: Toda instância da metaclasse *InternalUse* deve estar associada ao atributo *linkedWorkProductUse* de pelo menos uma instância da metaclasse *ProcessResponsabilityAssignment*.

Outras regras de boa-formação estabelecidas para o pacote *Process Structure* são relacionadas com os aspectos de duplicidade e opcionalidade e envolvem os elementos de processo *Atividade*, *Produto de Trabalho* e *Papel* e os seus relacionamentos. Essas regras, descritas abaixo, estabelecem, respectivamente, que nenhum elemento de processo ou um de seus relacionamentos devem ser definidos mais de uma vez em um processo de software, assim como que informações de opcionalidade devem ser consistentes entre si. Não é possível definir uma instância de atividade como opcional em um processo, por exemplo, se essa atividade possui instâncias de elementos internos definidos como obrigatórios.

Regra #25 – Uma atividade (Activity) opcional não deve possuir elementos obrigatórios.

Semântica: Toda instância da metaclasse *Activity* que possui o valor “*true*” para o atributo *isOptional* não deve possuir como parte nenhuma instância das metaclasses *InternalUse*, *ExternalUse* e *RoleUse* com o valor “*false*” para seus respectivos atributos *isOptional.*

Regra #29 – Um produto de trabalho do tipo interno (InternalUse) obrigatório deve ser associado a pelo menos um papel (RoleUse) obrigatório.

Semântica: Toda instância da metaclasse *ExternalUse* que possui o valor '*false*' para o atributo *isOptional* deve estar associada ao atributo *linkedWorkProductUse* de pelo menos uma instância da metaclasse *ProcessResponsabilityAssignment,* onde o atributo *linkedRoleUse* dessa instância de *ProcessResponsabilityAssignment* deve ser associado a uma instância da metaclasse de *RoleUse* que possui o atributo *isOptional* igual a '***false'***.

Regra #33 – Não pode existir mais de um papel (RoleUse) com o mesmo nome em um processo de software.

Semântica: Não pode existir mais de uma instância da metaclasse *RoleUse* com o mesmo valor para o atributo *name*.

Regra #34 – Não pode mais de um produto de trabalho do tipo externo (ExternalUse) com o mesmo nome em um processo de software.

Semântica: Não pode existir mais de uma instância da metaclasse *ExtenalUse* com o mesmo valor para o atributo *name*.

Regra #34 – Não pode existir mais de um produto de trabalho do tipo interno (InternalUse) com o mesmo nome em um processo de software.

Semântica: Não pode existir mais de uma instância da metaclasse *InternalUse* com o mesmo valor para o atributo *name*.

Regra #36 – Um papel (RoleUse) não pode ser definido mais de uma vez como responsável pelo mesmo produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse).

Semântica: Não pode existir mais de uma instância da metaclasse *ProcessResponsabilityAssignment* que possui a mesma instância da metaclasse *ExternalUse* ou da metaclasse *InternalUse* selecionada para o atributo *linkedWorkProductUse* e a mesma instância da metaclasse *RoleUse* selecionada para o atributo *linkedRoleUse*.

Regra #37 – Um produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse) não pode ser definido mais de uma vez como dependente do mesmo produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse).

Semântica: Não pode existir mais de uma instância da metaclasse *WorkProductUseRelationship* que possua o valor "*dependency*" para o atributo *relationType* e que possua a mesma instância da metaclasse *ExternalUse* ou da

metaclasse *InternalUse* selecionada para o atributo *source* e a mesma instância da metaclasse *ExternalUse* ou da metaclasse *InternalUse* selecionada para o atributo *target*.

Regra #38 – Um produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse) não pode ser definido mais de uma vez como o todo, através de uma relação de composição, do mesmo produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse).

Semântica: Não pode existir mais de uma instância da metaclasse *WorkProductUseRelationship* que possua o valor "*composition*" para o atributo *relationType* e que possua a mesma instância da metaclasse *ExternalUse* ou da metaclasse *InternalUse* selecionada para o atributo *source* e a mesma instância da metaclasse *ExternalUse* ou da metaclasse *InternalUse* selecionada para o atributo *target*.

Regra #39 – Um produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse) não pode ser definido mais de uma vez como o todo, através de uma relação de agregação, do mesmo produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse).

Semântica: Não pode existir mais de uma instância da metaclasse *WorkProductUseRelationship* que possua o valor "*agregation*" para o atributo *relationType* e que possua a mesma instância da metaclasse *ExternalUse* ou da metaclasse *InternalUse* selecionada para o atributo *source* e a mesma instância da metaclasse *ExternalUse* ou da metaclasse *InternalUse* selecionada para o atributo *target*.

Regras de boa-formação relacionadas com a consistência do sequenciamento de atividades também foram determinadas. Tais regras envolvem as metaclasses *WorkBreakdownElement*, *WorkSequence*, *Activity* e as metaclasses de enumeração *WorkSequenceKind e WorkBreakdownElementKind*.

As primeiras regras de formação para consistência do sequenciamento de atividades são relacionadas com os nodos iniciais e finais de um processo de software e estabelecem: (1) todo processo deve possuir exatamente um nodo inicial e um nodo final; (2) o nodo inicial de um processo de software não deve ser conectado como sucessor para nenhuma atividade desse processo; (3) o nodo final de um processo de software não deve ser conectado como predecessor para nenhuma atividade desse processo; (4) o nodo inicial de um processo de software deve ser o predecessor para pelo menos uma atividade desse processo; e (5) o nodo final de um processo de software deve ser o sucessor para pelo menos uma atividade desse processo.

Regra #12 – Um processo de software deve possuir exatamente um nodo final.

Semântica: Apenas uma instância da metaclasse *Activity* com o valor do atributo *specialNode* igual a "*end*" deverá ser instanciada em um processo de software.

Regra #13 – Um processo de software deve possuir exatamente um nodo inicial.

Semântica: Apenas uma instância da metaclasse *Activity* com o valor do atributo *specialNode* igual a "*start*" deverá ser instanciada em um processo de software.

Regra # 14 – O nodo inicial não pode ser sucessor para nenhuma atividade no sequenciamento definido em um processo de software.

Semântica: Uma instância da metaclasse *Activity* que possui o valor do atributo *specialNode* igual a "*start*" não pode estar associada no atributo *successor* de uma instância da metaclasse *WorkSequence.*

Regra # 15 – O nodo final não pode ser predecessor para nenhuma atividade no sequenciamento definido em um processo de software.

Semântica: Uma instância da metaclasse *Activity* que possui o valor do atributo *specialNode* igual a "*end*" não pode estar associada no atributo *predecessor* de uma instância da metaclasse *WorkSequence.*

Regra # 43 – O nodo inicial deve ser definido como predecessor para pelo menos uma atividade no sequenciamento definido em um processo de software.

Semântica: Toda instância da metaclasse *Activity* que possui o valor do atributo *specialNode* igual a "*start*" deve estar associada no atributo *predecessor* de pelo menos uma instância da metaclasse *WorkSequence.*

Regra # 44 – O nodo final deve ser definido como sucessor para pelo menos uma atividade no sequenciamento definido em um processo de software.

Semântica: Toda instância da metaclasse *Activity* que possui o valor do atributo *specialNode* igual a "*end*" deve estar associada no atributo *successor* de pelo menos uma instância da metaclasse *WorkSequence.*

Seguindo com as regras de boa-formação para consistência do sequenciamento de atividades em um processo de software, apresenta-se a seguir uma regra de boa-formação que respeita uma definição específica desta pesquisa. Tal definição estabelece que todo nodo inicial e final que estiver conectado, respectivamente, como predecessor e sucessor para outras atividades do processo, através de instâncias da metaclasse *WorkSequence* deve ser conectado com o tipo de sequência *finishToStart*. No caso

específico do nodo inicial, isso fará com que todas as conexões que partem desse nodo para outras atividades do processo sejam do tipo *finishToStart*. Já no caso do nodo final, a definição faz com que todas as conexões que chegam de outras atividades para esse nodo sejam do tipo *finishToStart*, ou seja, todas as atividades do processo precisam terminar para a chegada do nodo final desse processo.

Regra # 45 – O sequenciamento estabelecido entre as atividades e o nodo inicial ou final em um processo de software deve ser realizado através do tipo de sequenciamento finishToStart.

Semântica: Toda instância da metaclasse *Activity* que possui o valor do atributo *specialNode* igual a "*start*" ou "*end*" só pode ser associada com instâncias da metaclasse *WorkSequence* que possuem valor do atributo *linkKind* igual a "*finishToStart*".

Outras regras de boa-formação necessárias são as regras que estabelecem que os nodos inicial e final não podem ter relação com outros elementos de processo que não sejam especificamente relações de sequenciamento com outras atividades. O motivo para essa definição e criação de regras é devido ao fato das atividades que representam os nodos inicial e final de um processo serem consideradas nesta pesquisa como tipos de atividades especiais.

Nesse contexto, as regras de boa-formação a seguir estabelecem: (1) as atividades que representam os nodos inicial e final não devem estar associadas a papéis através de instâncias da metaclasse *ProcessPerformer*; e (2) as atividades que representam os nodos inicial e final de um processo não devem possuir elementos.

Regra #41 – As atividades do processo que representam os nodos inicial e final não devem ser associados com instâncias da metaclasse ProcessPerformer.

Semântica: Toda instância da metaclasse *Activity* que possui o valor do atributo *specialNode* igual a "*start*" ou "*end*" não pode estar associada no atributo *linkedTaskUse* de nenhuma instância da metaclasse *ProcessPerformer*.

Regra #42 – As atividades do processo que representam os nodos inicial e final não devem possuir elementos.

Semântica: Nenhuma instância da metaclasse *Activity* que possui o valor do atributo *specialNode* igual a "*start*" ou "*end*" não pode possuir outras instâncias através de um relacionamento de composição com metaclasses que mantém relacionamento de herança com a metaclasse *BreakdownElement*.

Por fim, as últimas regras de boa-formação relacionadas com sequenciamento das atividades são as que estabelecem as seguintes definições para um processo de software: (1) o sequenciamento das atividades não deve apresentar situações que representem *deadlocks* na execução de um processo de software; (2) para um sequenciamento consistente, é necessário garantir que todas as atividades são iniciadas após o nodo inicial e são finalizadas antes do nodo final; e (3) não devem existir duas ou mais sequências iguais entre duas atividades.

Antes de apresentar a descrição e semântica das regras citadas acima, faz-se necessário entender como elas foram definidas nesta pesquisa. Isso porque, embora tais regras sejam básicas para um processo de software, suas definições não foram triviais no contexto deste estudo, considerando a solução de sequenciamento proposta pelo metamodelo original SPEM original 2.0. As principais dificuldades encontradas foram relacionadas com os diferentes valores (*finishToStart*, *startToStart*, *finishToFinish* e *startToFinish*) que uma transição entre duas atividades podem assumir.

Desse modo, primeiramente, a partir da consideração da regra de boa-formação que estabelece a ausência de *deadlocks* no sequenciamento de atividades*,* constatou-se a primeira dificuldade desta pesquisa em estabelecer as regras de sequenciamento para um processo de software. Isso se deu, pois, analisando o sequenciamento de atividades estabelecido com a solução proposta pelo metamodelo original SPEM 2.0, não foi possível identificar todas as situações que configuram um *deadlock* na execução de um processo. Para facilitar o entendimento, como exemplo, a Figura 27 mostra um sequenciamento no qual não é possível identificar um ou mais *deadlocks*.

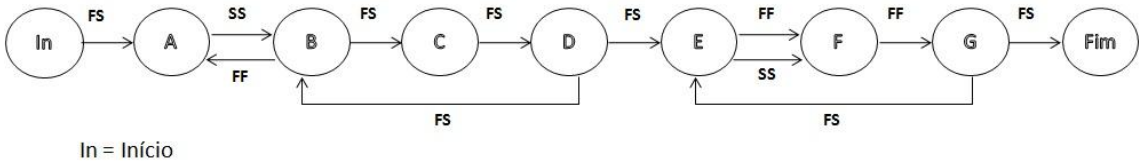

Figura 27 – Sequenciamento definido para atividades do metamodelo SPEM 2.0

Na Figura 27, existem três pontos do sequenciamento que formam ciclos e podem formar um ou mais *deadlocks* no processo de software. Tais pontos estão entre as atividades *A* e *B*; *B, C* e *D*; e *E*, *F* e *G*. Analisando a primeira situação, que envolve apenas duas atividades (*A* e *B*), é fácil verificar que o sequenciamento estabelecido é válido e não configura um *deadlock*. Diz-se isso, porque os sequenciamentos definidos entre as atividades *A* e *B* estabelecem que a atividade *B* deve começar depois do início da atividade *A* e terminar antes do fim da atividade *A*. Em outras palavras, em uma linha

de tempo, esse sequenciamento define que a execução da atividade *B* ocorre sempre dentro do tempo da execução da atividade *A*.

Para facilitar a compreensão do que foi dito, a Figura 28 mostra um gráfico, contendo uma linha de tempo, no eixo *X,* e as atividades *A* e *B,* no eixo *Y*. Nessa Figura, é mostrada, em verde, a execução da atividade *A* e *B* respeitando a transição SS, e, em vermelho, a execução da atividade *A* e B respeitando a transição FF. Já a parte em azul da figura mostra uma situação válida para execução das atividade *A* e *B,* respeitando ambas as transições SS e FF em que a atividade *B* possui seu início e fim dentro do tempo de execução da atividade *A*.

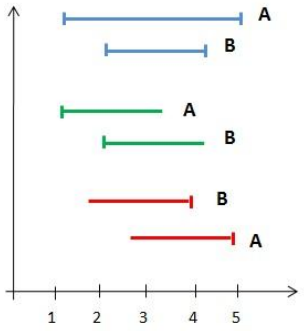

Figura 28 – Gráfico que demonstra a execução da transição *A* SS *B* e da transição *B* FF *A*

O número de situações possíveis para formação de ciclos envolvendo o sequenciamento de duas atividades é dezesseis. Esse número é resultado de todas as combinações possíveis com as transições (*finishToStart*, *startToStart*, *finishToFinish* e *startToFinish*) propostas pelo metamodelo original SPEM 2.0. Na montagem dos gráficos para as dezesseis combinações, como o mostrado na Figura 28, encontrou-se como resultado a formação de *deadlocks* em sete situações. Todos os resultados podem ser vistos na Tabela 12.

Tabela 12 – Combinações para as transições possíveis entre duas atividades

			Colunas			
			1	**2**	**3**	**4**	**5**	**6**
					A			
					SS	**SF**	**FF**	**FS**
Linhas	**1**							
	2							
	3	**B**	**SS**		*deadlock*	-	-	*deadlock*
	4		**SF**		-	-	-	*deadlock*
	5		**FF**		-	-	-	*deadlock*
	6		**FS**		*deadlock*	*deadlock*	*deadlock*	*deadlock*

Para demonstrar como a Tabela 12 foi montada, a Figura 29 mostra o exemplo da combinação de transições da coluna 5 e da linha 6 dessa tabela. Exatamente nesta célula (coluna 5 e linha 6) o resultado indicado é um *deadlock* para as combinações de

transições. Interpretando as informações dos cabeçalhos da coluna 5 e da linha 6, obtém-se como resultado o sequencimento mostrado no lado esquerdo na Figura 29. O lado direito desta Figura mostra o gráfico que contém uma linha de tempo no eixo *X* e as atividades *A* e *B* no eixo *Y*.

Ainda na Figura 29, em verde, a execução das atividades *A* e *B,* respeitando a transição FF, é mostrada e, em vermelho, apresenta-se a execução das atividades *A* e *B,* respeitando a transição FS. Nota-se, também na mesma figura, que nenhuma execução que respeita ambas as transições FF e FS foi mostrada. O fato para que isto tenha ocorrido é que a combinação das transições mostradas nessa imagem forma um *deadlock* entre as atividades *A* e *B*. O *deadlock* ocorre porque o fim da atividade *A* precisa acontecer para que *B* termine e o fim da atividade *B* precisa acontecer para que *A* inicie.

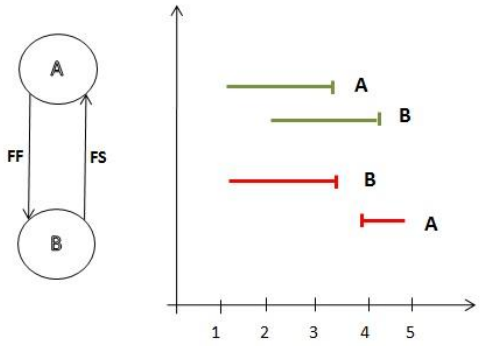

Figura 29 – Gráfico que demonstra a execução da transição *A* FF *B* e da transição *B* FS *A*

Uma vez que o primeiro ciclo da Figura 27 foi devidamente analisado e que, neste ponto desta pesquisa, foram identificados todos os ciclos entre duas atividades que formam *deadlock,* parte-se para análise dos ciclos entre as atividades *B, C* e *D* e entre as atividades *E*, *F* e *G.* A primeira forma utilizada para verificar a formação de *deadlocks* em ciclos que envolvem mais de duas atividades foi a tentativa de simplificação de qualquer ciclo para um ciclo de duas atividades. Para fazer tal simplificação, utilizaram-se as informações de transitividade entre as relações de sequenciamento.

Assim, considerando, por exemplo, o primeiro ciclo da Figura 27 que envolve três atividades (atividades *B, C* e *D*), têm-se as seguintes transições: *B finishToStart C*; *C finishToStart D*; e *D finishToStart B*. Utilizando as informações sobre transitividade entre as relações de sequenciamento, é possível dizer que se o fim da atividade *B* precisa acontecer para que *C* inicie e o fim da atividade *C* precisa acontecer para que *D* inicie, logo, diz-se que o fim da atividade *B* precisa acontecer para que *D* inicie. Dessa maneira, é possível incluir uma nova transição entre as atividades *B* e *D* do tipo *finishToStart*, conforme ilustrado na Figura 30.

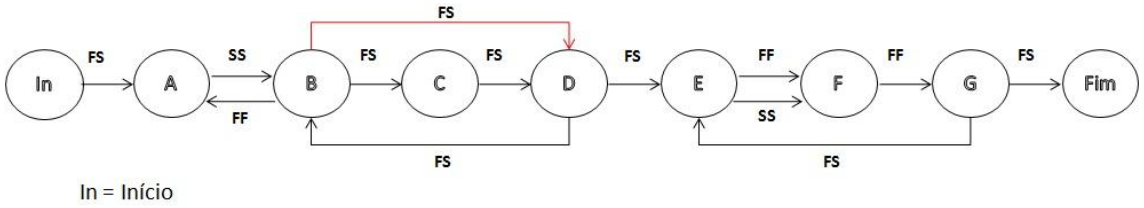

Figura 30 – Sequenciamento definido para atividades do metamodelo SPEM 2.0 com inclusão de nova transição obtida através de informações sobre transitividade das relações

A nova transição, que é destacada em vermelho na Figura 30, forma agora, um ciclo apenas entre as atividades *B* e *D* com ambas as transições entre os estes elementos do tipo *finishToStart*. Consultando a Tabela 12, já foi explicada acima, torna-se possível identificar a formação de um *deadlock* entre as atividades *B* e *D*, o que, por consequência, indica um *deadlock* no ciclo entre as atividades *B*, *C* e *D* mostrado na Figura 30.

Prosseguindo com a avaliação sobre os ciclos entre as atividades da Figura 27, analisaram-se as transições entre as atividades *E, F* e *G* para a tentativa de simplificação desse ciclo para apenas duas atividades. Uma vez que existem duas transições (*finishToFinish* e *startToStart*) entre as atividades *E* e *F,* foram realizadas duas análises. Assim, as primeiras transições analisadas foram as seguintes: *E finishToFinish F*; *F finishToFinish G*. Utilizando, mais uma vez, as informações sobre transitividade, é possível concluir que se o fim da atividade *E* precisa acontecer para que *F* termine e o fim da atividade *F* precisa acontecer para que *G* termine, logo, o fim da atividade *E* precisa acontecer para que *G* termine.

A partir dessa conclusão, pode-se incluir uma nova transição entre as atividades *E* e *G* do tipo *finishToFinish*. Continuando com as tentativas de simplificação, analisaram-se as seguintes transições para o mesmo ciclo: *E startToStart F*; *F finishToFinish G.* Para estas, não se encontrou nenhum tipo de conclusão relacionada com as atividades *E* e *G,* quando consideradas as informações de transitividade entre as relações de sequenciamento. Dessa forma, não foi possível a simplificação do ciclo entre as atividades *E, F* e *G* para um ciclo que contivesse apenas duas atividades, o que impossibilitou, até este momento da pesquisa, saber se esse ciclo representa ou não um *deadlock* no sequenciamento estabelecido na Figura 27.

Baseado no contexto acima, a tentativa de simplificar os ciclos, através das informações de transitividade para a descoberta de *deadlocks,* foi descartada. A nova solução adotada foi a interpretação das transições (*finishToStart*, *startToStart*, *finishToFinish* e *startToFinish*) propostas pelo metamodelo original SPEM 2.0, utilizando-

se dos conceitos *atividade_início* e *atividade_fim,* os quais foram propostos por esta pesquisa.

Com isso, pretende-se que todas as atividades envolvidas em um sequenciamento sejam divididas em *atividade_início* e *atividade_fim* e que seja incluída para estas atividades uma transição que parte da *atividade_início* e chega à *atividade_fim*, indicando sempre que o início de uma atividade precede o seu fim. Após esses passos, qualquer uma das transições propostas pelo metamodelo original SPEM 2.0 pode ser redefinida, utilizando os inícios e fins de cada atividade. A Figura 31, mostra como um exemplo da solução proposta, a transformação de um sequenciamento realizado entre atividades com as transições indicadas pelo metamodelo original SPEM 2.0 para o sequenciamento que utiliza os novos conceitos de *atividade_início* e *atividade_fim*.

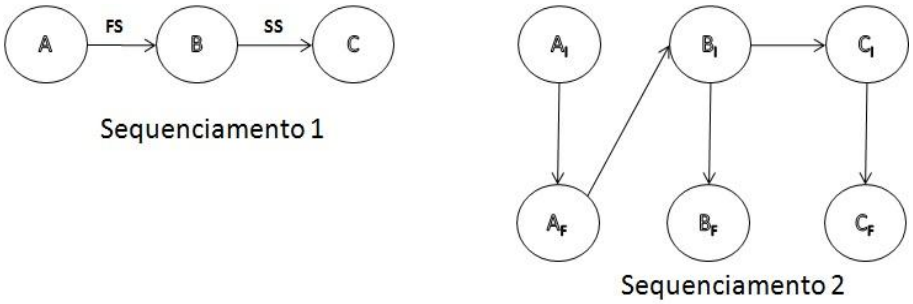

Figura 31 – Sequenciamentos definidos para atividades utilizando-se, respectivamente das transições da metaclasse *WorkSequenceKind* e dos conceitos *Atividade_Início* e *Atividade_fim*

Observa-se na Figura 31 que ambos os sequenciamentos 1 e 2 representam grafos direcionados[4]. Contudo, o primeiro grafo (representado pelo sequenciamento 1) possui informações nas arestas, o que, como já mencionado acima, dificulta a identificação de *deadlocks* em um processo de software. Dessa forma, nesta pesquisa, assume-se a utilização de grafos direcionados como o mostrado na parte direita da Figura 31 (representado pelo sequenciamento 2).

Além disso, outra definição estabelecida por esta pesquisa é que, para a obtenção de um sequenciamento consistente, o grafo resultante (após a transformação para um grafo com *atividades_início* e*atividades_fim*) deve ser acíclico e não possuir

[4] Um grafo é comumente representado como um par ordenado (V,A), em que V é o conjunto dos vértices do grafo, e A o conjunto das arestas. Os grafos direcionados são conhecidos como digrafos, grafos dirigidos ou grafos orientados e são tidos como grafos onde as arestas A são ordenadas (direcionadas), ou seja, a aresta (V1, V2) é diferente da aresta (V2, V1) [WES96].

laços. Isso porque, em um sequenciamento, tanto os ciclos quanto os laços representarão *deadlocks* na execução de um processo de software.

Para demonstrar a aplicabilidade da solução proposta para o sequenciamento de atividades e também complementar a compreensão sobre isso, todos os ciclos mostrados na Figura 27 foram novamente analisados. Para essa realização, inicialmente, o grafo mostrado nesta Figura foi transformado em um grafo que possui *atividades_início* e *atividades_fim*, conforme pode ser visto na Figura 32.

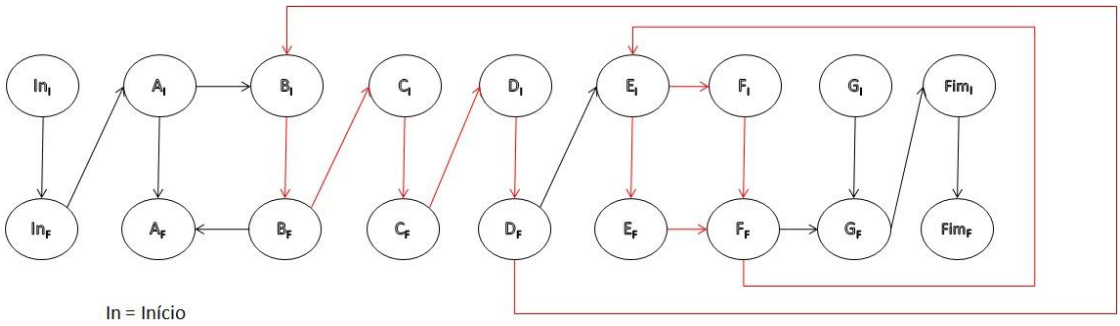

Figura 32 – Sequenciamento definido para atividades utilizando-se dos conceitos *Atividade_Início* e *Atividade_fim*

Nota-se, na Figura 32, que algumas arestas, as quais formam ciclos no grafo, foram destacadas em vermelho. Isso ocorreu, porque, como mencionado acima, a formação de ciclos representa inconsistências (nesse caso, *deadlocks*) em um sequenciamento que se utiliza da solução proposta nesta pesquisa.

Toda explicação anterior foi realizada com o objetivo de encontrar inconsistências relacionadas com os *deadlocks* de um processo de software. Contudo, a nova solução proposta para o sequenciamento de atividades, também auxilia na identificação de outras inconsistências tais como a garantia de que todas as atividades são iniciadas após o nodo inicial e finalizadas antes do nodo final. Embora esta possa parecer uma situação trivial, dificuldades foram encontradas, quando considerada somente a solução de sequenciamento proposta pelo metamodelo original SPEM 2.0. Um exemplo prático dessas dificuldades é ilustrado na Figura 33 que mostra um pequeno conjunto de atividades sequenciadas.

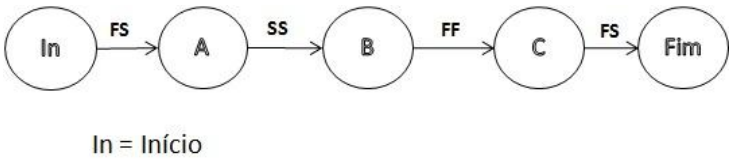

Figura 33 – Pequeno sequenciamento definido para atividades utilizando-se das transições da metaclasse *WorkSequenceKind*

Na Figura 33, observa-se que todas as atividades estão sequenciadas e, assim, conectadas com o nodo inicial e final do sequenciamento. Contudo, se considerarmos as informações das transições que se encontram nas arestas, é possível constatar que as atividades *A* e *C* não possuem, respectivamente, ligações com o nodo final e inicial, o que gera inconsistências no sequenciamento proposto.

Também analisando, especificamente, as transições da atividade *A,* nota-se que ela inicia após o término do nodo inicial e possui uma transição para a atividade *B,* a qual estabelece que, após seu início, a atividade *B* pode ser iniciada. Contudo, nenhuma transição é estabelecida considerando o fim da atividade *A*, o que torna o término desta atividade desconhecido. Embora no exemplo apresentado tenha sido fácil constatar que duas atividades possuíam problemas relacionados com suas ligações com nodos inicial e final, dificuldades existem na medida em que o sequenciamento apresentado envolve um conjunto de atividades maior, com mais transições. Dessa forma, mais uma vez, a solução proposta foi utilizada visando facilitar a identificação de novas inconsistências no sequenciamento de atividades.

A Figura 34 mostra o sequenciamento da Figura 33 transformado em um sequenciamento com *atividades_início* e *atividades_fim*. Desse modo, observa-se que as atividades *A* e *C* realmente não possuem, respectivamente, ligações com os nodos final e inicial. Contudo, a diferença da Figura 33 para a Figura 34 é que, nesta última, não é necessário interpretar nenhum tipo de informação de transição para identificação de erros relacionados com os nodos inicial e final. Nesse tipo de sequenciamento, todas as atividades que não possuem alguma aresta chegando às suas *atividades_início* são inconsistentes, pois não possuem nenhum tipo de ligação com o nodo inicial. Já as *atividades_fim* que não possuem nenhuma aresta partindo para outras atividades em um sequenciamento são também inconsistentes, uma vez que não possuem nenhum tipo de ligação com o nodo final.

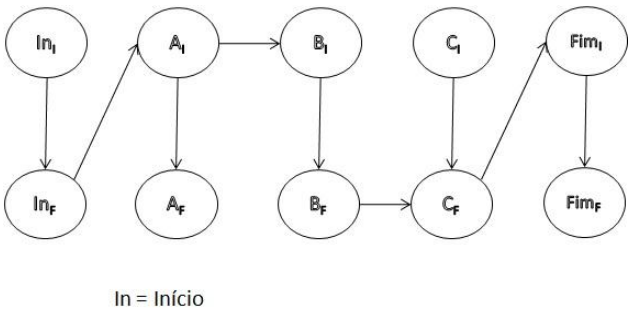

Figura 34 – Transformação do sequenciamento da Figura 34 para os conceitos *Atividade_Início* e *Atividade_fim*

Depois de explicar o funcionamento da solução proposta para o sequenciamento de atividades, as últimas regras de boa-formação que estabelecem as condições para a definição de um sequenciamento de atividades consistente serão apresentadas. Como já mencionado, elas são relacionadas com a formação de *deadlocks*, garantia de ligação de todas as atividades com os nodos inicial e final e ausência de transições duplicadas em um sequenciamento. Tais regras de boa-formação são:

Regra #16 – O sequenciamento de atividades não deve possuir situações que representem um deadlock na execução de um processo de software.

Semântica: Após a divisão das atividades em *atividade_início* e *atividade_fim* e a simplificação das informações sobre as transições não devem existir em nenhum ponto do sequenciamento resultante ciclos e laços.

Regra #17 – Não pode existir mais de uma transição igual entre duas atividades em um sequenciamento.

Semântica: Não pode existir duas ou mais instâncias da metaclasse *WorkSequence* que possuam o mesmo valor para o atributo *linkKind* e que possuam a mesma instância da metaclasse *Activity* selecionada para o atributo *predecessor,* bem como a mesma instância da metaclasse *Activity* selecionada para o atributo *successor*.

Regra #18 – Todas as atividades que são sequenciadas em um processo de software deverão ter ligação com o nodo inicial e final do processo para garantir que toda atividade é iniciada após o nodo inicial e possui seu término antes do nodo final .

Semântica: Após a divisão das atividades em *atividade_início* e *atividade_fim* e a simplificação das informações sobre as transições, toda *atividade_início* deve possuir pelo menos uma aresta chegando nessa atividade e toda *atividade_fim* deve possuir pelo menos uma aresta partindo para outra atividade.

Embora todo conjunto de regras de boa-formação relacionadas com o sequenciamento de atividades tenha sido apresentado acima, um fato importante, o qual não foi explicado até o momento, diz respeito a como o sequenciamento funciona quando uma ou mais atividades possuem subatividades. Esse é o caso do exemplo mostrado na Figura 35, no qual as atividades superiores da estrutura mostrada são sequenciadas e possuem subatividades que, por sua vez, também são divididas em novas subatividades.

Analisando a Figura 35, observa-se que todas as atividades estão sequenciadas. Isso porque o sequenciamento realizado nas atividades superiores é passado por transitividade para suas atividades inferiores. Interpretando o sequenciamento, tem-se que a atividade *A* deve ser executada antes da atividade *B*. Como a atividade *A* é composta por pelas atividades *A1* e *A2* que, por sua vez, são compostas, respectivamente, pelas atividades *A1.1*, *A1.2* e *A2.1* e *A2.2,* diz-se que todo esse conjunto de atividades deve ter sua execução terminada para que qualquer subatividade de *B* possa ser iniciada.

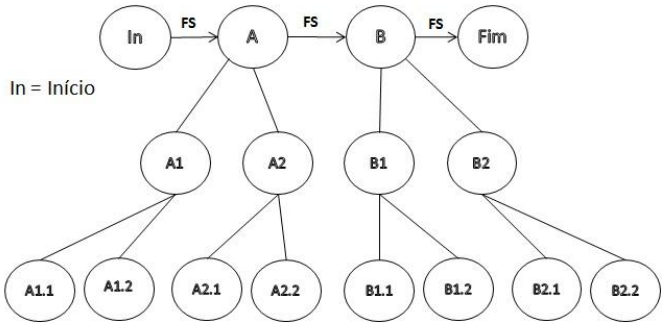

Figura 35 – Sequenciamento definido para atividades que possuem subatividades

No sequenciamento definido na Figura 35, novas sequencias podem ser incluídas entre as subatividades de uma atividade. Esse é o caso do exemplo mostrado na Figura 36 que possui uma nova sequencia para as atividades *A1.1* e *A1.2,* a qual estabelece que a atividade *A1.1* deve ser iniciada antes do início da atividade *A1.2*.

A única restrição estabelecida nesta pesquisa para o sequenciamento feito com atividades que possuem subatividades é que o sequenciamento deve ser feito sempre entre atividades que estão num mesmo nível. Observando o sequenciamento da Figura 36, por exemplo, considerar-se-ia inconsistente uma sequência estabelecida entre as atividades *A* e *B1*. Isso porque essas atividades estão em níveis hierarquicamente diferentes na estrutura proposta.

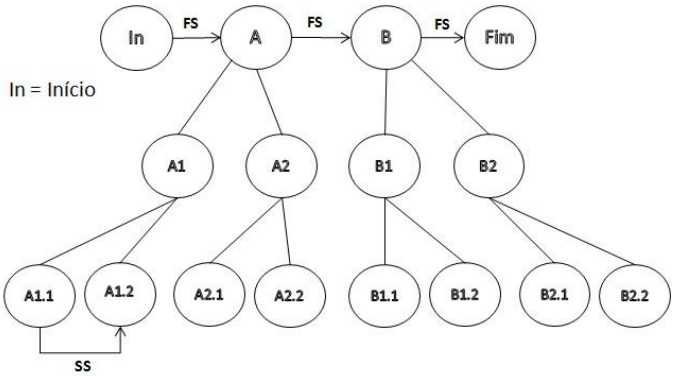

Figura 36 – Sequenciamento definido para atividades e também para suas subatividades

As últimas regras de boa-formação definidas foram para o autorrelacionamento *usedActivity* da metaclasse *Activity* e para o relacionamento *supressedBreakdownElement* que é definido entre as metaclasses *Activity* e *BreakdownElement*. Contudo, como esses relacionamentos são considerados como mecanismos de adaptação de processos do metamodelo original SPEM 2.0, suas regras de boa-formação serão detalhadas na Seção 4.4 deste trabalho.

4.3.2 Pacote *Method Content*

Nesta seção as alterações e regras de boa-formação incluídas no pacote *Method Content* serão apresentadas. Antes disso, a relação das alterações e regras de boa-formação com as premissas definidas para esta pesquisa é mostrada na Tabela 13. Nesta tabela, é possível observar que algumas regras de boa-formação e premissas também não são relacionadas com alterações em metaclasses e relações do metamodelo sSPEM 2.0, assim como no pacote *Process Structure*. Isso ocorre, mais uma vez, devido ao fato de que o atendimento de algumas premissas não exige modificações sobre metaclasses e relações, mas envolve apenas a definição de algumas regras que controlem a instanciação desses elementos em um processo de software.

Tabela 13 – Relação de alterações e regras de boa-formação incluídas no pacote *Method Content*

Alteração	**Regra(s) de Boa-Formação**	**Premissa(s)**
Inclusão do atributo *relationType* para metaclasse *WorkProductDefinitionRelationship* e da metaclasse de enumeração *RelationType*.	Regra #5, Regra #7, Regra #49	P #4 P #5
Redefinição do relacionamento entre as metaclasses *WorkProductDefinition* e *Default_TaskDefinitionParameter*.	Regra #4	P #7
-	Regra #11	P #8
-	Regra #33, Regra #34, Regra #35, Regra #36, Regra #37, Regra #38, Regra #39, Regra #40	P #21 P #22

4.3.2.1 Alterações sobre Metaclasses e Relações do Pacote Method Content

As mudanças realizadas no pacote *Method Content* foram feitas apenas sobre as metaclasses *WorkProductDefinitionRelationship, WorkProductDefinition* e *Default_TaskDefinitionParameter* e têm o mesmo significado de alterações já realizadas sobre as metaclasses *WorkProductUseRelationship, WorkProductUse* e

ProcessParameter do pacote *Process Structure*. Tais alterações são: a inclusão do atributo *relationType* para a metaclasse *WorkProductDefinitionRelationship;* a inclusão da metaclasse do tipo enumeração chamada *WorkProductRelationshipKind* que define os valores válidos (*dependência*, *composição* e *agregação*) para o atributo *relationType;* e a alteração da multiplicidade no relacionamento entre as metaclasses *WorkProductDefinition* e *Default_TaskDefinitionParameter* de 0..* para exatamente 1, estabelecendo que toda instância de *Default_TaskDefinitionParameter* deve estar associada a um produto de trabalho. A Figura 37 apresenta as principais metaclasses e relacionamentos do pacote *Method Content* e destaca com as Letras A, B e C as modificações citadas acima.

4.3.2.2 Regras de Boa-Formação do Pacote Method Content

O subconjunto de regras de boa-formação incluídas no pacote *Method Content* faz parte do conjunto de regras incluídas nos pacotes *Process Structure* e *Process with Methods*. Isso ocorre, pois os elementos e relacionamentos definidos no pacote *Method Content* têm basicamente os mesmos significados de elementos definidos em outros pacotes do metamodelo sSPEM 2.0. Dessa forma, é importante considerar que, embora as regras apresentadas nesta seção sejam específicas para instâncias de metaclasses definidas no pacote *Method Content,* elas recebem os mesmos identificadores de regras dos pacotes *Process Structure* e *Process with Methods*. A única alteração que essas regras possuem é a adequação das metaclasses que devem ser utilizadasem cada regra.

A primeira regra de boa-formação incluída no pacote *Method Content* é expressa através da alteração realizada sobre a multiplicidade da relação entre as classes *Default_TaskDefinitionParameter* e *WorkProductDefinition* (ver Figura 37). Essa regra, incluída no pacote *Process Structure* (conforme Seção 4.3.1.2), é apresentada abaixo estabelece que toda instância da metaclasse *Default_TaskDefinitionParameter* deverá estar associada a exatamente um produto de trabalho.

Regra #4 – Um parâmetro de entrada e/ou saída (Default_TaskDefinitionParameter) de uma tarefa deve estar sempre associado a exatamente um produto de trabalho (WorkProductDefinition).

Semântica: Toda instância da metaclasse *Default_TaskDefinitionParameter* deve possuir uma instância da metaclasse *WorkProductDefinition* associada a seu atributo *parameterType*.

Outras regras de boa-formação incluídas são as regras Regra #5, #49 e #7, as quais foram definidas no pacote *Process Structure* (conforme Seção 4.3.1.2) e são específicas sobre os relacionamentos entre produtos de trabalho.

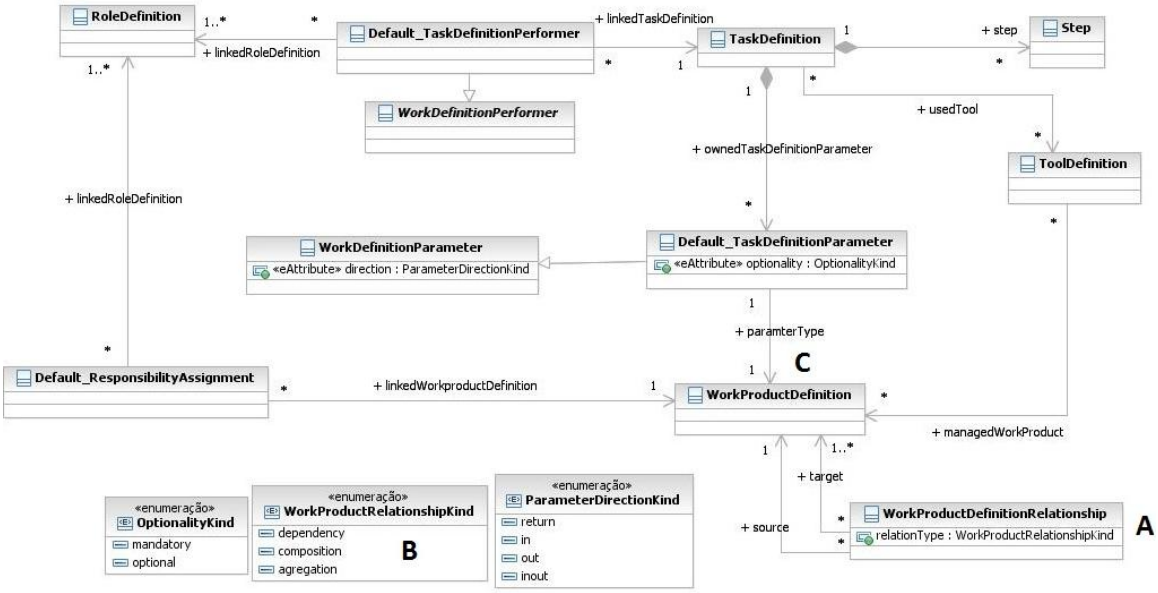

Figura 37 – Metaclasses e relações incluídas e/ou alteradas no pacote *Method Content*

Regra #5 - Um produto de trabalho (WorkProductDefinition) não pode ser o "todo" em um relacionamento de composição se uma de suas "partes" já representa o seu "todo" em outro relacionamento de composição ou representa o seu "todo" pela transitividade da relação de composição.

Semântica*:* Dado que existe uma instância da metaclasse *WorkProductDefinitionRelationship 1* com o seguintes valores de atributos: *relationType* igual a *composition*; *source* associado com a instância da metaclasse *WorkProductDefinition A*; e *target* associado com as instâncias da metaclasse *WorkProductDefinition B* e *C*, não pode existir outra instância da metaclasse *WorkProductDefinitionRelationship* com os seguintes valores de atributos: *relationType* igual a *composition*; *source* associado com a instância da metaclasse *WorkProductDefinition B* ou *C*; e *target* associado com a instância da metaclasse *WorkProductDefinition A*.

Regra #5 -* Um *produto de trabalho (WorkProductDefinition) não pode ser o "todo" em um relacionamento de agregação se uma de suas "partes" já representa o seu "todo" em outro relacionamento de agregação ou representa o seu "todo" pela transitividade da relação de agregação.

Semântica*:* Dado que existe uma instância da metaclasse *WorkProductDefinitionRelationship 1* com o seguintes valores de atributos: *relationType* igual a *agreggation*; *source* associado com a instância da metaclasse *WorkProductDefinition A*; e *target* associado com as instâncias da metaclasse *WorkProductDefinition B* e *C*, não pode existir outra instância da metaclasse *WorkProductDefinitionRelationship* com os seguintes valores de atributos: *relationType* igual a *agreggation*; *source* associado com a instância da metaclasse *WorkProductDefinition B* ou *C*; e *target* associado com a instância da metaclasse *WorkProductDefinition A*.

Regra #5 - U***m produto de trabalho (WorkProductDefinition) não pode depender de um produto de trabalho (WorkProductDefinition) que já é seu dependente.***

Semântica*:* Dado que existe uma instância da metaclasse *WorkProductDefinitionRelationship 1* com o seguintes valores de atributos: *relationType* igual a *dependency*; *source* associado com a instância da metaclasse *WorkProductDefinition A*; e *target* associado com as instâncias da metaclasse *WorkProductDefinition B* e *C*, não pode existir outra instância da metaclasse *WorkProductDefinitionRelationship* com os seguintes valores de atributos: *relationType* igual a *dependency*; *source* associado com a instância da metaclasse *WorkProductDefinition B* ou *C*; e *target* associado com a instância da metaclasse *WorkProductDefinition A*.

Regra #49 - Um produto de trabalho (WorkProductDefinition) não pode representar o "todo" e a "parte" em um relacionamento de composição.

Semântica*:* Uma instância da metaclasse *WorkProductDefinitionRelationship* que possui o valor "*composition*" para o atributo *relationType* não pode possuir a mesma instância da metaclasse *WorkProductDefinition* associada nos seus atributos *source* e *target*.

Regra #49 - Um produto de trabalho (WorkProductDefinition) não pode representar o "todo" e a "parte" em um relacionamento de agregação.

Semântica*:* Uma instância da metaclasse *WorkProductDefinitionRelationship* que possui o valor "*agregation*" para o atributo *relationType* não pode possuir a mesma instância da metaclasse *WorkProductDefinition* associada nos seus atributos *source* e *target*.

Regra #49 - Um produto de trabalho (WorkProductDefinition) não pode depender de si próprio.

Semântica*:* Uma instância da metaclasse *WorkProductDefinitionRelationship* que possui o valor "*dependency*" para o atributo *relationType* não pode possuir a mesma instância da metaclasse *WorkProductDefinition* associada nos seus atributos *source* e *target*.

Regra #7 - Um produto produto de trabalho (WorkProductDefinition) não pode ser "parte" em mais de um relacionamento de composição.

Semântica*:* Não podem existir duas ou mais instâncias da metaclasse *WorkProductDefinitionRelationship* onde o valor do atributo *relationType* é "*composition*" e o valor do atributo *target* destas instâncias possuem a mesma instância da metaclasse *WorkProductDefinition*.

Ainda relacionado com as relações entre produtos de trabalho, incluiu-se uma nova regra de boa-formação. Tal regra envolve os elementos tarefa e produto de trabalho e foi definida, inicialmente, no pacote *Process with Methods* (ver Seção 4.3.3.2).

Regra #11 – Todas as dependências de um produto de trabalho (WorkProductDefinition) devem estar conectadas como entrada nas suas tarefas (TaskDefinition) de produção.

Semântica: Dado que existe uma instância da metaclasse *WorkProductDefinitionRelationship 1* com o seguintes valores de atributos: *relationType* igual a *dependency*; *source* associado com a instância da metaclasse *InternalUse A*; e *target* associado com as instâncias da metaclasse *ExternalUse B* e *C*. E, que a instância da metaclasse *WorkProductDefinition A* está associada a uma instância da metaclasse *TaskDefinition T1* através de uma instância da metaclasse *Default_TaskDefinitionParameter* com o valor do atributo *direction* igual a "*out*". Nesse caso, as instâncias da metaclasse *WorkProductDefinition B* e *C* deverão também estar associadas a instância da metaclasse *TaskDefinition T1* através de instâncias da metaclasse *Default_TaskDefinitionParameter* com o valor do atributo *direction* igual a "*in*".

As últimas regras de boa-formação incluídas são as regras relacionadas com o aspecto de duplicidade, as quais foram definidas tanto no pacote *Process Structure* (conforme Seção 4.3.1.2) quanto no pacote *Process with Methods* (conforme Seção 4.3.3.2).

Regra #33 – Não pode existir mais de um papel (RoleDefinition) com o mesmo nome no repositório (Method Content).

Semântica Não pode existir mais de uma instância da metaclasse *RoleDefinition* com o mesmo valor para o atributo *name*.

Regra #34 – Não pode existir mais de um produto de trabalho (WorkProductDefinition) com o mesmo nome no repositório (Method Content).

Semântica: Não pode existir mais de uma instância da metaclasse *WorkProductDefinition* com o mesmo valor para o atributo *name*.

Regra #35 – Um papel (RoleDefinition) não pode ser definido mais de uma vez como executor da mesma tarefa (TaskDefinition).

Semântica: Não pode existir mais de uma instância da metaclasse *Default_TaskDefinitionPerformer* que possua a mesma instância da metaclasse *RoleDefinition* selecionada para o atributo *linkedRoleDefinition* e a mesma instância da metaclasse *TaskDefinition* selecionada para o atributo *linkedTaskDefinition*.

Regra #36 – Um papel (RoleDefinition) não pode ser definido mais de uma vez como responsável pelo mesmo produto de trabalho (WorkProductDefinition).

Semântica: Não pode existir mais de uma instância da metaclasse *Default_ResponsabilityAssignment* que possua a mesma instância da metaclasse *WorkProductDefinition* selecionada para o atributo *linkedWorkProductDefinition* e a mesma instância da metaclasse *RoleDefinition* selecionada para o atributo *linkedRoleDefinition*.

Regra #37 – Um produto de trabalho (WorkProductDefinition) não pode ser definido mais de uma vez como dependente do mesmo produto de trabalho (WorkProductDefinition).

Semântica: Não pode existir mais de uma instância da metaclasse *WorkProductDefinitionRelationship* que possua o valor "*dependency*" para o atributo *relationType* e que possua a mesma instância da metaclasse *WorkProductDefinition* selecionada para o atributo *source* e a mesma instância da metaclasse *WorkProductDefinition* selecionada para o atributo *target*.

Regra #38 – Um produto de trabalho (WorkProductDefinition) não pode ser definido mais de uma vez como o todo, através de uma relação de composição, do mesmo produto de trabalho (WorkProductDefinition).

Semântica: Não pode existir mais de uma instância da metaclasse *WorkProductDefinitionRelationship* que possuam o valor "*composition*" para o atributo

relationType e que possua a mesma instância da metaclasse *WorkProductDefinition* selecionada para o atributo *source* e a mesma instância da metaclasse *WorkProductDefinition* selecionada para o atributo *target*.

Regra #39 – Um produto de trabalho (WorkProductDefinition) não pode ser definido mais de uma vez como o todo, através de uma relação de agregação, do mesmo produto de trabalho (WorkProductDefinition).

Semântica: Não pode existir mais de uma instância da metaclasse *WorkProductDefinitionRelationship* que possua o valor "*agregation*" para o atributo *relationType* e que possua a mesma instância da metaclasse *WorkProductDefinition* selecionada para o atributo *source* e a mesma instância da metaclasse *WorkProductDefinition* selecionada para o atributo *target*.

Regra #40 - Uma tarefa (TaskDefinition) não pode possuir dois ou mais parâmetros de entrada e/ou saída (Default_TaskDefinitionParameter) iguais.

Semântica*:* Nenhuma instância da metaclasse *TaskDefinition* pode possuir mais de uma instância da metaclasse *Default_TaskDefinitionParameter* com os mesmos valores para os atributos *direction* e *parameterType*.

4.3.3 Pacote *Process with Methods*

Esta seção descreve o conjunto de alterações e regras de boa-formação incluídas no pacote *Process with Methods*. A relação de tais alterações e regras de boa-formação com as premissas desta pesquisa estão mostradas na Tabela 14. Observa-se, na referida tabela, que nesse pacote, assim como nos pacotes *Process Structure* e *Method Content*, algumas regras de boa-formação são definidas sem a necessidade de mudanças específicas sobre metaclasses ou relacionamentos do metamodelo.

Nas últimas linhas da Tabela 14, nota-se também que algumas alterações realizadas no pacote *Process with Methods* não estão associadas a nenhuma regra de boa-formação e premissas. Isto ocorre, uma vez que essas alterações foram realizadas com o objetivo exclusivo de prover uma melhor organização para os elementos de um processo de software e de um repositório (*Method Content*), assim como de diferenciar as atividades de definição e adaptação de um processo de software. Maiores detalhes sobre tais alterações serão realizadas na próxima seção deste trabalho.

Tabela 14 – Relação de alterações e regras de boa-formação incluídas no pacote *Process with Methods*

Alteração	**Regra(s) de Boa-Formação**	**Premissa(s)**
-	Regra #24	P #6
Redefinição do relacionamento entre as metaclasses *ProcessPerformer* e *TaskUse*.	Regra #10	P #7
Redefinição do relacionamento entre as metaclasses *TaskUse* e *ProcessParameter*.	Regra #9	P #7
-	Regra #21	P #7
-	Regra #11, Regra #19	P #8
-	Regra #20	P #9
-	Regra #6, Regra #12, Regra #13, Regra #14, Regra #15, Regra #16, Regra #17, Regra #18, Regra#41, Regra #42, Regra #43, Regra #44, Regra #45 e Regra #50	P #10 P #11 P #12
-	Regra #22	P #13
Redefinição do relacionamento entre as metaclasses *ProcessPerformer* e *RoleUse*.	Regra #48	P #14
-	Regra #23, Regra #25, Regra #26, Regra #27, Regra #28, Regra #30, Regra #31, Regra #32	P #15 P #16 P #17 P #18 P #20
-	Regra #35, Regra #40	P #21 P #22
Inclusão das metaclasses *RolePackage, TaskPackage, ToolPackage, WorkProductDefinitionPackage* e *MethodContentRelationshipPackage*	-	-
Inclusão das metaclasses *RoleUsePackage, TaskUsePackage, ActivityPackage, WorkProductUsePackage* e *ProcessRelationshipPackage*	-	-
Redefinição dos relacionamentos de composição da metaclasse *ProcessPackage*	-	-
Redefinição dos relacionamentos de composição da metaclasse *MethodContentPackage*	-	-
Inclusão do atributo *isAuthoring* para a metaclasse ProcessPackage	-	-

4.3.3.1 Alterações sobre Metaclasses e Relações do Pacote Process with Methods

Todas as alterações já descritas nas seções 4.3.1 e 4.3.2 para o metamodelo SPEM 2.0 estão incluídas no pacote *Process with Methods*, uma vez que este pacote *merge* todo conteúdo dos pacotes *Process Structure* e *Method Content* (conforme Seção 2.6.1.4*).* Basicamente, as mudanças feitas especificamente no pacote *Process with*

Methods foram realizadas com três objetivos: (1) a inclusão de algumas metaclasses que organizam os elementos de processo e elementos do repositório (*Method Content*) em pacotes de elementos; (2) a modificação da metaclasse *ProcessPackage* para tornar possível a identificação de quando um processo de software está sendo criado ou adaptado no metamodelo SPEM 2.0; e (3) a modificação sobre alguns relacionamentos existentes entre as metaclasses do *Pacote with Methods* para tornar possível a criação de algumas regras de boa-formação que envolvem o elemento de processo tarefa (definido neste pacote do metamodelo SPEM 2.0).

As primeiras metaclasses incluídas para o pacote *Process with Methods* foram as metaclasses que organizam o conteúdo do repositório (*Method Content*) em pacotes de elementos e relacionamentos. As novas metaclasses são: *TaskPackage*, *WorkProductPackage*, *ToolPackage*, *RolePackage* e *MethodContentRelationshipPackage*. Todas as novas metaclasses incluídas foram associadas através de um relacionamento de composição com a metaclasse já existente no metamodelo SPEM 2.0, denominada *MethodContentPackage,* conforme mostrado na Figura 38.

Analogamente às inclusões descritas acima, outras metaclasses foram incluídas, contudo, agora, com objetivo de organizar os elementos do processo em pacotes de elementos e relacionamentos. As novas metaclasses são: *TaskUsePackage*, *RoleUsePackage*, *WorkProductUsePackage*, *ProcessRelationshipPackage* e *ActivityPackage*. Elas foram associadas através de um relacionamento de composição com a metaclasse já existente no metamodelo SPEM 2.0, denominada *ProcessPackage,* conforme mostrado na Figura 39.

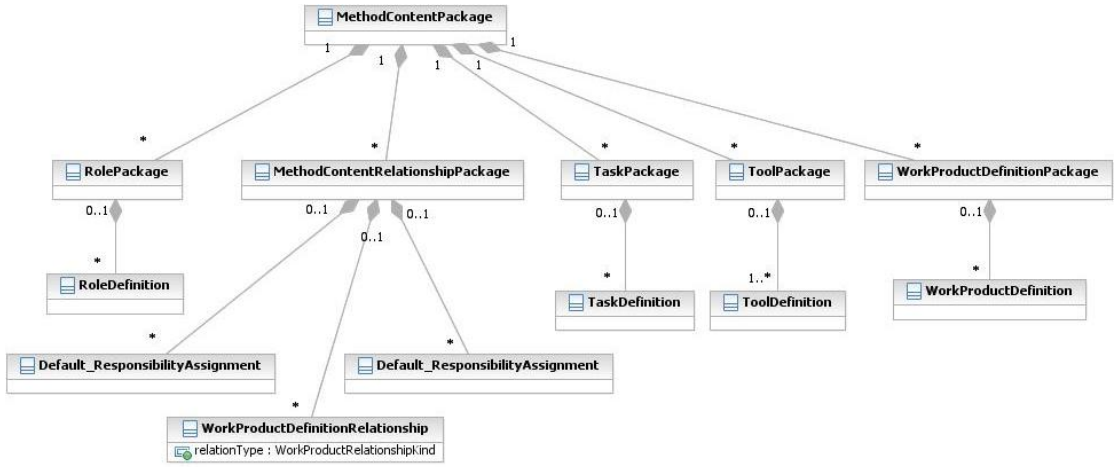

Figura 38 – Inclusão de metaclasses no pacote *Process with Methods* para organizar o conteúdo do repositório (*Method Content*)

Observa-se, portanto, nas Figuras 38 e 39, que relacionamentos de composição também foram incluídos entre as novas metaclasses e algumas metaclasses específicas do metamodelo SPEM 2.0. O objetivo dessas novas relações é permitir que elementos e relacionamentos de processo sejam criados especificamente dentro de apenas um pacote. Considerando, por exemplo, o repositório (*Method Content*), com as novas metaclasses e relações, só será possível criar produtos de trabalho dentro do pacote Produto de Trabalho. Já considerando os processos de software (*Process Package*), o funcionamento é um pouco diferente. Isso porque, embora será possível criar produtos de trabalho dentro do pacote Produto de Trabalho, também será possível criar estes e outros elementos dentro de uma atividade, devido ao fato de a metaclasse *Activity* manter relação de composição com praticamente todos os elementos de um processo de software (conforme Seção 2.6.1.4).

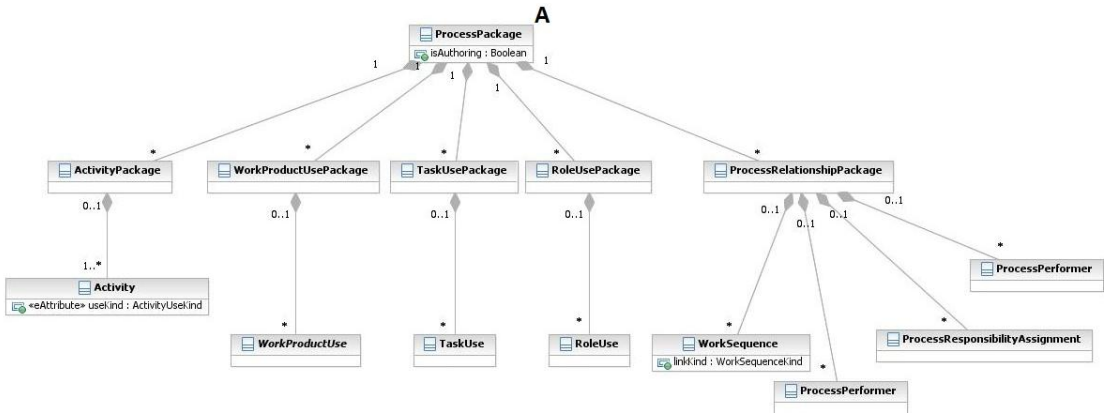

Figura 39 – Inclusão de metaclasses no pacote *Process with Methods* para organizar o conteúdo dos processos de software

Embora, como descrito acima, o principal objetivo da criação das novas metaclasses tenha sido a organização de conteúdo, tanto para o repositório (*Method Content)* quanto para os processos de software (*Process Package*), as metaclasses criadas especificamente para os processos (aquelas mostradas na Figura 39) possuem também outra função. Tais metaclasses são responsáveis por evitar a duplicação de conteúdo em um processo de software gerado a partir do metamodelo sSPEM 2.0. Diz-se isto, pois, originalmente, na especificação do metamodelo SPEM 2.0 é definido que uma instância de um elemento de processo que é criado dentro de uma atividade só deverá estar disponível para ser utilizada em suas subatividades. Isso quer dizer que, se esse elemento precisa ser utilizado em outra atividade, ele precisará ser criado novamente, gerando, assim, duplicações de elementos em um processo de software.

Com base nesse contexto, para evitar duplicações de elementos é definido que qualquer elemento criado dentro de um pacote é considerado um elemento global, ou

seja, pode ser utilizado por qualquer atividade do processo. Além disso, as relações que são definidas nos pacotes de relacionamento podem estar associadas a elementos que estão em qualquer atividade do processo.

Seguindo com as modificações realizadas no pacote *Process with Methods* foi realizada a inclusão de um atributo na metaclasse *ProcessPackage* chamado *isAuthoring,* como mostrado na Figura 39, indicado pela Letra A. Esse atributo, que é do tipo *boolean,* permite diferenciar quando um processo de software está sendo definido (*isAuthoring = true*) ou quando as atividades de adaptação estão sendo realizadas no processo em questão (*isAuthoring = false)*. Baseado na função desse novo atributo, é proposto, que cada *Process Package* represente apenas um processo de software. Assim, a criação de cada novo processo de software deverá ser feito em uma nova instância da metaclasse *ProcessPackage*.

Outro conjunto de modificações realizados no pacote *Process with Methods* foi relacionado com a alteração de alguns relacionamentos entre metaclasses existentes. O primeiro relacionamento alterado foi entre as metaclasses *TaskUse* e *ProcessPerformer*, conforme pode ser visto na Figura 40, indicado pela Letra A. Observa-se, nessa figura, que através da multiplicidade estabelecida, toda instância da metaclasse *TaskUse* deverá estar associada a pelo menos uma instância da metaclasse *ProcessPerformer.* Isso garante que toda tarefa será associada a um papel no processo de software. Originalmente, no metamodelo SPEM 2.0, a relação modificada possui multiplicidade 0..*, o que não garante que uma instância de *TaskUse* será associada a uma instância de *RoleUse* através de um *ProcessPerformer.*

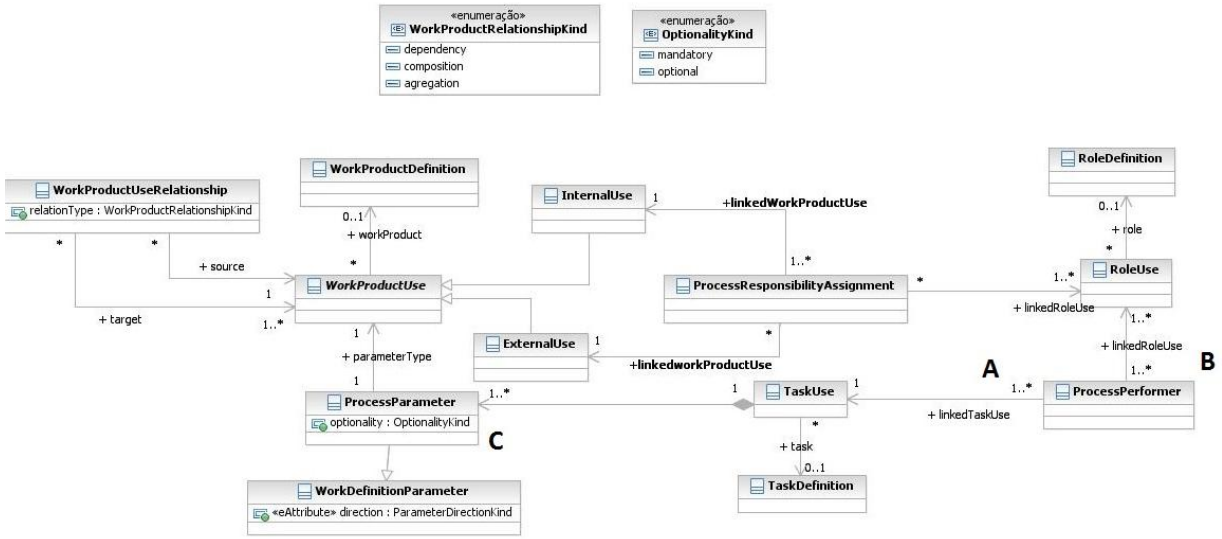

Figura 40 – Relações alteradas no pacote *Process with Methods*

Alterações também foram feitas sobre o relacionamento entre as metaclasses *ProcessPerformer* e *RoleUse*, conforme indicado pela Letra B da Figura 40. Esse relacionamento, no metamodelo original SPEM 2.0, estabelece que uma instância de *RoleUse* pode estar associada a 0..* instâncias de *ProcessPerformer*. Nesta pesquisa, a multiplicidade da relação em questão foi alterada para 1..*, garantindo que toda instância de *RoleUse* será associada a uma instância de *TaskUse* através de um *ProcessPerformer*. Dessa maneira, fica garantido que todo papel irá sempre participar da execução de pelo menos uma tarefa em um processo de software.

Por fim, o último relacionamento que teve informações de multiplicidade alterada foi entre as metaclasses *TaskUse* e *ProcessParameter,* conforme pode ser visto na Figura 40, indicado pela Letra C. No metamodelo original SPEM 2.0, é definido que toda instância de *TaskUse* pode possuir nenhum ou vários parâmetros através da criação de instâncias da metaclasse *ProcessParameter*. Nesta pesquisa a multiplicidade dessa relação foi alterada para 1..*, estabelecendo que toda instância de *TaskUse* deverá possuir ao menos uma instância da metaclasse *ProcessParameter*. Isso garante que todas as tarefas de um processo de software possuirão pelo menos um parâmetro de entrada e/ou saída em termos de produtos de trabalho.

4.3.3.2 Regras de Boa-Formação do Pacote Process with Methods

Inicialmente, é importante considerar que todas as regras de boa-formação definidas para o pacote *Process Structure* e *Method Content* estão automaticamente incluídas no pacote *Process with Methods* através do mecanismo de *merge*. Basicamente, as novas regras de boa-formação incluídas no pacote *Process with Methods* são as regras de boa-formação para consistência que envolvem o elemento de processo tarefa e seus relacionamentos com outros elementos de processo.

As primeiras regras apresentadas a seguir são as regras que estabelecem, respectivamente, que uma tarefa deve estar associada a pelo menos uma instância da metaclasse *ProcessPerformer* e que todo papel em um processo de software precisa participar da execução de pelo menos uma tarefa. Faz-se necessário considerar que ambas as regras de boa-formação citadas acima e descritas a seguir são expressas com as alterações entre as multiplicidades de relações do pacote *Process with Methods* (ver Figura 40).

Regra # 10 – Uma tarefa (TaskUse) deve possuir ao menos um relacionamento com a metaclasse ProcessPerformer.

Semântica*:* Toda instância da metaclasse *TaskUse* deve estar associada no atributo *linkedTaskUse* de pelo menos uma instância da metaclasse *ProcessPerformer*.

Regra # 48 – Um papel (RoleUse) deve possuir ao menos um relacionamento com a metaclasse ProcessPerformer.

Semântica*:* Toda instância da metaclasse *RoleUse* deve estar associada no atributo *linkedRoleUse* de pelo menos uma instância da metaclasse *ProcessPerformer*.

Seguindo com a definição das regras de boa-formação para o elemento de processo tarefa e para os outros elementos de processo relacionados com este elemento tem-se:

Regra # 9 – Uma tarefa (TaskUse) deve modificar ao menos um produto de trabalho do tipo interno ou externo (InternalUse ou ExternalUse) e/ou produzir ao menos um produto de trabalho do tipo interno (InternalUse).

Semântica*:* Toda instância da metaclasse *TaskUse* deve possuir ao menos uma instância da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*out*" ou "*inout*".

Regra #11 – Todas as dependências de um produto de trabalho do tipo interno (InternalUse) devem estar conectadas como entrada nas suas tarefas (TaskUse) de produção.

Semântica: Dado que existe uma instância da metaclasse *WorkProductUseRelationship 1* com o seguintes valores de atributos: *relationType* igual a *dependency*; *source* associado com a instância da metaclasse *InternalUse A*; e *target* associado com as instâncias da metaclasse *ExternalUse B* e *C*. E, que a instância da metaclasse *InternalUse A* está associada a uma instância da metaclasse *TaskUse T1* através de uma instância da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*out*". Nesse caso, as instâncias da metaclasse *ExternalUse B* e *C* deverão também estar associadas a instância da metaclasse *TaskUse T1* através de instâncias da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*in*".

Regra #24 – Toda tarefa (TaskUse) que produz um produto de trabalho do tipo interno (InternalUse) deverá estar associada com ao menos um papel (RoleUse) que é responsável por este produto de trabalho.

Semântica: Dado que existe uma instância da metaclasse *ProcessResponsabilityAssignment 1* com os seguintes valores de atributo: *linkedWorkProductUse* associado com a instância da metaclasse *InternalUse A* e *linkedRoleUse* associado com a instância da metaclasse *RoleUse R1*. E, que a instância da metaclasse *InternalUse A* está associada a uma instância da metaclasse *TaskUse T1* através de uma instância da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*out*". Nesse caso, deverá existir uma instância da metaclasse *ProcessPerformer* com o seguintes valores de atributo: *linkedTaskUse* associado com a instância da metaclasse *TaskUse T1* e *linkedRoleUse* associado com a instância da metaclasse *RoleUse R1.*

Outras regras de boa-formação, envolvendo o elemento de processo tarefa e o elemento de processo atividade também foram estabelecidas. Essas regras definem, respectivamente, que toda atividade deve possuir pelo menos uma tarefa e que as entradas e saídas de uma atividade devem ser compatíveis com as entradas e saídas de suas atividades e tarefas internas. Caso exista , por exemplo, um parâmetro (*ProcessParameter*) na *atividade A* (*Activity*) indicando que ela produz o *produto de trabalho interno X* (*InternalUse*), deve existir pelo menos uma instância de tarefa (*TaskUse*) dentro da *atividade A* que produz o *produto de trabalho interno X* (*InternalUse*).

Regra # 21 – Uma atividade (Activity) deve possuir ao menos uma tarefa (TaskUse).

Semântica*:* Toda instância da metaclasse *Activity* deve possuir como parte ao menos uma instância da metaclasse *TaskUse*.

Regra #22 – Os parâmetros de entrada e/ou saída (ProcessParameter) de uma atividade (Activity) devem ser compatíveis com os parâmetros de entrada e/ou saída (ProcessParameter) de suas tarefas (TaskUse).

Semântica*:* Para toda instância da metaclasse *ProcessParameter* que é associada com uma instância da metaclasse *Activity 1* deve existir outra instância da metaclasse *ProcessParameter* com os mesmos valores para os atributos *direction* e *parameterType*. Esta outra instância da metaclasse *ProcessParameter* deverá estar associada a uma instância da metaclasse *TaskUse* que é parte da metaclasse *Activity 1*.

Prosseguindo com a definição de regras de boa-formação para o elemento tarefa, têm-se as regras para este elemento relacionadas com os aspectos de duplicidade e opcionalidade. Tais regras são:

Regra #23 – Um papel (RoleUse) obrigatório deve ser associado a pelo menos uma tarefa (TaskUse) obrigatória.

Semântica: Toda instância da metaclasse *RoleUse* que possui o valor '*false*' para o atributo *isOptional* deve estar associada ao atributo *linkedRoleUse* de pelo menos uma instância da metaclasse *ProcessPerformer,* onde o atributo *linkedTaskUse* dessa instância de *ProcessPerformer* deve ser associado a uma instância da metaclasse de *TaskUse* que possui o atributo *isOptional* igual a '*false'*.

Regra #25 – Uma atividade (Activity) opcional não deve possuir elementos obrigatórios.

Semântica: Toda instância da metaclasse *Activity* que possui o valor "*true*" para o atributo *isOptional* não deve possuir como parte nenhuma instância das metaclasses *InternalUse*, *ExternalUse, TaskUse* e *RoleUse* com o valor "*false*" para seus respectivos atributos *isOptional.*

Regra #26 – Uma atividade (Activity) obrigatória devel possuir ao menos uma tarefa (TaskUse) obrigatória.

Semântica: Toda instância da metaclasse *Activity* que possui o valor "*false*" para o atributo *isOptional* deve possuir como parte ao menos uma instância da metaclasse *TaskUse* com o valor "*false*" para seu atributo *isOptional.*

Regra #27 – Um produto de trabalho do tipo externo (ExternalUse) obrigatório deve ser consumido e/ou modificado por pelo menos uma tarefa (TaskUse) obrigatória.

Semântica: Toda instância da metaclasse *ExternalUse* que possui o valor "*false*" para o atributo *isOptional* deve estar associado ao atributo *parameterType* de pelo uma instância da metaclasse *ProcessParameter* que tenha o valor do atributo *direction* igual a "*in*" ou "*inout*" e que seja parte de uma instância da metaclasse *TaskUse* com o valor "*false*" para o atributo *isOptional*.

Regra #28 – Um produto de trabalho do tipo interno (InternalUse) obrigatório deve ser produzido por pelo menos uma tarefa (TaskUse) obrigatória.

Semântica: Toda instância da metaclasse *InternalUse* que possui o valor "*false*" para o atributo *isOptional* deve estar associado ao atributo *parameterType* de pelo uma instância da metaclasse *ProcessParameter* que tenha o valor do atributo *direction* igual a "*out"* e que seja parte de uma instância da metaclasse *TaskUse* com o valor "*false*" para o atributo *isOptional*.

Regra #30 – Uma tarefa (TaskUse) obrigatória deve modificar pelo menos um produto de trabalho do tipo externo (ExternalUse) obrigatório e/ou produzir ao menos um produto de trabalho do tipo interno (InternalUse) obrigatório.

Semântica: Toda instância da metaclasse *TaskUse* que possui o valor "*false*" para o atributo *isOptional* deve possuir ao menos uma instância da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*out*" e o valor do atributo *parameterType* associado com uma instância da metaclasse *InternalUse* que possui o valor "*false*" para o atributo *isOptional* ou uma instância da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*inout*" e o valor do atributo *parameterType* associado com uma instância da metaclasse *InternalUse* ou da metaclasse *ExternalUse* que possui o valor "*false*" para o atributo *isOptional.*

Regra #31 – Toda tarefa (TaskUse) obrigatória que produz um produto de trabalho do tipo interno (InternalUse) obrigatório deverá estar associada com ao menos um papel (RoleUse) obrigatório que é responsável por este produto de trabalho.

Semântica: Dado que existe uma instância da metaclasse *ProcessResponsabilityAssignment 1* com os seguintes valores de atributo: *linkedWorkProductUse* associado com a instância da metaclasse *InternalUse A* (com atributo *isOptional* igual a "*false*") e *linkedRoleUse* associado com a instância da metaclasse *RoleUse R1* (com atributo *isOptional* igual a "*false*"). E, que a instância da metaclasse *InternalUse A* está associada a uma instância da metaclasse *TaskUse T1* (com atributo *isOptional* igual a "*false*") através de uma instância da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*out*". Nesse caso, deverá existir uma instância da metaclasse *ProcessPerformer* com o seguintes valores de atributo: *linkedTaskUse* associado com a instância da metaclasse *TaskUse T1* e *linkedRoleUse* associado com a instância da metaclasse *RoleUse R1.*

Regra #32 – Um tarefa (TaskUse) obrigatória deve ser associada a pelo menos um papel (RoleUse) obrigatório.

Semântica: Toda instância da metaclasse *TaskUse* que possui o valor '*false*' para o atributo *isOptional* deve estar associada ao atributo *linkedTaskUse* de pelo menos uma instância da metaclasse *ProcessPerfomer,* onde o atributo *linkedRoleUse* dessa instância de *ProcessPerfomer* deve ser associado a uma instância da metaclasse de *RoleUse* que possui o atributo *isOptional* igual a '*false'*.

Regra #35 – Um papel (RoleUse) não pode ser definido mais de uma vez como executor da mesma tarefa (TaskUse).

Semântica: Não pode existir mais de uma instância da metaclasse *ProcessPerformer* que possua a mesma instância da metaclasse *RoleUse* selecionada para o atributo *linkedRoleUse* e a mesma instância da metaclasse *TaskUse* selecionada para o atributo *linkedTaskUse*.

Regra #40 - Uma tarefa (TaskUse) não pode possuir dois ou mais parâmetros de entrada e/ou saída (ProcessParameter) iguais.

Semântica*:* Nenhuma instância da metaclasse *TaskUse* pode possuir mais de uma instância da metaclasse *ProcessParameter* com os mesmos valores para os atributos *direction* e *parameterType*.

Uma vez que o elemento tarefa pode também ser seqüenciado, no metamodelo sSPEM 2.0, regras de boa-formação relacionadas com este aspecto também foram definidas. Para realizar as definições necessárias, inicialmente, algumas regras de boa-formação do pacote *Process Structure*, as quais estão incluídas nesse pacote pelo mecanismo de *merge*, foram modificadas. As regras modificadas foram aquelas relacionadas com o sequenciamento de atividades e são: *Regra #12*, *Regra #13, Regra #14, Regra #15, Regra #16, Regra #17, Regra #18, Regra #41, Regra #42, Regra #43, Regra #44* e *Regra #45*.

A modificação realizada no pacote *Process with Methods* sobre as referidas regras é que, agora, elas passam também a funcionar com instâncias da metaclasse *TaskUse*. Uma descrição completa sobre a redefinição das regras de boa-formação citadas acima podem ser encontradas no Apêndice B. Contudo, apenas enquanto uma forma de demonstrar como as modificações foram realizadas, mostra-se-á, a seguir, a *Regra #12* do pacote *Process with Methods*:

Regra #12 – Um processo de software deve possuir exatamente um nodo final.

Semântica: Apenas uma instância da metaclasse *Activity* ou da metaclasse *TaskUse* com o valor do atributo *specialNode* igual a "*end*" deverá ser instanciada em um processo de software.

Conforme exposto, a descrição da regra de boa-formação acima é exatamente igual à regra de boa-formação descrita no pacote *Process Structure*. Contudo, no pacote *Process with Method*s, a semântica desta regra é diferente, visto que, agora, o elemento

tarefa também poderá ser sequenciado e, dessa forma, apenas um dos elementos tarefa e atividade poderá ser instanciado para representar o nodo final do processo de software.

Além das modificações realizadas sobre algumas regras de boa-formação, novas regras de boa-formação relacionadas com sequenciamento de atividades e tarefas também foram definidas. A primeira delas, a qual é considerada uma restrição estabelecida nesta pesquisa, define que em um sequenciamento, uma atividade não deve ter como predecessora e/ou sucessora uma tarefa, e vice-versa. Tal regra de boa-formação estabelece que o sequenciamento de processos deverá ser realizado entre elementos do mesmo tipo.

Regra #50 – Em um sequenciamento definido para um processo de software uma atividade não deve ter como predecessora e/ou sucessora uma tarefa, e vice-versa.

Semântica: Toda instância da metaclasse *WorkSequence* só pode possuir instâncias de uma mesma metaclasse associadas aos atributos *predecessor* e *successor*.

Outra regra de boa-formação definida é aquela que estabelece que os nodos inicial e final, em um sequenciamento, devem ser do mesmo tipo, ou seja, quando se define uma atividade como nodo inicial, deve definir-se também uma atividade como nodo final, e vice-versa. O mesmo ocorre em se tratando dos nodos inicial e final, utilizando-se do elemento tarefa.

Regra #6 – Em um sequenciamento definido para um processo de software os nodos inicial e final devem ser do mesmo tipo.

Semântica: Não pode existir uma instância da metaclasse *Activity* com o valor do atributo *specialNode* igual a “*start*” ou “*end*” e uma instância da metaclasse *TaskUse* com o valor do atributo *specialNode* igual a “*start*” ou “*end*”.

Prosseguindo com a definição de regras de boa-formação que são relacionadas com o sequenciamento das tarefas, definiram-se as duas últimas regras, também relacionadas com o elemento produto de trabalho, que estabelecem as seguintes condições: (1) todas as dependências de um produto de trabalho devem ser produzidas antes dele em um processo de software; e (2) todos os produtos de trabalho devem ser produzidos antes de serem consumidos.

Regra # 19 – Todas as dependências de um produto de trabalho do tipo interno (InternalUse) devem ser produzidas antes deste produto de trabalho em um processo de software.

Semântica: Toda instância da metaclasse *WorkProductUseRelationship* que possui o valor do atributo *relationType* igual a "*dependency*" deverá ter as instâncias da metaclasse *InternalUse* que estão associadas no seu atributo *target* produzidas antes da instância da metaclasse *InternalUse* que está associada no seu atributo *source*. Em um sequenciamento que considera a divisão das tarefas em *tarefas_início* e *tarefas_fim* isso quer dizer que as intâncias da metaclasse *InternalUse* associadas no atributo *target* de uma instância da metaclasse *WorkProductUseRelationship* deverão ser produzidas por tarefas que precedem a(s) tarefa(s) que produzem a intância da metaclasse *InternalUse* associada no atributo *source* da mesma instância da metaclasse *WorkProductUseRelationship.*

Regra # 20 – Todos os produtos de trabalho do tipo interno (InternalUse) devem ser produzidos antes de serem consumidos em um processo de software.

Semântica: Para toda instância da metaclasse *InternalUse* que esta associada a uma instância da metaclasse *TaskUse* através de uma instância da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*in*" ou "*inout*", deverá existir pelo menos uma associação da mesma instância da metaclasse *InternalUse* com uma instância da metaclasse *TaskUse* através de uma instância da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*out*". Considerando a precedência das tarefas, a instância da metaclasse *TaskUse* que possui a instância da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*out*" deverá sempre preceder a tarefa que possui a instância da metaclasse *ProcessParameter* com o valor do atributo *direction* igual a "*in*" ou "*inout*".

4.3.4 Pacote *Method Plugin*

Não foram necessárias alterações para nenhuma metaclasse e/ou relacionamento do pacote *Method Plugin*. Contudo, é necessário considerar que através do mecanismo de *merge* todos os elementos do metamodelo sSPEM 2.0 estão incluídos no pacote *Method Plugin*.

Com relação à definição de regras de boa-formação específicas para o pacote *Method Plugin*, uma regra foi definida para a metaclasse *Variability* e seus relacionamentos. Contudo, considerando que a metaclasse *Variability* e seus relacionamentos são considerados como mecanismos de adaptação de processos no

metamodelo sSPEM 2.0, a regra de boa-formação específica é descrita na próxima seção deste trabalho.

4.4 Mecanismos de Adaptação de Processos

Nesta seção, os mecanismos de adaptação e duas das regras de boa-formação para consistência de um processo de software são apresentados. Inicialmente, serão descritos os mecanismos de adaptação dos relacionamentos *usedActivity* e *supressedBreakdownElement,* os quais são definidos no pacote *Process Structure*. Em seguida, será exposto o mecanismo de adaptação da metaclasse *Variability*, a qual é definida no pacote *Method Plugin*.

Faz-se necessário considerar que todos os mecanismos acima, embora sejam considerados como mecanismos de adaptação do metamodelo original SPEM 2.0, podem ser utilizados também para a definição de processos de software. Desse modo, nas seções seguintes, as diferenças de funcionamento desses mecanismos nas atividades adaptação e definição dos processos de software, serão explicadas. Salienta-se também que as duas regras de boa-formação que serão apresentadas para os mecanismos *usedActivity* e *Variability* não foram relacionadas com nenhuma premissa para consistência de processos de software identificadas nesta pesquisa. Isso porque tais regras são específicas dos mecanismos *usedActivity* e *Variability* e foram definidas para garantir alguns aspectos de consistência durante a aplicação desses mecanismos.

4.4.1 Relacionamentos *usedActivity* e *supressedBreakdownElement*

O relacionamento *usedActivitiy* é definido no pacote *Process Structure* como um autorrelacionamento para a metaclasse *Activity*. Este relacionamento, detalhado na Seção 2.6.1.5 deste trabalho, é considerado como um mecanismo de adaptação do metamodelo original SPEM 2.0, pois permite que o conteúdo definido para uma atividade seja reutilizado em outras atividades de um mesmo processo de software ou, até mesmo, a partir de atividades de outros processos. Isso possibilita, por exemplo, o reuso de um processo completo, já que, no metamodelo original SPEM 2.0, e também no metamodelo sSPEM 2.0, o elemento atividade é indicado como o elemento a ser usado para instanciação de processos de software.

Nenhuma alteração foi realizada em termos de metaclasses ou relacionamentos no pacote *Process Structure,* considerando o autorrelacionamento *usedActivity*. Contudo, uma regra de boa-formação específica para este relacionamento foi definida. Além disso, as semânticas dos tipos de herança (*extension*, *localcontribution* e *localreplacement*) definidos para o relacionamento *usedActivity* também foram alteradas visando manter a consistência das instâncias da metaclasse *Activity*.

A regra de boa-formação definida para o relacionamento *usedActivity* estabelece que uma atividade não pode herdar seu próprio conteúdo, ou seja, uma instância de atividade não pode estabelecer um autorrelacionamento *usedActivity* do tipo *extension*.

Regra #46 – Uma atividade não pode herdar seu próprio conteúdo.

Semântica: Nenhuma instância da metaclasse *Activity* deverá possuir um autorrelacionamento *usedActivity*.

Seguindo com as definições sobre o relacionamento *usedActivity,* se encontram as alterações realizadas sobre as semânticas dos tipos de herança *extension*, *localContribution* e *localReplacement* (ver Seção 2.6.1.5).

A primeira modificação realizada foi realizada especificamente sobre os tipos de herança *localContribution* e *localReplacement* que são utilizados no metamodelo origianl SPEM 2.0 em conjunto com o tipo de herança *extension.* Tal modificação define que, em ambos os tipos de herança, as atividades que apontam para outras atividades, utilizando-se destes valores, não podem estar vazias, ou seja, no caso de uma contribuição (*localContribution*), algum elemento deve existir na atividade de origem para que realmente exista uma contribuição quando o conteúdo da atividade de destino for herdada. Já no caso de uma substituição (*localReplacement*), a atividade de origem não pode estar vazia, uma vez que seu conteúdo substituirá o conteúdo da atividade de destino.

A próxima alteração realizada para o relacionamento *usedActivity* foi feita sobre os tipos de herança *extension* e *localContribution* e diz respeito ao resultado dos elementos herdados quando estes tipos de valores são utilizados. Na semântica definida pela especificação do metamodelo original SPEM 2.0, é dito apenas que, quando uma atividade aponta para outra atividade com o valor do atributo *useKind* igual a *extension,* todo conteúdo da atividade que recebe o apontamento deve ser copiado para a atividade que realiza o apontamento. O mesmo ocorre quando o tipo de herança *localContribution* é utilizado. A única diferença é que, para o uso do *localContribution* ser permitido, ele deve

ser utilizado em conjunto com o valor *extension* e o resultado da interpretação do tipo de herança *localContribution* é a soma do conteúdo herdado com o conteúdo da atividade que faz o apontamento.

Embora o funcionamento de ambos os tipos de herança citados estejam bem estabelecidos no metamodelo original SPEM 2.0, algumas inconsistências são encontradas quando o resultado desses mecanismos é analisado em detalhes. O problema ocorre quando atividades que estão sendo herdadas se relacionam com elementos do processo que estão fora desta atividade. Especificamente, sobre essa situação, não são encontradas referências na semântica dos tipos de herança *extension* e *localContribution* do metamodelo original SPEM 2.0.

Para facilitar o entendimento do que foi exposto, a Figura 41 mostra um exemplo, contendo uma pequena estrutura de atividades e um relacionamento *usedActivity* com o tipo de herança *extension* sendo realizado. Torna-se importante considerar que, tanto para o exemplo mostrado na Figura 41 quanto para todos os exemplos que serão utilizados nesta seção, várias informações de processo (informações de sequenciamento, criação de nodos inicial e final, entre outros) foram omitidas e dessa forma, os exemplos podem possuir inconsistências de processo. Isso foi feito apenas como forma de simplificar os exemplos sobre os tipos de herança do relacionamento *usedActivity*.

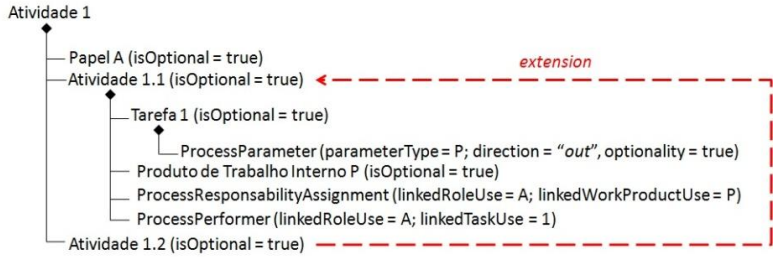

Figura 41 – Exemplo de aplicação do relacionamento *usedActivity* do tipo *extension*

No exemplo da Figura 41, as metaclasses que representam os relacionamentos *ProcessParameter, ProcessResponsabilityAssignment* e *ProcessPerformer* tiveram suas nomenclaturas descritas em inglês por não terem sido traduzidas nesta pesquisa como elementos de processos. Elas expressam, respectivamente, que a *Tarefa 1* produz o *Produto de Trabalho P*, o *Produto de Trabalho P* possui como responsável o *Papel A* e que a *Tarefa 1* é desempenhada pelo *Papel A*. Observa-se, no exemplo, que algumas relações definidas dentro da *Atividade 1.1* utilizam o elemento de processo *Papel A* definido na sua atividade superior, ou seja, tais relações utilizam um elemento que está fora da *Atividade 1.1*.

Ainda observando o exemplo da Figura 44, nota-se que a *Atividade 1.2* aponta para a *Atividade 1.1,* utilizando o relacionamento *usedActivity* com o tipo de herança *extension*. Nesse tipo de situação o metamodelo original SPEM 2.0 não deixa claro exatamente como devem ser os resultados dos tipos de herança *extension* e *localContribution*.

Nesta pesquisa, definiu-se que, nos mecanismos de herança *extension* e *localContribution,* todos os elementos que são utilizados por uma atividade, através de seus relacionamentos, e estão definidos em suas atividades superiores (atividades pai) ou estão definidos nos pacotes de elementos, são considerados como parte desta atividade e, desse modo, serão herdados. Além disso, no caso da atividade a ser herdada possuir um ou mais elementos envolvidos em instâncias de relacionamentos definidos nos pacotes de relacionamentos, tais relacionamentos deverão ser herdados sempre que possível.

Considera-se também possível herdar uma instância de relacionamento se ele não apresentar nenhuma inconsistência depois da interpretação da herança. Por exemplo, para que um relacionamento de responsabilidade sobre um produto de trabalho (*ProcessResponsabilityAssignment*) seja definido sem inconsistências, ele precisa estar associado a exatamente um produto de trabalho e pelo menos um papel.

Assim, considerando um processo que contenha a definição de um relacionamento de responsabilidade definido nos pacotes de relacionamento e a definição do produto de trabalho e do papel que estão associados a este relacionamento em diferentes atividades do processo, só será possível herdar este relacionamento, se ambas as atividades que contém o produto de trabalho e o papel forem herdadas. O mesmo ocorre com as seguintes relações do processo: *WorkSequence*, *WorkProductRelationshipUse* e *ProcessPerformer*.

No caso específico da relação *WorkSequence,* que sempre é definida nos pacotes de relacionamento (conforme Seção 5.3) e, para ser consistente, necessita de exatamente uma instância das metaclasses *Activity* ou *TaskUse* associada como predecessor, bem como de uma instância das metaclasses *Activity* ou *TaskUse* associada como sucessor, sabe-se que esse tipo de relação, muitas vezes, não será herdada, causando, assim, inconsistências no sequenciamento dos processos de software. Contudo, é importante considerar que, embora inconsistências sejam geradas através do

mecanismo de herança, todas elas são esperadas e apontadas pelas regras de boa-formação para consistência, as quais foram apresentadas na Seção 4.3.

Ainda quanto à herança de relacionamentos, é necessário ressaltar que também se considera possível herdar um relacionamento, se, no processo que está recebendo o conteúdo herdado, já existir um ou mais elementos que pertencem a esse relacionamento. Considere, por exemplo, que durante um mecanismo de herança entre dois processos diferentes, um produto de trabalho tenha sido herdado e não tenha sido possível herdar o seu relacionamento de responsabilidade, devido ao papel associado a essa relação não fazer parte do conteúdo a ser herdado (não estar definido nos pacotes de elementos, não estar definido na atividade que é herdada e nem em uma de suas atividades superiores). Neste caso do exemplo, se a atividade que contém o papel definido no relacionamento for herdada mais tarde será possível herdar também o relacionamento. Isso porque será detectado que um dos elementos que fazem parte deste relacionamento já se encontra no processo que está recebendo o conteúdo herdado.

Para facilitar o entendimento do exposto acima, a Figura 42 mostra a interpretação do mecanismo de herança proposto na Figura 41. Observa-se, na referida figura, que, na solução proposta por esta pesquisa, o *Papel A* é considerado como parte da *Atividade 1.1*. A justificativa a isso se dá porque esse elemento é utilizado por relações da *Atividade 1.1* e é definido na sua atividade superior. Assim, o resultado da extensão feita através das atividades *Atividade 1.1* e *Atividade 1.2,* mostra que todos os elementos herdados na *Atividade 1.2* utilizam o *Papel A*.

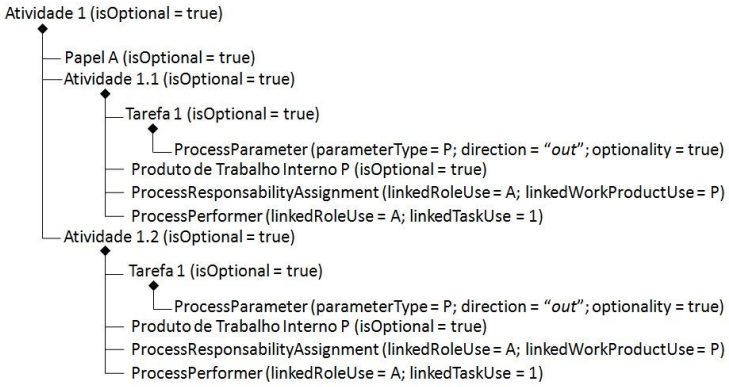

Figura 42 – Resultado da aplicação do relacionamento *usedActivity* do tipo *extension*

Considerando que o exemplo acima é bastante simples e não mostrou o tipo de herança *localContribution,* um novo exemplo é mostrado na Figura 43.

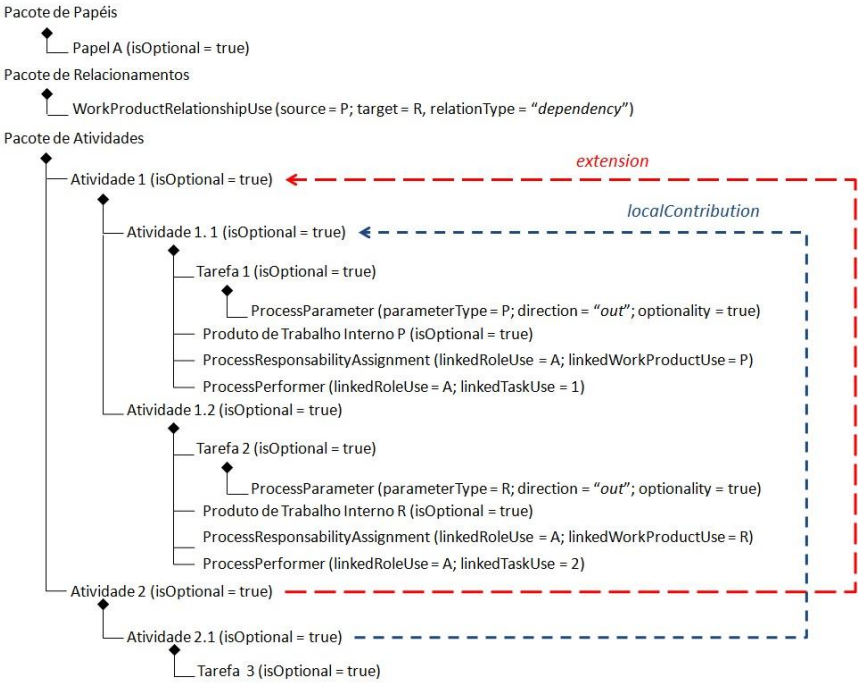

Figura 43 – Exemplo do relacionamento *usedActivity* do tipo *extension* e *localContribution*

Nota-se, na Figura acima, que além do uso do tipo de herança *localContribution,* o novo exemplo já possui a instanciação de alguns pacotes com elementos que podem ser utilizados por qualquer atividade (no exemplo o elemento *Papel A*) e com a definição de alguns relacionamentos que se utilizam de elementos instanciados em diferentes atividades. No caso do exemplo mostrado na Figura 43, o relacionamento que utiliza instâncias de elementos defindidos em atividades diferentes estabelece que o *Produto de Trabalho P* depende do *Produto de Trabalho R*.

Faz-se importante considerar que, pelas restrições de visibilidade definidas na especificação do metamodelo SPEM 2.0 (ver Seção 4.3.3.1), tanto o *Papel A* quanto o relacionamento de dependência não poderiam ter sido definidos localmente em qualquer uma das atividades. Isso porque, se fossem definidos localmente, esses elementos não poderiam ser utilizados por elementos de outras atividades (neste caso o *Papel A*), ou, ainda, utilizar (no caso do relacionamento de dependência) tais elementos.

O resultado, após a interpretação dos tipos de herança *extension* e *localContribution,* são mostrados na Figura 44.

Figura 44 – Resultado da aplicação do relacionamento *usedActivity* do tipo *extension* e *localContribution*

No resultado exibido na Figura 44, observa-se que os produtos de trabalho e os dois relacionamentos que definem as responsabilidades sobre esses produtos de trabalho foram movidos para os pacotes de produtos de trabalho e de relacionamentos, respectivamente. Isso foi feito para evitar que inconsistências relacionadas com duplicação de elementos fossem reproduzidas após os mecanismos de herança serem executados.

Para finalizar o entendimento sobre como os mecanismos de herança *extension* e *localContribution* funcionam na solução proposta, um exemplo de herança entre processos diferentes foi elaborado. Para esse exemplo, o conjunto de elementos de processo da Figura 44 é considerado. Contudo, para enriquecer a ilustração, definiu-se localmente para a *Atividade 2.1* um novo produto de trabalho chamado *Y* e considerou-se que o *Produto de Trabalho P,* que já era dependente do *Produto de Trabalho R,* também depende do *Produto de Trabalho Y*. Para estabelecer essa relação, bastou incluir o *Produto de Trabalho Y* no atributo *target* do relacionamento de dependência que se encontra no pacote de relacionamentos.

Prosseguindo com o exemplo, criou-se um novo processo de software com apenas uma atividade (*Atividade Estender*) e se definiu um relacionamento de *extension* com a *Atividade 1.1* da Figura 44. O resultado, após a interpretação do mecanismo de herança é mostrado na Figura 45.

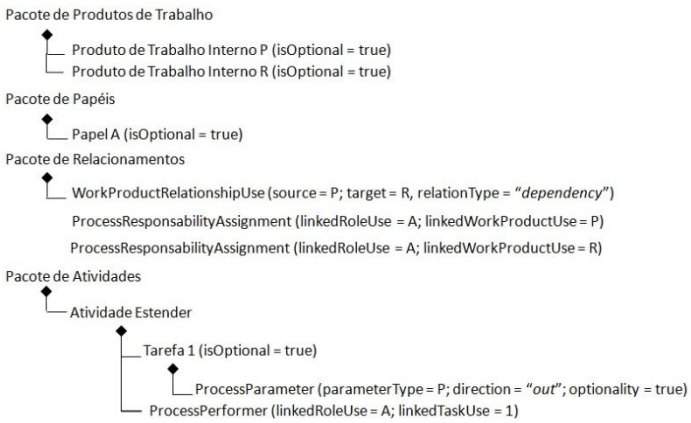

Figura 45 – Exemplo de aplicação do relacionamento *usedActivity* do tipo *extension* entre atividades de processos diferentes

O resultado exibido na Figura 45 mostra que todo conteúdo da *Atividade 1.1* foi herdado para a *Atividade Estender* juntamente com todos os elementos e relacionamentos que estavam no pacote e possuiam alguma relação com o conteúdo dessa atividade. Esse é o caso do *Produto de Trabalho R* e seu relacionamento de responsabilidade (*ProcessResponsabilityAssignment*) que foram copiados, uma vez que esse produto de trabalho é utilizado pela *Atividade 1.1* através do relacionamento de dependência que o *Produto de Trabalho P* tem com o *Produto de Trabalho R.*

A diferença da aplicação dos mecanismos de *extension* e *localContribution* entre processos diferentes é que inconsistências quase sempre são incluídas no processo que herda algum conteúdo. Nota-se na Figura 45, por exemplo, que o *Produto de Trabalho R*, o qual é do tipo Interno, não herda sua tarefa de produção, gerando, assim, uma inconsistência no processo resultante. Contudo, é importante considerar que, como já mencionado acima, todas as inconsistências causadas nos mecanismos de herança são esperadas e são apontadas pelas regras de boa-formação para consistência definidas nesta pesquisa.

As únicas inconsistências geradas através do mecanismo de herança entre dois processos diferentes, que não podem ser identificadas através das regras de boa-formação, estão relacionadas com a herança dos relacionamentos de dependência e composição. No caso específico do exemplo descrito acima, nota-se, também na Figura 45, que o relacionamento de dependência não foi herdado por completo.

No exemplo proposto, o *Produto de Trabalho P* dependia dos produtos de trabalho *R* (que encontrava-se no pacote) e *Y* (que foi definido localmente na *Atividade 2.1*) e, após o mecanismo de herança ser interpretado, o *Produto de Trabalho P* depende apenas do *Produto de Trabalho* R sem existir nenhuma referência ao *Produto de Trabalho Y*. Caso o *Produto de Trabalho R* também estivesse definido localmente dentro de outra atividade, o relacionamento de dependência nem seria criado no processo resultante e, após uma validação de consistência nenhum erro seria encontrado para tal processo. A mesma situação ocorrerá com os produtos de trabalho que mantêm relações de composição e dependem de uma de suas partes ou do todo para ser instanciado.

Para resolver a situação acima, na solução proposta toda vez que um produto de trabalho herdado em um mecanismo de herança (*extension* ou *localContribution*) depende de outros produtos de trabalho ou representa uma parte ou o todo em uma relação de composição, essa informação deverá ser herdada juntamente com esse produto de trabalho, sendo criado automaticamente uma inconsistência no processo resultante. Em se tratando de um relacionamento de dependência, a inconsistência só será resolvida quando todas as dependências do produto de trabalho em questão forem criadas no processo.

Já no caso de um relacionamento de composição, a inconsistência poderá ser resolvida de duas formas. Caso o produto de trabalho herdado represente uma parte em uma relação de composição, a inconsistência é resolvida quando o produto de trabalho que representa o todo dessa relação é criado. Mas, no caso do produto de trabalho herdado representar o todo de uma relação de composição, a inconsistência é resolvida quando qualquer uma de suas partes é criada no processo.

Com base em todo contexto evidenciado, a nova descrição para a semântica dos tipos de herança *localContribution* e *localReplacement* foi estabelecida. Um ponto importante a ser considerado é que tais semânticas, descritas abaixo, funcionam da mesma maneira para ambas as atividades de adaptação e definição dos processos de software.

- *extension:* este mecanismo permite reutilizar dinamicamente as subestruturas (elementos aninhados pela relação de composição) de uma atividades em outras atividades. Dessa forma, a atividade que é associada à outra atividade, pela relação *usedActivity* e pelo valor para *useKind* igual a *extension,* herda todo conteúdo dessa atividade. Considera-se que, durante o mecanismo de herança, todos os elementos que são utilizados, através de relacionamentos pela atividade que sofre o apontamento

extension e estão definidos em suas atividades superiores (atividades pai) ou estão definidos nos pacotes de elementos e relacionamentos, são considerados como parte dessa atividade e, desse modo, são herdados.

- *localContribution:* este mecanismo deve ser sempre utilizado em conjunto com o mecanismo de extensão (*extension*). Ele permite que adições locais (contribuições) sejam feitas em atividades que estão sendo herdadas através do mecanismo de extensão. Assim, toda atividade que aponta para outra atividade, utilizando-se do valor *localContribution* não pode estar vazia. Essa atividade deve possuir, no mínimo, um elemento de processo instanciado como parte dela. Uma atividade A, por exemplo, poderia herdar toda estrutura da atividade B pelo mecanismo de *extension*. Contudo, poderia ser necessário fazer adições locais (contribuições) para a atividade A através da relação *localContribution*. Para fazer isso, devem existir subatividades dentro das atividades A e B. Na (por exemplo, A.1), devem ser definidas as contribuições locais e, nesse momento, deve ser apontada subatividade de A para subatividade de B (por exemplo B.1). O resultado, em A.1, será todo conteúdo de B.1 somado aos elementos pré-definidos em A.1. Considera-se que durante o mecanismo de herança, todos os elementos que são utilizados através de relacionamentos pela atividade que sofre o apontamento *localContribution* e estão definidos em suas atividades superiores (atividades pai) ou estão definidos nos pacotes de elementos e relacionamentos são considerados como parte desta atividade e, dessa maneira, são herdados.

Até o momento, a única definição específica para o tipo de herança *localReplacement* foi o estabelecimento de que a atividade que realiza um apontamento deste tipo não pode estar vazia. Contudo, uma vez que o tipo de herança *localReplacement* envolve a exclusão dos elementos e relacionamentos que serão substituídos pelos novos elementos e relacionamentos, alterações em seu funcionamento foram propostos nesta pesquisa. As alterações realizadas sobre esse tipo de herança dizem respeito à inclusão de análise de impacto sobre os elementos resultantes em um processo de software, toda vez que um apontamento do tipo *localReplacement* é realizado. O principal objetivo da análise de impacto é identificar possíveis elementos e relacionamentos que, por motivo de dependências, precisam também ser excluídos.

Contudo, a análise de impacto só é necessária quando a herança que se utiliza de *localReplacement* ocorre entre duas atividades que pertencem a processos diferentes. Isso porque, quando este tipo de herança é utilizado entre atividades do mesmo processo, o conteúdo que está sendo excluído pelo mecanismo *localReplacement* ainda permanece

no processo, uma vez que ele é mantido na atividade que sofre o apontamento. Para demonstrar o funcionamento do *localReplacement,* considerando duas atividades em um mesmo processo, a Figura 46 mostra um exemplo.

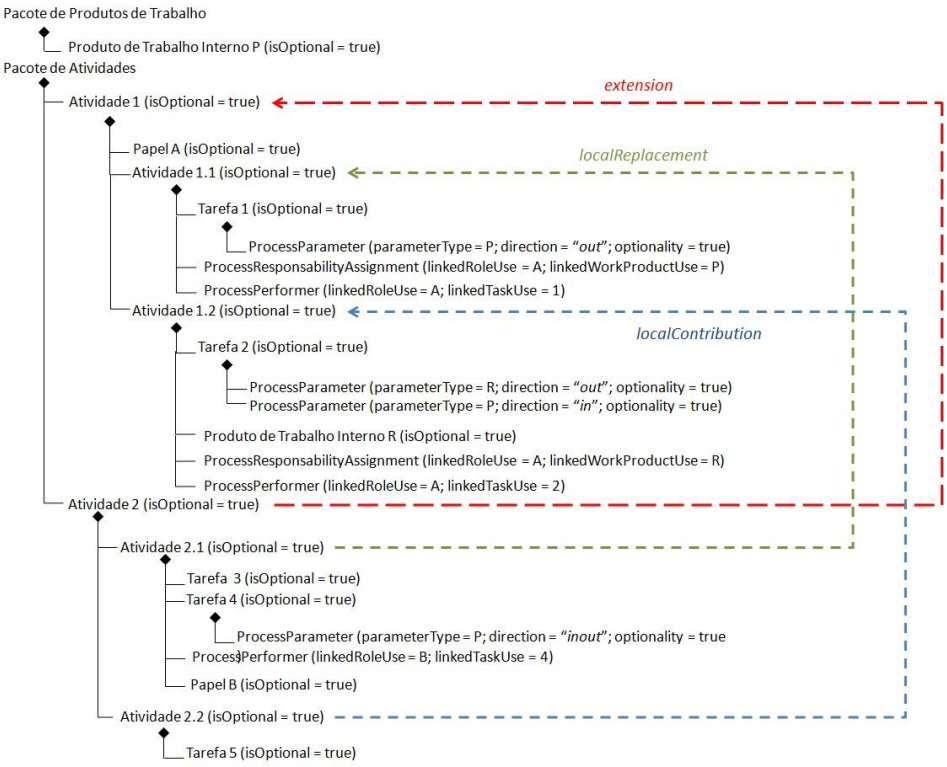

Figura 46 – Exemplo de aplicação do relacionamento *usedActivity* do tipo *extension, localContribution* e *localReplacement*

No exemplo acima, é utilizado o tipo de herança *extension* em conjunto com os tipos de herança *localContribution* e *localReplacement,* utilizando-se duas atividades de um mesmo processo de software.

O resultado da interpretação dos tipos de herança aplicados na Figura 46 é mostrado, logo abaixo, na Figura 47. Observa-se, nesse figura, que todos os elementos do processo foram mantidos no resultado da interpretação de todas as heranças. Isso ocorre porque, embora a *Atividade 2.1* tenha substituído o conteúdo da *Atividade 1.1*, durante a interpretação dos mecanismos de herança, todo o conteúdo da *Atividade 1.1* é mantido no processo e, dessa maneira, as exclusões feitas durante a substituição não impactam nenhum outro elemento do processo.

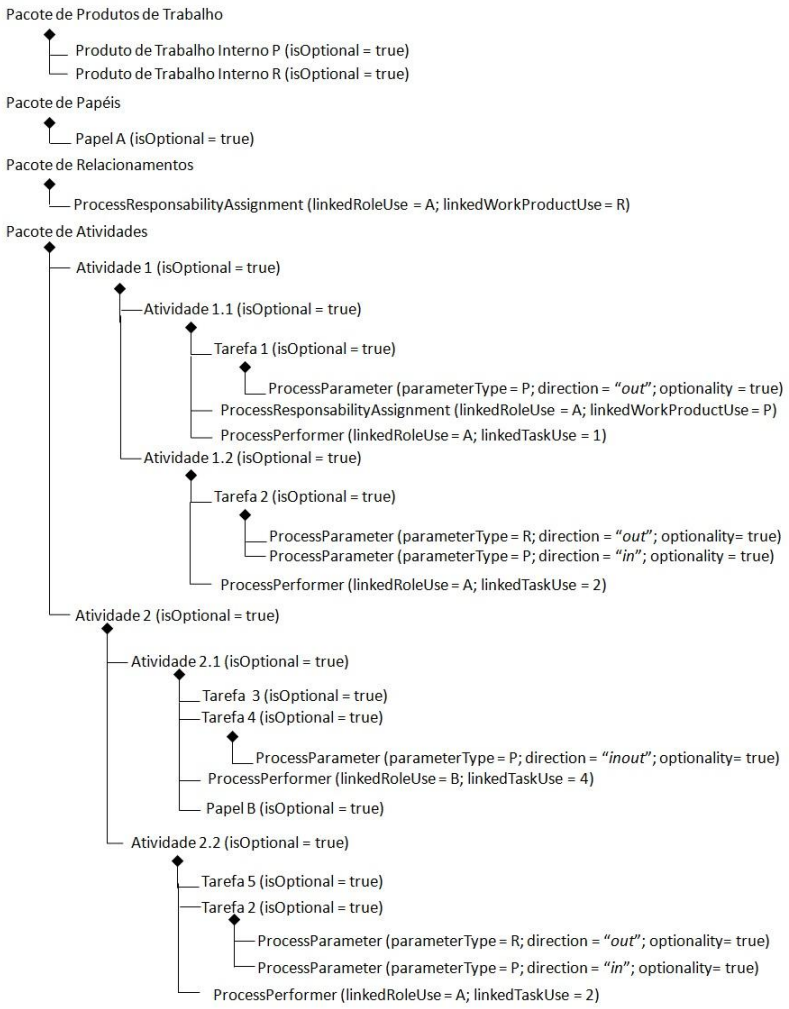

Figura 47 – Resultado da aplicação do relacionamento *usedActivity* do tipo *extension, localContribution* e *localReplacement*

Para entender a diferença da aplicação do tipo de herança *localReplacement* em atividades de mesmo processo ou de processos diferentes, a Figura 48 mostra um exemplo análogo ao da Figura 46, contudo, agora, considerando os tipos de herança sendo aplicados em atividades localizadas em processos distintos.

Observa-se, na Figura 48, que o conteúdo da *Atividade 1,* do exemplo mostrado na Figura 46, foi mantido para a atividade chamada *Atividade Processo 1*. A única modificação realizada foi no conteúdo da *Atividade 2* que, no novo exemplo, é representado pela atividade chamada *Atividade Processo 2*. O conteúdo definido, agora, para a *Atividade Processo 2* estabelece a produção de um novo *Produto de Trabalho W* na sua atividade *Atividade 2.1* que é executado por um novo papel chamado *Papel B*. Isso foi feito, pois, uma vez que o conteúdo da *Atividade 2* (Figura 46) no novo exemplo passa a estar em outro processo de software (*Atividade Processo 2*), não é possível que uma de suas tarefas modifique produtos de trabalho produzidos na *Atividade 1* (representada no novo exemplo pela *Atividade Processo 1*).

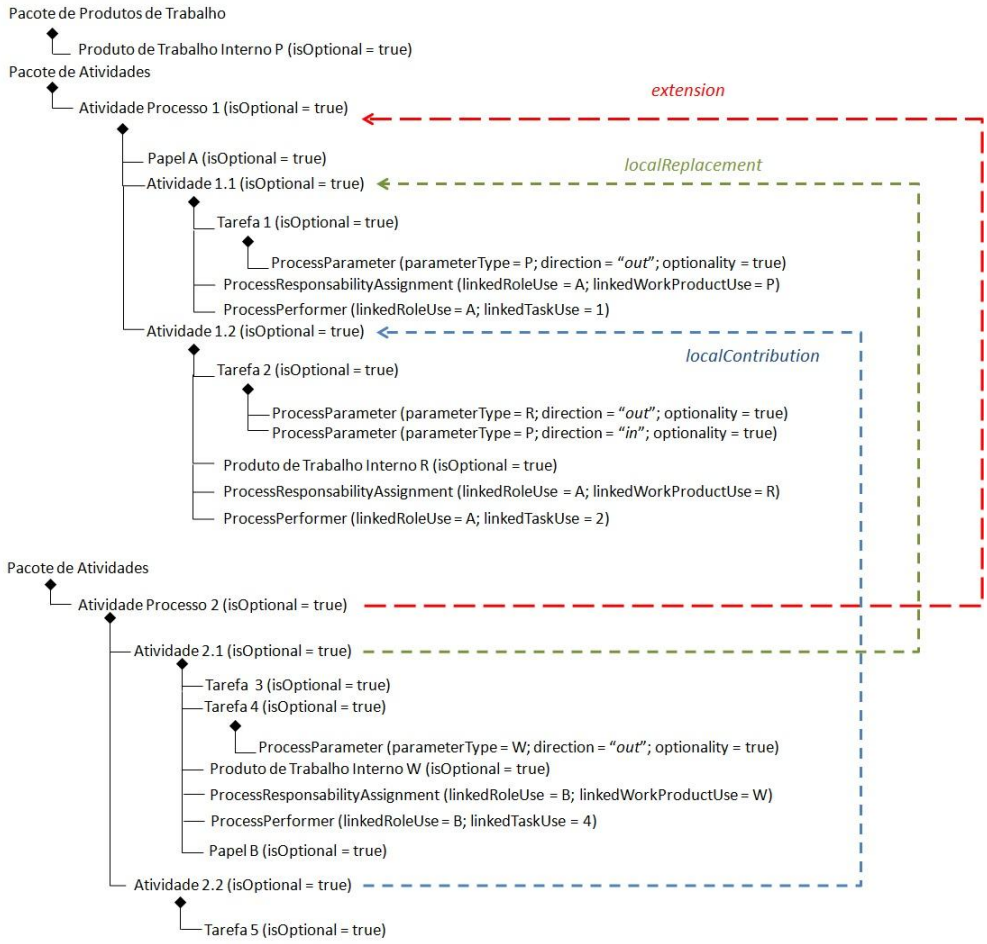

Figura 48 – Exemplo de aplicação do relacionamento *usedActivity* do tipo *extension, localContribution* e *localReplacement* entre atividades de processos diferentes

O resultado da interpretação dos tipos de herança aplicados no exemplo mostrado na Figura 48 é exibido na Figura 49.

Basicamente, o que é mostrado na Figura 49 é que todo conteúdo da *Atividade 1.2* (que pertende a *Atividade Processo 1* – Figura 48) é herdada pela *Atividade 2.2* (que pertende a *Atividade Processo 2* – Figura 49) e que o conteúdo da *Atividade 1.1* (que pertende a *Atividade Processo 1* – Figura 48) é substituido pelo conteúdo da *Atividade 2.1* (que pertende a *Atividade Processo 2* – Figura 49). Em outras palavras, no resultado final, todo o conteúdo da *Atividade 1.1,* da *Atividade Processo 1,* não existe nas subatividades da *Atividade Processo 2*.

Dessa forma, como o *Produto de Trabalho P* (que era produzido na *Atividade 1.1* da *Atividade Processo 1*) não existe mais na *Atividade Processo 2,* a análise de impacto proposta por esta pesquisa para o tipo de herança *localReplacement* indicou que o parâmetro de entrada para a *Tarefa 2* (destacado em vermelho na Figura 49) na *Atividade Processo 2* deverá ser excluído. Essa exclusão torna-se necessária, em função de que esse parâmetro indica que a *Tarefa 2* consome o *Produto de Trabalho P*, o qual, como já explicado, não existe mais nesse processo. Faz-se necessário considerar também que,

nesse exemplo, a *Tarefa 2* não precisou ser excluída, pois o parâmetro de entrada para o *Produto de Trabalho P* foi definido como opcional.

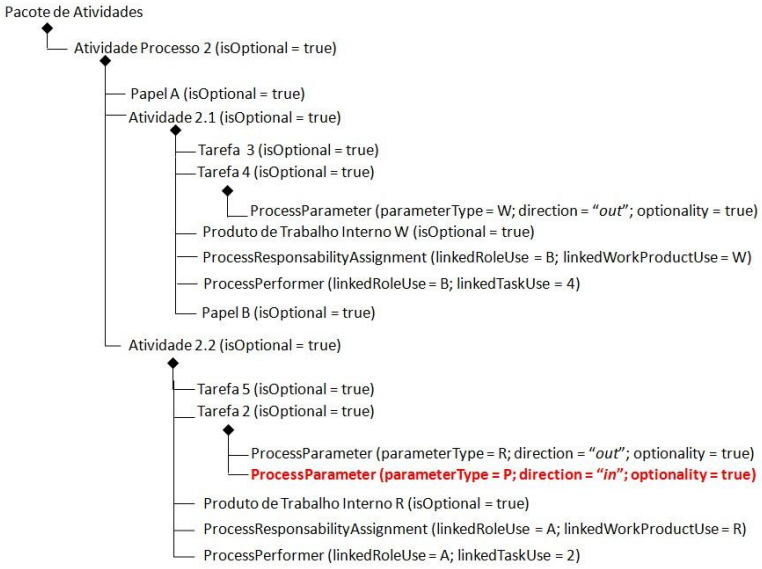

Figura 49 – Resultado da aplicação do relacionamento *usedActivity* do tipo *extension, localContribution* e *localReplacement* entre Atividades de Processos Diferentes

Para realizar a análise de impacto para o tipo de herança *localReplacement,* tornou-se necessário mapear todos os impactos causados pela exclusão de cada elemento que pode pertencer a uma atividade. O resultado desse mapeamento será apresentado ainda nesta seção juntamente com as alterações propostas para o relacionamento *supressedBreakdownElement*.

A nova semântica do tipo de herança *localReplacement* é a seguinte:

- *localReplacement:* este mecanismo deve ser sempre utilizado em conjunto com o mecanismo de extensão (*extension*). Ele permite que substituições locais sejam feitas em atividades que estão sendo herdadas através do mecanismo de extensão. Assim, toda atividade que aponta para outra atividade, utilizando-se do valor *localReplacement* não pode estar vazia. Essa atividade deve possuir, no mínimo, um elemento de processo instanciado como parte dessa atividade. Por exemplo, uma atividade A poderia herdar toda estrutura da atividade B pelo mecanismo de *extension*. Contudo, poderia ser necessário fazer substituições locais para partes da atividade A através da relação *localReplacement*. Para isso, devem existir subatividades dentro das atividades A e B. Na subatividade de A (por exemplo, A.1) deve ser definido o conteúdo local que vai substituir o conteúdo sendo herdado e, neste momento, deve ser apontada a subatividade de A para subatividade de B (por exemplo B.1). O resultado em A.1 substituirá todo conteúdo de B.1. Considerar que durante a interpretação do mecanismo *localReplacement* para atividades que estão em processos distintos ,uma análise de impacto deverá ser realizada

para todos os elementos no processo resultante. A análise de impacto mapeará todas as dependências dos elementos excluídos e indicará se novas exclusões são necessárias.

Uma diferença existe para o funcionamento do tipo de herança *localReplacement* nas atividades de adaptação e definição de um processo de software. Essa diferença está relacionada com a opcionalidade dos elementos de processo *atividade*, *tarefa*, *produto de trabalho* e *papel* e especifica que, para a atividade de definição de um processo de software (onde o *Process Package* possui o valor do atributo *isAuthoring* = *true*), durante a interpretação dos mecanismos de herança, não são consideradas as opcionalidades dos elementos de processo acima e, dessa forma, qualquer exclusão poderá ser feita através do mecanismo de substituição.

Já durante a atividade de adaptação (onde o *Process Package* possui o valor do atributo *isAuthoring* = *false*), as opcionalidades dos elementos de processo são consideradas e, dessa maneira, não é possível realizar substituições quando elementos de processo obrigatórios necessitam ser excluídos. Especificamentedurante a adaptação de um processo, para os casos onde as substituições são propostas para elementos obrigatórios ou, ainda, quando a substituição de qualquer elemento opcional causar a exclusão de um ou mais elementos de processo obrigatórios, a operação de substituição deverá ser cancelada.

Além das alterações realizadas especificamente sobre o relacionamento *usedActivity,* algumas definições também foram modificadas no relacionamento *supressedBreakdownElement,* o qual se estabelece entre as metaclasses *Activity* e *BreakdownElement* e, como já explicado na Seção 2.6.1.5 deste trabalho, só pode ser utilizado em conjunto com o relacionamento *usedActivity*.

Originalmente, no metamodelo SPEM 2.0, o relacionamento *supressedBreakdownElement* permite que todas as atividades que herdam conteúdos de outras atividades pelo tipo de herança *extension* possam ter qualquer elemento excluído. Dessa forma, toda vez que o relacionamento *extension* é interpretado, a relação *supressedBreakdownElement* é liberada, permitindo a exclusão de elementos do conteúdo herdado. Contudo, na especificação original do metamodelo SPEM 2.0, assim como para o tipo de herança *localReplacement*, não existem referências sobre o impacto das exclusões realizadas pelo relacionamento *supressedBreakdownElement* sobre outros elementos e relacionamentos, o que pode gerar várias inconsistências no processo resultante.

Na tentativa de evitar inconsistências em um processo de software causadas pelas exclusões do relacionamento *supressedBreakdownElement,* uma análise de impacto foi incluída para quando esse tipo de relação for utilizada. Basicamente, o funcionamento da análise de impacto proposta para o relacionamento *supressedBreakdownElement* é o mesmo daquela já explicada para o tipo de herança *localReplacement,* como mostra o exemplo a seguir.

O exemplo proposto para demonstrar o funcionamento da relação *supressedBreakdownElement* foi realizado sobre a *Atividade Processo 2* exibido na Figura 52. Essa atividade foi escolhida por já ter utilizado o tipo de herança *extension* e, desse modo, possuir o uso do relacionamento *supressedBreakdownElement* liberado. A proposta do exemplo foi excluir o *Produto de Trabalho Interno W* e a *Tarefa 5* da *Atividade Processo 2*, os quais são mostrados em negrito na Figura 50.

É possível observar na Figura 50 que o *Produto de Trabalho Interno W* é um produto de trabalho central na *Atividade 2.1* e que sua exclusão causa impactos nessa atividade, já que ele é o único produto de trabalho produzido pela *Tarefa 4*. Já observando a *Tarefa 5* da *Atividade 2.2,* nota-se que sua exclusão não causa nenhum impacto aos outros elementos da *Atividade Processo 2*, uma vez que essa tarefa não possui relação com outros elementos (o que representa, inclusive, uma inconsistência).

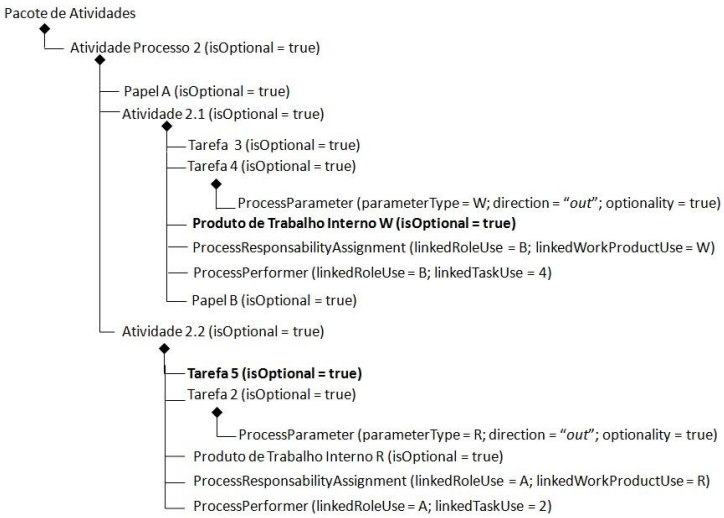

Figura 50 – Exemplo de aplicação do relacionamento *supressedBreakdownElement*

Para verificar todos os elementos afetados pela exclusão do *Produto de Trabalho Interno W* e da *Tarefa 5,* a análise de impacto foi realizada e o resultado é mostrado na Figura 51.

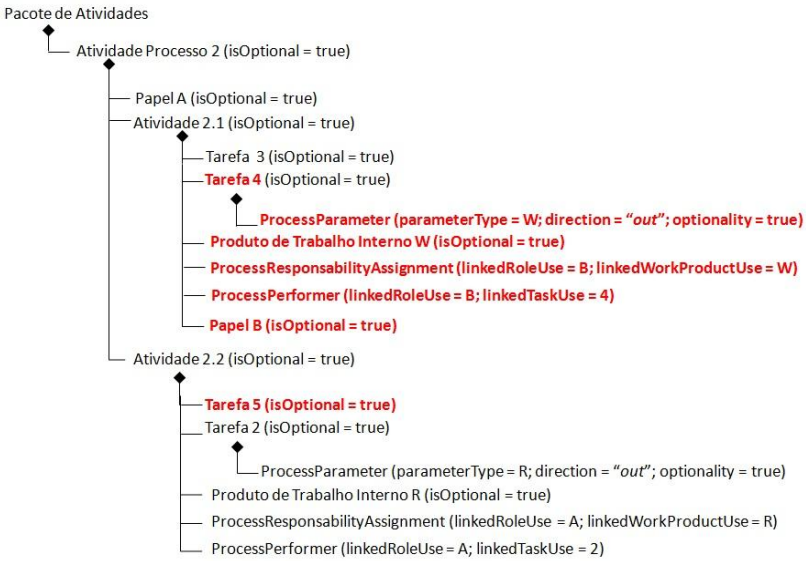

Figura 51 – Análise de impacto resultante da aplicação do relacionamento *supressedBreakdownElement*

No resultado mostrado na Figura 51, todos os elementos destacados em vermelho são os elementos que necessitam ser excluídos juntamente com o *Produto de Trabalho Interno W* e a *Tarefa 5.* Para chegar a tal resultado foi necessário analisar todos os relacionamentos de cada elemento excluído com objetivo de verificar a necessidade de novas exclusões. Como forma de entender como a análise de impacto é realizada, a Tabela 15 mostra os elementos impactados quando um elemento do tipo produto de trabalho interno é excluído (no exemplo, o *Produto de Trabalho Interno W*).

Tabela 15 – Análise de impacto para a exclusão de uma instância do elemento *InternalUse*

Elemento do Processo	**Impacto**
Metaclasse *Activity*	- A instância de *InternalUse* excluída deve ter suas associações com as instâncias de *Activity* eliminadas.
Metaclasse *TaskUse*	**-**
Metaclasse *RoleUse*	**-**
Metaclasse *InternalUse*	- Se a instância de *InternalUse* excluída está associada ao atributo *target* de uma ou mais instâncias de *WorkProductUseRelationship* com valor de atributo *relationType* igual a *dependency,* então a instância de *InternalUse* associada no atributo *source* da(s) instância(s) *WorkProductUseRelationship* deve(m) ser excluída(s). - Se a instância de *InternalUse* excluída for o único elemento associado no atributo *target* de uma instância de *WorkProductUseRelationship* com valor de atributo *relationType* igual a *composition*, então a instância de *InternalUse* associada no atributo *source* da(s) instância(s) *WorkProductUseRelationship* deve(m) ser excluída(s). - Se a instância de *InternalUse* excluída está associada

	ao atributo *source* de uma ou mais instâncias de *WorkProductUseRelationship* com valor de atributo *relationType* igual a *composition,então todas as instâncias de InternalUse associadas no atributo target da(s) instância(s) WorkProductUseRelationship deve(m) ser excluída(s).*
Metaclasse *ExternalUse*	- Se a instância de *InternalUse* excluída está associadoa ao atributo *target* de uma ou mais instâncias de *WorkProductUseRelationship* com valor de atributo *relationType* igual a *dependency,* então a instância de *ExternalUse* associada no atributo *source* da(s) instância(s) *WorkProductUseRelationship* deve(m) ser excluída(s). - Se a instância de *InternalUse* excluída for o único elemento associado no atributo *target* de uma de *WorkProductUseRelationship* com valor de atributo *relationType* igual a *composition*, então a instância de *ExternalUse* associada no atributo *source* da(s) instância(s) *WorkProductUseRelationship* deve(m) ser excluída(s). - Se a instância de *InternalUse* excluída está associada ao atributo *source* de uma ou mais instâncias de *WorkProductUseRelationship* com valor de atributo *relationType* igual a composition*, então todas as instâncias de ExternalUse associadas no atributo target da(s) instância(s) WorkProductUseRelationship deve(m) ser excluída(s).*
Metaclasse *ProcessParameter*	- Todas as instâncias de *ProcessParameter* que estão associadas com a instância de *InternalUse* excluída devem também ser excluídas.
Metaclasse *ProcessPerformer*	**-**
Metaclasse *WorkSequence*	**-**
Metaclasse *WorkProductRelationshipUse*	*- A instância de InternalUse* excluída deve ter suas associações com as instâncias de *WorkProductRelationshipUse* eliminadas. Se alguma das instâncias de *WorkProductRelationshipUse* afetadas não possuir mais instâncias de *InternalUse* e/ou *ExternalUse* associadas aos seus atributos *source* e *target,* então estas instâncias deverão também ser excluídas.
Metaclasse *ProcessResponsabilityAssingment*	- Todas as instâncias de *ProcessResponsabilityAssingment* que estão associadas com a instância de *InternalUse* excluída devem também ser excluídas.

Um ponto a ser considerado na análise de impacto mostrada na Tabela 15 é que, assim como o tipo de herança *localReplacement,* o relacionamento *supressedBreadkdownElement* considera as informações de opcionalidade diferentes para as atividades de adaptação (onde o *Process Package* possui o valor do atributo *isAuthoring = false*) e definição (onde o *Process Package* possui o valor do atributo

isAuthoring = *true*) de processos de software. Para esse tipo de relacionamento, igualmente ao tipo de herança *localReplacement,* foi estabelecido que, durante a definição de um processo de software, as informações de opcionalidade não são consideradas e qualquer elemento pode ser excluído. Já durante as atividades adaptação, uma vez que as opcionalidades dos elementos de processo são consideradas, não é possível realizar exclusões de elementos de processo obrigatórios.

Sendo assim, definiu-se que, durante as atividades de adaptação, qualquer operação de exclusão (realizadas pelo *supressedBreakdownElement*) ou de substituição (realizadas pelo *localReplacement*) que indicar como resultado da análise de impacto a exclusão de um ou mais elementos de processo (atividade, tarefa, papel e produtos de trabalho do tipo interno ou externo) obrigatórios deverá ser cancelada.

Uma descrição completa para a análise de impacto de cada um dos elementos excluídos em uma atividade pode ser vista no Apêndice C. Por fim, é importante citar que toda a análise de impacto mostrada na Tabela 15 e Apêndice C, a qual é definida para o relacionamento *supressedBreadkdownElement,* é a mesma utilizada no tipo de heranaça *localReplacement*.

4.4.2 Metaclasse Variability

O mecanismo de adaptação *Variability* é definido no pacote *Method Plugin* e possui muitas similaridades com o mecanismo *usedActivity*. As diferenças entre estes dois mecanismos, é que o *Variability* permite definir variantes do conteúdo do processo (elementos do *Method Content*) e das atividades de um processo de software (*Activity*) e não é considerado somente como um mecanismo de herança, tal como é o caso do *usedActivity*.

Outra diferença fundamental entre os mecanimos *usedActivity* e *Variablity* é que este último pode ser utilizado em todos os elementos do repositório do processo e também nos elementos do tipo atividade de um processo. Além disso, para utilizar o mecanismo *Variability* não existe necessidade de combinações entre tipos de herança.

Embora existam algumas diferenças conceituais e de funcionamento entre os mecanismos *usedActivity* e *Variability*, as modificações realizadas para o mecanismo *Variability* são praticamente as mesmas feitas para o mecanismo *usedActivity*. A primeira delas, por exemplo, foi a definição da regra de boa-formação que estabelece que uma

instância de um elemento do repositório ou uma atividade do processo não pode definir um autorrelacionamento do tipo *Variablity*. Isso evita que esses elementos apontem para sua própria instância e tentem variar (herdar, contribuir ou substituir) seu próprio conteúdo.

Regra #47 – Elementos do repositório de conteúdo (Method Content) e atividades (Activity) não podem variar seu próprio conteúdo.

Semântica: Nenhuma instância da metaclasse *MethodContent* ou da metaclasse *Activity* deverá estabelecer um autorrelacionamento *VariabilityBasedOnElement*.

Após a definição da regra de boa-formação acima, foram realizadas modificações nas semânticas dos tipos de herança *Contributes, Replaces* e *Extends-replaces* para estabelecer que os elementos que realizam esses tipos de apontamento não podem estar vazios, ou seja, eles necessitam ter algum valor de atributo preenchido e/ou associações com outros elementos. A motivação para a mudança da semâncica destes tipos de herança é que, como já explicado para o mecanismo *usedActivity,* no caso de uma contribuição (*Contributes*) algum novo valor de atributo ou alguma nova associação deve existir no elemento de origem (*variabilitySpecialElement*) para que realmente exista uma contribuição quando o conteúdo da elemento de destino (*variabilityBasedOnElement)* for herdado. Já no caso de uma substituição (*Replaces*) total ou uma substituição parcial (*Extends-replaces*) o elemento de origem (*variabilitySpecialElement*) não pode estar vazio, uma vez que seu conteúdo substituirá total ou parcialmente o conteúdo do elemento de destino (*variabilityBasedOnElement)*.

Outra modificação realizada para todos os tipos de herança foi realizada para o momento em que qualquer um desses tipos de herança é aplicado para o elemento atividade em um processo de software. Essa modificação estabelece que durante uma herança, todos os elementos utilizados, através de relacionamentos pela atividade que está sendo herdada e estão definidos em suas atividades superiores (atividades pai) ou estão definidos nos pacotes de elementos e relacionamentos, são considerados como parte dessa atividade e, desse modo, são herdados.

Por fim, a última modificação realizada foi feita para a semântica dos tipos de herança que envolvem mecanismos de substituição, como é o caso das heranças *Replaces* e *Extends-Replaces*. A alteração proposta foi definida apenas para quando o elemento *atividade* utiliza esses tipos de herança e estabele que uma análise de impacto deverá ser realizada em todo processo durante qualquer mecanismo de substituição.

Toda análise de impacto proposta para esses tipos de herança funciona exatamente da mesma forma que a análise de impacto explicada na Seção 4.4.1 e Apêndice C deste trabalho.

Nesse sentido, com base em todas as modificações descritas acima, as novas semânticas para os tipos de herança *Contributes*, *Replaces, Extends* e *Extends-replaces* são:

- *Contributes*: este mecanismo permite que contribuições sejam realizadas em um elemento do repositório ou uma atividade do processo sem modificar seu conteúdo existente. Assim, todo elemento que aponta para outro elemento, utilizando-se do valor *Contributes* não pode estar vazio, ou seja, deve ter algum atributo (do tipo *String*) preenchido e/ou possuir pelo menos uma instância de elemento associada em um de seus relacionamentos (*incoming* e/ou *outcoming*) do tipo 0..*. As propriedades que utilizam o mecanismo de contribuição são: valores de atributos e instâncias de associação. O efeito, após a interpretação do mecanismo *Contributes,* é que o elemento base é substituído com uma visão variante aumentada do elemento que combina os valores dos atributos e instâncias de associação. A forma como essa combinação é realizada depende do tipo do atributo ou associação (veja Tabela 2). Atributos, por exemplo, do tipo *String* de um elemento são concatenados (como, por exemplo, contribuir com a descrição de uma atividade). Ainda, elementos adicionais são adicionados em uma associação 0..* (como, por exemplo, adicionar um elemento do tipo *Guidance* em uma Tarefa ou Atividade). Considerar que, quando o mecanismo *Contributes* está sendo realizado entre instâncias do elemento de processo *Activity,* todos os elementos que são utilizados através de relacionamentos pela atividade que está sendo herdada (*variabilityBasedOnElement*) e, que estão definidos em suas atividades superiores (atividades pai) ou estão definidos nos pacotes de elementos e relacionamentos, são considerados como parte dessa atividade e, por isso, são herdados para a nova atividade (*variabilitySpecialElement*).

- *Replaces*: este mecanismo permite que um elemento do repositório ou o conteúdo de uma atividade do processo sejam substituídos. Assim, todo elemento que aponta para outro elemento utilizando-se do valor *Replaces* não pode estar vazio, ou seja, ele deve ter algum atributo preenchido e/ou possuir pelo menos uma instância de elemento associada em qualquer um de seus relacionamentos (*incoming* e/ou *outcoming*). As propriedades que utilizam o mecanismo de substituição são: valores de atributos e instâncias de associação. O mecanismo *Replaces* permite, por exemplo, que seja alterado o papel

responsável por um produto de trabalho ou, ainda, que seja alterada a descrição de uma tarefa. Considerar que durante a interpretação do mecanismo *Replaces* para atividades que estão em processos distintos, uma análise de impacto deverá ser realizada para todos os elementos no processo resultante. A análise de impacto deverá mapear todas as dependências dos elementos excluídos e indicar se novas exclusões são necessárias.

- *Extends*: este mecanismo permite reutilizar elementos do repositório ou o conteúdo de uma atividade do processo, proporcionando uma espécie de herança. As propriedades que utilizam o mecanismo de extensão são: valores de atributos e instâncias de associação. O resultado desse mecanismo é que o elemento que estende outro elemento tem as mesmas propriedades que o elemento estendido, mas pode sobrepor os valores destas propriedades com seus próprios valores, o que é uma variante do conteúdo já definido. Sendo assim, o mecanimos *Extends* permite, por exemplo, que uma versão especial de um Registro de Revisão genérico seja definida para um tipo específico de revisão. Torna-se necessário considerar que, quando o mecanismo *Extends* está sendo realizado entre instâncias do elemento de processo *Activity,* todos os elementos que são utilizados através de relacionamentos pela atividade que está sendo herdada (*variabilityBasedOnElement*) e estão definidos em suas atividades superiores (atividades pai) ou estão definidos nos pacotes de elementos e relacionamentos são considerados como parte desta atividade e, dessa maneira, são herdados para a nova atividade (*variabilitySpecialElement*).

- *Extends-replaces*: este mecanismo combina os efeitos do mecanismo de extensão e substituição. As propriedades que utilizam o mecanismo de extensão-substituição são: valores de atributos e instâncias de associação. A diferença deste mecanismo para o de substituição é que esse último substitui completamente todos os atributos e instâncias de associação do elemento original com novos valores e instâncias de associação, removendo todos os valores e instâncias de associação do elemento original que não sofreram substituição. Já o mecanismo ora apresentado permite substituir apenas partes do elemento original, alterando apenas alguns valores de atributos e instâncias de associação. No mecanismo *Extends-replaces*, os valores de atributos e instâncias de associação que não foram substituídos permanecem com os valores do elemento original. Assim, para que realmente exista uma substuição parcial, o elemento que aponta para outro elemento, utilizando-se do valor *Extends-Replaces,* não pode estar vazio, ou seja, ele deve ter algum atributo preenchido e/ou possuir pelo menos uma instância de elemento associada em qualquer um de seus relacionamentos *outcoming* ou pelo menos

uma instância de elemento associada em um de seus relacionamentos *incoming* do tipo 0..1. Considerar que quando o mecanismo *Extends-replaces* está sendo realizado entre instâncias do elemento de processo *Activity,* todos os elementos que são utilizados através de relacionamentos pela atividade que está sendo herdada (*variabilityBasedOnElement*) e estão definidos em suas atividades superiores (atividades pai) ou estão definidos nos pacotes de elementos e relacionamentos são considerados como parte desta atividade e, por isso, são herdados para a nova atividade (*variabilitySpecialElement*). Considerar também que, durante a interpretação do mecanismo *Extends-Replaces* para atividades que estão em processos distintos, uma análise de impacto deverá ser realizada para todos os elementos no processo resultante. A análise de impacto deverá mapear todas as dependências dos elementos excluídos e indicar se novas exclusões são necessárias.

4.5 Pontos de Conformidade

Uma vez que esta pesquisa estendeu o metamodelo original SPEM 2.0, utilizando os princípios de empacotamento definidos na UML, existe a necessidade de especificar o(s) ponto(s) de conformidade para o metamodelo sSPEM 2.0. Dessa forma, o ponto de conformidade estabelecido é descrito a seguir:

- ***sSpem Complete****,* que define a utilização de todos os pacotes do metamodelo. Este nível estabelece a implementação de todo o metamodelo, focando na construção de processos de software completos e reutilizáveis.

Embora, nesta pesquisa, não tenham sido realizadas modificações nos pacotes *Managed Content* e *Process Behavior,* eles foram mantidos no metamodelo sSPEM 2.0 e, dessa forma, considera-se que todos os pacotes podem ser utilizados na definição de um processo de software, fato que levou a especificação do ponto de conformidade *sSpem Complete*.

4.6 Considerações Finais

Este capítulo apresentou uma extensão ao metamodelo SPEM 2.0 denominada sSPEM 2.0 (*conSistent* SPEM 2.0). Durante a descrição do capítulo, todas as alterações em termos de elementos e relacionamentos, bem como as regras de boa-formação para

consistência definidas para o metamodelo sSPEM 2.0 foram descritas. Em adição, foi apresentado também neste capítulo, os mecanismos de adaptação *usedActivity*, *supresssedBreakdownElement* e *Varialibity*.

O próximo capítulo desta tese apresenta um guia para construção e adaptação dos processos de software. O guia especifica o fluxo de atividades a ser seguido para a definição e adaptação de processos utilizando o metamodelo sSPEM 2.0 e as regras de boa-formação, os quais integram a infraestrututra para consistência apresentada neste trabalho.

5 GUIA PARA DEFINIÇÃO E ADAPTAÇÃO DE PROCESSOS DE SOFTWARE

Este capítulo apresenta um guia que especifica o fluxo de atividades a ser seguido para a definição e adaptação de processos, utilizando a infraestrututra para consistência, apresentada neste trabalho. Para representar o fluxo de atividades do guia proposto, a Figura 52 mostra um diagrama que é composto por 4 etapas: (1) a definição de uma biblioteca que armazena conteúdo e processos; (2) a definição do repositório de conteúdo; (3) a definição dos processos de software; e (4) a adaptação dos processos de software. É necessário considerar que todas as etapas mostradas na Figura 52 devem ser realizadas utilizando o metamodelo sSPEM 2.0, sendo as etapas 1, 2 e 3 executas pelo papel Engenheiro de Processo e a etapa 4 executada pelo papel Gerente de Projeto.

A seguir, apresenta-se um detalhamento das etapas para definição e adaptação de processos de software:

5.1 Definição da Biblioteca – Etapa 1

A primeira atividade que deve ser realizada antes da definição de repositórios de conteúdo e processos é a definição de uma biblioteca. Basicamente, a criação de uma biblioteca é bastante simples e envolve apenas dois passos. Inicialmente, para representação da biblioteca, uma instância da metaclasse *Method Library* deverá ser criada e nomeada. Em seguida, uma instância da metaclasse *Method Plugin* deverá ser definida como parte da instância de *Method Library*. Um *Method Plugin* é sempre criado para uma biblioteca, pois este elemento representa no metamodelo sSPEM 2.0 um contêiner físico que armazena todo conteúdo do repositório (*Method Content)* e os processos de software (*Process Structure*).

5.2 Definição do Repositório de Conteúdo – Etapa 2

Inicialmente, para definir um repositório de conteúdo, uma instância da metaclasse *MethodContent* deverá ser criada como parte de uma instância da metaclasse *MethodPlugin*. Em seguida, conforme necessário, instâncias das metaclasses que representam os pacotes de elementos e relacionamentos podem ser criadas. Tais metaclasses são: *RolePackage*, *WorkProductDefinitionPackage*, *TaskPackage*, *ToolPackage* e *MethodContentRelationshipPackage*.

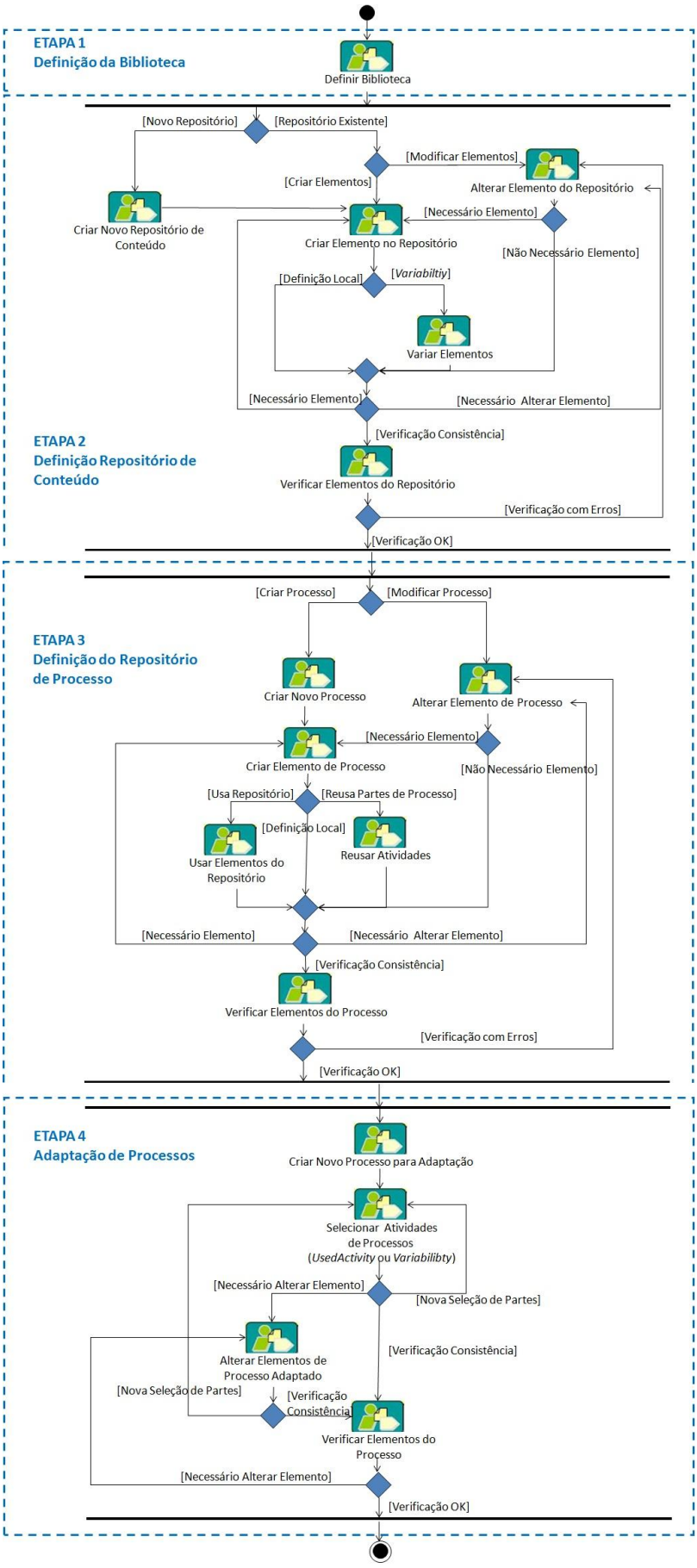

Figura 52 – Diagrama que representa o fluxo de atividades para definição e adaptação de processos

A partir da definição dos pacotes acima, qualquer instância de elemento e relacionamento pode ser criada nos seus respectivos pacotes (atividade *Criar Elemento no Repositório*). Identifica-se apenas que, durante a criação destes elementos e relacionamentos, as regras de boa-formação definidas para o pacote *Method Content* (conforme Seção 4.3.2.2) deverão ser respeitadas. Durante a atividade de criação de um elemento, é possível que as informações sejam definidas localmente (informando as informações do elemento diretamente nos seus atributos) para este elemento ou ainda, que o mecanismo *Variability* seja utilizado através da atividade *Variar Elementos*. Neste último caso, para execução da atividade *Variar Elementos,* deve-se relacionar o elemento criado através do seu atributo *variabilityBasedOnElement* com o elemento que se deseja criar uma variação, estabelecendo o tipo de herança através do atributo *variabilityType* (que possui os valores *Replaces*, *Contributes*, *Extends* e *Extends-Replaces*) (conforme Seção 4.4.2).

Após definir cada elemento do repositório, o Engenheiro de Processo deverá verificar a necessidade de criação de novos elementos (atividade *Criar Elemento no Repositório*) ou alteração de elementos existentes deste repositório (atividade *Alterar Elemento do Repositório*). Caso nenhuma destas atividades sejam necessárias, a atividade *Verificar Elementos do Repositório* deverá ser executada.

O objetivo desta última atividade é checar a consistência dos elementos do repositório, utilizando para isto, as regras de boa-formação para consistência definidas nesta pesquisa para a atividade de definição de processos. O funcionamento da atividade *Verificar Elementos do Repositório* envolve inicialmente, a seleção do elemento que deve ser verificado para que, a partir disto, todos elementos que estão estruturalmente dentro (em outras palavras, elementos aninhados pela relação de composição) do elemento selecionado sejam verificados. Assim, caso o Engenheiro de Processo selecione a instância da metaclasse *MethodContent* para verificação, todo o conteúdo do repositório será validado e o resultado desta validação conterá todas as suas inconsistências, caso elas existam.

A última atividade da etapa 1 envolve a alteração dos elementos do repositório, caso a atividade de verificação retorne erros. A ideia é que a atividade *Alterar Elementos de Repositório* seja executada até que todas as inconsistências sejam eliminadas do repositório.

5.3 Definição do Repositório de Processos – Etapa 3

Uma vez que o repositório de conteúdos foi criado, a criação de um novo processo pode ser iniciada (atividade *Criar Novo Processo*). Para fazer isto, o Engenheiro de Processo deverá definir inicialmente, uma instância da metaclasse *ProcessPackage* (com valor de atributo *isAuthoring = true*) como parte da instância de *MethodPlugin.* Em seguida, para criação de um novo processo devem ser definidas, respectivamente, uma instância da metaclasse *ActivityPackage* e uma instância da metaclasse *Activity*. O estabelecimento de uma instância de *ActivityPackage* é obrigatória pois nenhum elemento pode ser criado fora dos pacotes. Já o estabelecimento da instância de *Activity* se torna necessária pois, tanto no metamodelo original SPEM 2.0 quanto na solução proposta por esta pesquisa (sSPEM 2.0), este é o elemento que deverá representar o processo de software sendo criado.

Prosseguindo com a atividade de criação de um processo de software, assim como em um repositório de conteúdo, conforme a necessidade, instâncias das metaclasses que representam os pacotes de elementos e relacionamentos do processo podem ser criadas. Em um *ProcessPackage* tais metaclasses são: *WorkProductUsePackage*, *RoleUsePackage*, *TaskUsePackage* e *ProcessRelationshipPackage*.

Após a criação dos pacotes, os elementos e relacionamentos podem começar a ser definidos para o processo em questão através da atividade *Criar Elemento do Processo*. Para realizar esta atividade basta criar uma instância do elemento desejado. Contudo, durante esta criação algumas regras estabelecidas nesta pesquisa necessitam ser respeitadas.

As regras definidas são relacionadas com o local onde os elementos e relacionamentos devem ser criados para um processo de software e foram estabelecidas, pois, como já explicado na Seção 4.3.3.1 deste trabalho, todos elementos criados nos pacotes são considerados como elementos globais, ou seja, podem ser utilizados por qualquer atividade do processo. Além disso, as relações que são definidas nos pacotes de relacionamento podem estar associadas a elementos que estão em qualquer atividade do processo. Já os elementos que são criados dentro de uma atividade em específico só podem ser utilizados por elementos e relacionamentos que estejam definidos em uma de suas sub-atividades.

Baseado no contexto acima, inicialmente, considerando os relacionamentos, ficou estabelecido que qualquer instância de *WorkSequence* deverá ser definida dentro do pacote de relacionamentos. Isso se torna necessário, uma vez que este tipo de relacionamento deverá ter acesso a todas as atividades e tarefas de um processo de software.

Seguindo com as definições sobre o local de criação dos relacionamentos, recomenda-se que as relações de responsabilidade sobre os produtos de trabalho (*ProcessResponsabilityAssigment*) e os relacionamentos estabelecidos entre estes mesmos elementos (*WorkProductUseRelationship*) sejam também criados nos pacotes de relacionamentos. Embora isso não seja obrigatório, faz-se esta recomendação por tratarem-se de relacionamentos que valem para o processo todo e que em muitas situações só podem mesmo serem criados nos pacotes de relacionamentos, como é o caso onde os elementos envolvidos em alguma destas relações estejam definidos em atividades diferentes.

Por fim, ainda referente a definição de relacionamentos, apenas como recomendação estabelece-se que toda instância de *ProcessPerformer* deverá ser criada no mesmo local onde está localizada a tarefa associada a esta relação.

As últimas regras relacionadas com o local de criação dos elementos e relacionamentos, estabelecem que os nodos iniciais e finais devem ser criados da seguinte forma: quando o sequenciamento é estabelecido entre tarefas e os nodos especiais são instâncias destes elementos eles deverão ser definidos nos pacotes de tarefas. Já quando o sequenciamento é feito entre as atividades de um processo os nodos iniciais e finais devem ser criados no nível mais superior do processo, ou seja, diretamente dentro da atividade que representa tal processo. A motivação para definir em qual local os nodos iniciais e finais devem ser criados em um processo de software, foi a preservação destes elementos quando uma operação de substituição e/ou exclusão é realizada nesse processo.

Após a definição de uma instância de elemento para um processo de software, a definição de seu conteúdo pode ser feita de duas maneiras, conforme definição original do metamodelo SPEM 2.0. A primeira forma diz respeito a definição local do conteúdo do elemento criado, ou seja, uma instância do tipo *Use* é criada para um elemento ou relacionamento e suas informações são incluidas diretamente nos seus atributos. A segunda forma, a qual é indicada pelo metamodelo original SPEM 2.0 como sendo a principal maneira de definição de um processo, é a utilização do conteúdo estabelecido no

repositório de conteúdos. Para fazer uso deste repositório, a atividade *Usar Elementos do Repositório* deverá ser executada. Esta atividade consiste em apontar a instância (*Use)* do elemento de processo criado (*papel*, *produto de trabalho* ou *tarefa)* através de um atributo (*role*, *workProduct* ou *task*) para um elemento do mesmo tipo no repositório. Neste momento, todo conteúdo definido para o elemento que sofre o apontamento e também alguns de seus elementos relacionados são copiados para o processo. Toda explicação sobre a cópia de elementos do repositório para um processo de sofware é realizada na Seção 5.5 deste capítulo.

Outra forma de definir conteúdo para um processo de software é através da execução da atividade *Reusar Atividades* que envolve o uso dos mecanismos *Variability* e *usedActivity*, os quais permitem que partes do mesmo processo ou ainda, de processos diferentes sejam herdados. No contexto de um processo de software, a atividade *Reusar Atividades* só pode ser executada se o elemento definido na sua atividade anterior (*Criar Elemento do Processo*) for uma instância do elemento *atividade*. Isso porque, *atividades* são os únicos elementos em um processo que podem utilizar os mecanismos *Variability* e *usedActivity.* Para executar a atividade *Reusar Atividades* com o mecanismo *Variability* é necessário apontar a instância da *atividade* criada através de seu atributo *variabilityBasedOn* para a *atividade* que se deseja criar uma variação estabelecendo o tipo de herança através do atributo *variabilityType* (que possui os valores *Replaces*, *Contributes*, *Extends* e *Extends-Replaces*). Já para executar a atividade *Reusar Atividades* com o mecanismo *usedActivity* é necessário apontar a instância da *atividade* criada através de seu atributo *usedActivity* para a *atividade* que se deseja herdar o conteúdo estabelecendo o tipo de herança *extension* através do atributo *useKind*. Após a execução do mecanismo *usedActivity*, o relacionamento *supressedBreakdownElement* torna-se disponível para realizar exclusões no conteúdo herdado através do atributo *suppressedBreakdownElement* do elemento *atividade* (conforme Seção 4.4.1).

Após definir um elemento do processo, o Engenheiro de Processo deverá verificar a necessidade de criação de novos elementos (atividade *Criar Elemento de Processo*) ou alteração de elementos existentes deste processo (atividade *Alterar Elemento do Processo*). Caso nenhuma destas atividades sejam necessárias, a atividade *Verificar Elementos do Processo* deverá ser executada.

O objetivo desta última atividade é checar a consistência dos elementos do processo, utilizando para isto, as regras de boa-formação para consistência definidas nesta pesquisa para a atividade de definição de processos. O funcionamento da atividade

Verificar Elementos do Processo envolve inicialmente, a seleção do elemento que deve ser verificado para que, a partir disto, todos elementos que estão estruturalmente dentro (em outras palavras, elementos aninhados pela relação de composição) do elemento selecionado sejam verificados. Assim, caso o Engenheiro de Processo selecione a instância da metaclasse *ProcessPackage* para verificação, todo o conteúdo do processo será validado e o resultado desta validação conterá todas as suas inconsistências, caso elas existam.

A última atividade desta etapa envolve a alteração dos elementos do processo, caso a atividade de verificação retorne erros. A ideia é que a atividade *Alterar Elementos de Processo* seja executada até que todas as inconsistências sejam eliminadas do processo de software.

5.4 Adaptação de Processos – Etapa 4

Uma vez que pelo menos um processo está disponível no repositório de processos, as atividades de adaptação podem começar a ser realizadas pelo Gerente de Projeto. Basicamente, as atividades de adaptação propostas pelo metamodelo original SPEM 2.0 e, consequentemente, por esta pesquisa (sSPEM 2.0), são baseadas em reusar partes (*atividades*) de um ou mais processos de software para o processo que está sendo adaptado, permitindo que modificações sejam realizadas nestes conteúdos através dos mecanismos *usedActivity, supressedBreakdownElement* e *Variability.*

Para executar a primeira atividade de adaptação, um novo processo deve ser criado através da execução da atividade *Criar Novo Processo para Adaptação*. Esta atividade envolve, respectivamente, a criação de uma instância da metaclasse *ProcessPackage* (com valor de atributo *isAuthoring = false*) como parte da instância de *MethodPlugin*, uma instância da metaclasse *ActivityPackage* e uma instância da metaclasse *Activity*. Assim como na atividade de construção de processos, o estabelecimento de uma instância de *ActivityPackage* é obrigatória pois nenhum elemento pode ser criado fora dos pacotes. Já o estabelecimento da instância de *Activity* é necessário para representar o processo de software sendo criado.

Após a definição de um novo processo de software que irá gerar um processo específico através dos mecanimos de adaptação, a atividade *Selecionar Atividades de Processo* deverá ser executada. Esta atividade poderá ser executada basicamente de duas formas: (1) se é desejado que o processo criado herde todo o conteúdo de outro

processo a própria instância de *Activity* criada para representar o processo de software pode realizar o apontamento para outra *atividade* que representa outro processo de software utilizando os mecanismos *Variability* (utilizando os atributos *variabilityBasedOnElement* e *variabilityType*) ou *usedActivity* (utilizando os atributos *usedActivity* e *useKind*). No caso específico do *usedActivity,* conforme necessário, outras instâncias de atividades podem ser criadas para utilização dos tipos de herança adicionais *localContribution* e *localReplacement* (conforme Seção 4.4.1). Além disso, após qualquer atividade ser herdada através do mecanismo *usedActivity* será possível excluir qualquer um de seus elementos e relacionamentos (utilizando o atributo *supressedBreakdownElement*), uma vez que o relacionamento *supressedBreakdownElement* será liberado; e (2) se é desejado criar um processo reutilizando (através do *usedActivity*) ou criando variações (através do *Variability*) de apenas partes de um ou vários outros processos, é necessário que uma nova instância de *Activity* seja definida como parte da instância de *Activity* que representa o processo para que esta nova *Activity* realize o apontamento para a *atividade* desejada em outro processo de software utilizando os mecanismos *Variability* (utilizando os atributos *variabilityBasedOnElement* e *variabilityType*) e *usedActivity* (utilizando os atributos *usedActivity* e *useKind*). A ideia é que somente uma parte de outro processo de software seja herdada.

A atividade *Selecionar Atividades de Processo* pode ser executada várias vezes até que todo conteúdo necessário seja herdado para o processo sendo definido. É possível ainda que após a execução desta atividade alterações sejam realizadas no conteúdo herdado através da execução da atividade *Alterar Elementos do Processo Adaptado*. Considera-se, inclusive, que durante a execução desta última atividade elementos novos possam ser incluídos no processo de software. Salienta-se apenas que durante a criação ou alteração de qualquer elemento, assim como durante as atividades de definição de processos, todas as regras de boa-formação definidas para os pacotes *Process Structure* e *Process with Methods* devem ser respeitadas.

Após executar a atividade *Selecionar Atividades de Processo* e *Alterar Elementos do Processo Adaptado,* o Gerente de Projeto deverá executar a atividade *Verificar Elementos do Processo.*

O objetivo desta última atividade é checar a consistência dos elementos do processo, utilizando para isto, as regras de boa-formação para consistência definidas nesta pesquisa para a atividade de definição de processos. A atividade *Verificar*

Elementos do Processo tem o mesmo funcionamento desta atividade na Etapa 3 (conforme Seção 5.3).

A última atividade desta etapa envolve a alteração dos elementos do processo, caso a atividade de verificação retorne erros. A ideia é que a atividade *Alterar Elementos de Processo Adaptado* seja executada até que todas as inconsistências sejam eliminadas do processo de software.

5.5 Funcionamento do Repositório de Conteúdos (Method Content) baseado na Engenharia de Método

A forma definida, no metamodelo original SPEM 2.0, de utilização dos elementos do repositório de conteúdos (*Method Content)* em processos de software é realizada no pacote *Process with Meth*ods. Isso ocorre, pois, como já explicado na Seção 2.6.1.4, esse pacote inclui todo conteúdo dos pacotes *Process Stru*cture e *Method Content,* através do mecanismo de *merge,* e define localmente associações para que seja possível que um elemento de processo definido no *Process Structure* (tarefa, produto de trabalho ou papel) aponte para um elemento de definição do *Method Content,* conforme mostrado na Figura 53.

Assim, é possível, por exemplo, criar uma instância da metaclasse *WorkProductUse* (definida no *Process Structure*) que aponta para uma metaclasse *WorkProductDefinition* (definida no *Method Content*). Tal apontamento é realizado através da associação definida no pacote *Process with Methods* e indica que a instância de processo (instância do tipo *Use*) está associada à instância do repositório (instância do tipo *Definition*), fazendo com que, toda vez que o repositório seja atualizado, a instância de processo também sofra essas atualizações.

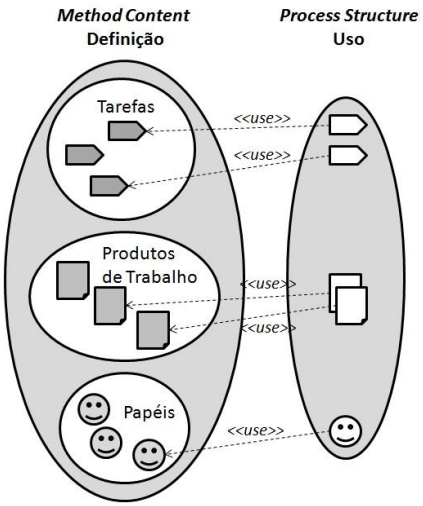

Figura 53 – Representação da associação do pacote *Process with Methods* que permite realizar os apontamentos do pacote *Process Structure* para o *Pacote Method Content*

A especificação textual do metamodelo original SPEM 2.0 não realiza um detalhamento sobre a forma como os elementos devem ser copiados do repositório para um processo de software. Nesse metamodelo, é apenas definido que quando um elemento de processo aponta para um elemento do repositório cópias congruentes dos elementos que se relacionam com este elemento devem ser fornecidas.

Basicamente, a proposta do metamodelo original SPEM 2.0 é modularizar conteúdos de processo em um repositório, permitindo assim a utilização destes módulos em diferentes processos de software. Esta ideia vai ao encontro da abordagem utilizada pelos autores da área de Engenharia de Método, uma vez que tais autores defendem a criação de fragmentos de método (processo e produto) armazenados em repositórios para reutilização em processos distintos.

Fazendo o mapeamento dos conceitos utilizados no repositório de conteúdos (*Method Content)* para os conceitos utilizados pelos autores da Engenharia de Método [Har94], [Rol98], [Ral04], pode-se considerar que as *tarefas* e *produtos de trabalho* criados no repositório de conteúdos do SPEM 2.0 representam, respectivamente, os fragmentos de processo e os fragmentos de produto de um processo de software. Ambos os tipos de fragmentos, irão se relacionar com o elemento de processo *papel* e, mais especificamente, os fragmentos de processo irão também se relacionar com os elementos do tipo *ferramenta*. Tais relacionamentos são do mesmo modo bastante comuns na área de Engenharia de Método.

Ainda que seja possível fazer o mapeamento dos conceitos relacionados com o repositório de conteúdos (*Method Content*) no metamodelo SPEM 2.0 para os conceitos na área de Engenharia de Método, algumas divergências podem ser encontradas quanto

ao funcionamento proposto pelo metamodelo SPEM 2.0 para este repositório. Inicialmente, diz-se isto, pois, de acordo com a especificação textual do SPEM 2.0 é possível criar um elemento do tipo *papel* no processo e fazer o apontamento para a definição de um *papel* no repositório. Isso dá a ideia de que este elemento representa um fragmento, o que não existe na Engenharia de Método, uma vez que, embora esses elementos estejam presentes em um repositório, não existem fragmentos específicos para eles. O que há são as relações dos fragmentos de processo e fragmentos de produto com esse elemento.

Outra diferença do metamodelo SPEM 2.0 relacionada com a área da Engenharia de Método é específica com a formação de um fragmento de produto no repositório. Faz-se tal assertiva, em função de que, embora não exista uma definição rígida para um fragmento de produto na Engenharia de Método, a visão da maioria dos autores, tais como Harmsen *et al.* em [Har94] e Ralyté em [Ral04], é a de que, mesmo que um fragmento de produto esteja associado a vários fragmentos de processo, no momento que este fragmento de produto é selecionado para um processo, somente os fragmentos de processo responsáveis por produzir tal fragmento são também selecionados. Isso funciona de forma diferente no metamodelo SPEM 2.0, uma vez que esse metamodelo estabelece que, toda vez que um produto de trabalho é criado no processo e aponta para a definição de um produto de trabalho (o que representa um fragmento de produto) no repositório, todas as relações deste produto de trabalho devem ser copiadas. Isso faz com que todas as tarefas (que representam os fragmentos de processo) relacionadas com esse produto de trabalho, incluindo aquelas que apenas o consomem e/ou o modificam, sejam também copiadas para o processo de software. Por exemplo, para a criação de um produto de trabalho no processo que aponte para o produto de trabalho *Visão* (considerando o processo RUP) no repositório, seria necessária a criação de muitas outras tarefas no processo, já que este é um produto de trabalho central (consumido e modificado em vários pontos do processo) para as tarefas de requisitos do processo RUP.

Baseado no exposto acima, considera-se que a proposta do metamodelo original SPEM 2.0 para funcionamento do repositório de conteúdo apresenta alguns problemas de coesão e acoplamento. Por esse motivo, esta pesquisa definiu um novo funcionamento do repositório de conteúdos para o metamodelo sSPEM 2.0.

Inicialmente, relacionado ao elemento papel definiu-se nesta pesquisa que este tipo de elemento não pode sofrer apontamentos diretos a partir de um processo de software. Isso significa que não é possível criar uma instância de papel no processo

(instância do tipo *Use*) e apontar para uma instância do repositório (instância do tipo *Definition*). A única forma de usar uma instância de papel do repositório em um processo de software será através do apontamento de suas tarefas (fragmentos de processo) e produtos de trabalho (fragmentos de produto) relacionados. No momento em que, por exemplo, uma tarefa do processo aponta para uma tarefa do repositório, os papéis relacionados (executores) com esta tarefa serão copiados para o processo.

Neste momento, instâncias de processo (instância do tipo *Use*) do elemento papel serão criadas e apontadas para as instâncias de definição desses elementos no repositório. Da mesma maneira, quando um produto de trabalho do processo aponta para um produto de trabalho do repositório, os papéis relacionados (responsáveis) com esse produto de trabalho serão também copiados para o processo de software, sendo realizados os devidos apontamentos das instâncias de processo para as instâncias do repositório.

Para o uso do elemento produto de trabalho do repositório em processos de software, várias definições foram realizadas com base na área de Engenharia de Método. Inicialmente, definiu-se que durante o apontamento entre produtos de trabalho (processo x repositório) devem ser copiadas para o processo apenas as tarefas que estão relacionadas com o produto de trabalho através de uma relação de produção, ou seja, serão copiadas somente as tarefas que produzem tal produto de trabalho. Em seguida, definiu-se que quando tais tarefas são copiadas, elas devem ser trazidas de forma completa, ou seja, para cada tarefa de processo que realiza apontamento para uma tarefa de repositório, devem ser copiadas todas suas relações e elementos destas relações. É necessário estabelecer que, quando produtos de trabalho do tipo externo (*ExternalUse*), os quais não são produzidos em um processo de software, realizam apontamento para um produto de trabalho do repositório, nenhuma tarefa é copiada para o processo.

Especificamente, considerando as relações que o produto de trabalho que sofre apontamento possui com outros produtos de trabalho no repositório, definiu-se que nem todas estas relações e elementos serão copiados. Considerando as três relações possíveis entre produtos de trabalho (*dependência, agregação* e *composição*) identificou-se que elas possuem funcionamentos distintos. Relacionado com a relação de dependência durante um apontamento entre produtos de trabalho (processo x repositório), definiu-se que serão copiadas para o processo somente as relações entre produtos de trabalho que estabelecem que o produto de trabalho que está sofrendo o apontamento depende de outro produto de trabalho.

Assim, fica estabelecido que, durante um apontamento entre produtos de trabalho, serão copiados para o processo o próprio produto de trabalho que sofre o apontamento, as relações de dependência e os produtos de trabalho dos quais este produto de trabalho depende. Nesse caso, é necessário considerar que, quando a relação de dependência estabelece que outros produtos de trabalho dependem do produto de trabalho que está sofrendo o apontamento, elas não serão copiadas, pois não são necessárias no processo de software.

Já com relação às relações de agregação entre produtos de trabalho, definiu-se que não é necessário copiá-las para um processo de software durante o apontamento entre produtos de trabalho. Por fim, considerando as relações de composição, definiu-se que, quando o produto de trabalho que está sofrendo o apontamento possui uma relação que estabelece que esse produto é *parte* de outro produto de trabalho tanto, a relação quanto o produto de trabalho que representa o *todo* devem ser copiados para o processo durante o apontamento.

Caso o produto de trabalho que está sofrendo o apontamento possua uma relação com outros produtos de trabalho que estabelecem que ele é o *todo* e os outros produtos de trabalho são suas *partes,* definiu-se que ao menos uma de suas partes deverão ser copiados para o processo de software. Tanto o funcionamento da relação de agregação quanto o funcionamento da relação de composição foram definidos visando respeitar as definições estabelecidas para estas relações na UML. Outra definição realizada é que todo produto de trabalho que é copiado para o processo copia também suas relações com os papéis do repositório, trazendo automaticamente esses elementos para o processo de software.

Para facilitar o entendimento do funcionamento do repositório de conteúdos (*Method Content)* definido nesta pesquisa, a Figura 54 mostra um exemplo.

O exemplo mostrado na Figura 54 apresenta os termos dos elementos e das relações do repositório de conteúdo (*Method Content*), bem como dos processos de software em inglês. Nessa figura, a parte central mostra o apontamento de três produtos de trabalho localizados em processos diferentes para o repositório de conteúdo (localizado na parte superior da figura). Em seguida, são apresentados na mesma figura (localizado na parte inferior), os resultados dos elementos copiados após os apontamentos.

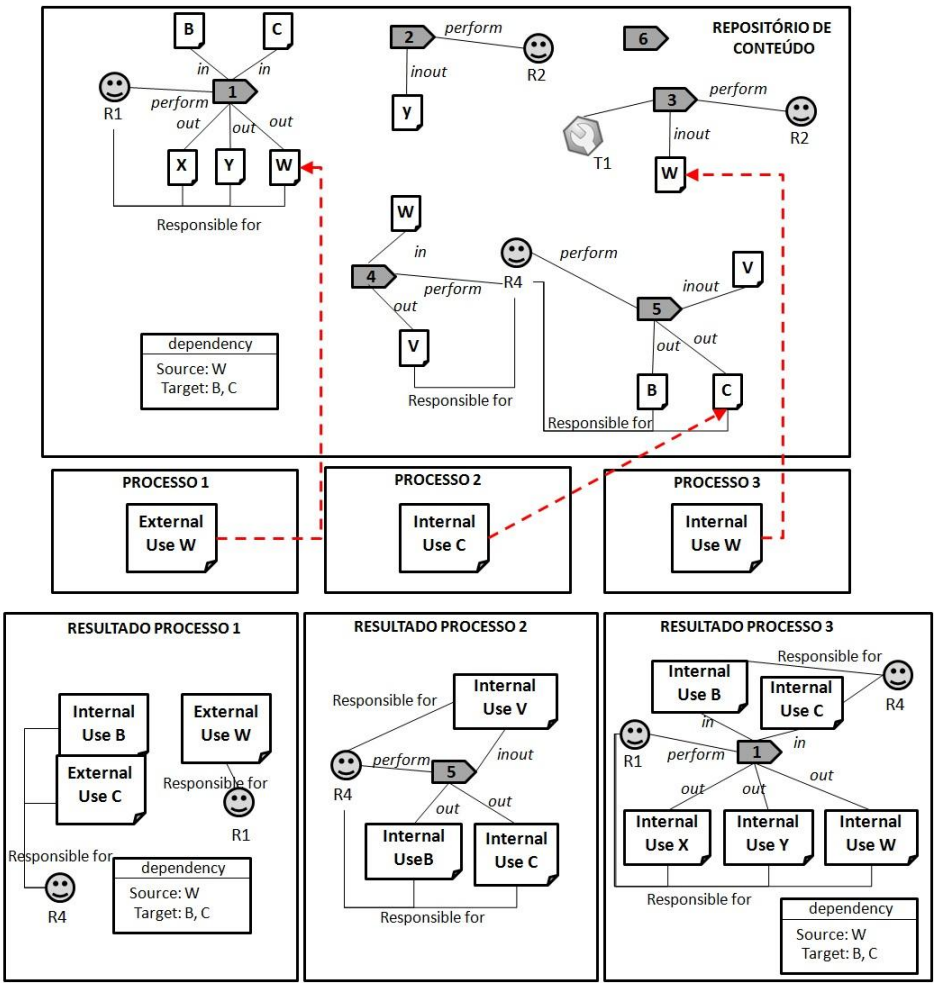

Figura 54 – Exemplo de funcionamento do *Method Content* proposto nesta pesquisa

Analisando, em detalhes, os apontamentos realizados a partir dos *Processo 1* e *3*, ambos realizados com o *produto de trabalho W*, observa-se que seus resultados são diferentes. Isso ocorre, porque, no apontamento realizado a partir do *Processo 1,* uma instância de produto de trabalho externo (*ExternalUse*) foi criada, enquanto que a instância de produto de trabalho que realiza o apontamento no *Processo 3* é do tipo interno (*InternalUse*). Os resultados do *Processo 1* mostram que os *produtos de trabalho B* e *C* também foram copiados, uma vez que o *produto de trabalho W* depende desses produtos de trabalho no repositório de conteúdo.

Além disso, para todos os produtos de trabalho copiados foram também trazidos do repositório suas relações com seus papéis responsáveis. Já analisando o resultado do *Processo 3,* verifica-se que, por se tratar de um produto de trabalho interno realizando o apontamento, sua tarefa de produção foi trazida do repositório juntamente com todas as suas entradas e saídas em termos de produtos de trabalho, bem como os papéis responsáveis por sua execução e também responsáveis pelos produtos de trabalho

copiados. Adicionamente, as relações de dependência entre os produtos de trabalho também foram copiadas.

É necessário considerar que durante a cópia dos produtos de trabalho para qualquer um dos processos de software, é o usuário quem escolhe se o produto de trabalho sendo copiado é do tipo interno ou externo (*ExternalUse* ou *InternalUse*). Apenas nos casos onde uma tarefa sendo copiada mantém relação de produção com um ou mais produtos de trabalhos, não será possível o usuário definir o tipo de produto de trabalho a ser criado, uma vez que para estes produtos de trabalho, instâncias de produtos de trabalho internos serão criadas automaticamente.

Considere-se, por exemplo, o apontamento no *Processo 3* realizado a partir *produto de trabalho interno W*. Durante tal apontamento, foi possível o usuário escolher que tipos de instâncias de produtos de trabalho (interno ou externo) deveriam ser criados para os *produtos de trabalho B* e *C*. Contudo, para os *produtos de trabalho X* e *Y*, os quais são produzidos pela *tarefa 1*, instâncias de produtos de trabalho do tipo interno foram criadas automaticamente.

Após entender como funciona a cópia dos elementos para um processo de software, explica-se a última definição realizada para o repositório de conteúdos do metamodelo sSPEM 2.0. Tal definição é relacionada com as alterações locais realizadas nos elementos que foram copiados do repositório (feitas nos processos de software). Tais definições são: (1) todo elemento copiado do repositório pode ser modificado nos processo de software; e (2) toda vez que um elemento de repositório sofre atualizações, todos os elementos dos processos de software que realizam apontamento para o elemento modificado e não sofreram alterações locais devem ser também atualizados.

5.6 Considerações Finais

Este capítulo apresentou o guia que auxilia a definição e adaptação de processos de software a partir do metamodelo sSPEM 2.0 e das regras de boa-formação. Também foi descrito como funciona o repositório de conteúdos nesse metamodelo.

O próximo capítulo desta tese apresenta a formalização das regras de boa-formação para consistência em lógica de primeira ordem.

6 FORMALIZAÇÃO DAS REGRAS DE BOA-FORMAÇÃO PARA CONSISTÊNCIA EM LÓGICA DE PRIMEIRA ORDEM

Este capítulo apresenta a formalização das regras de boa-formação que foram descritas no Capítulo 4. Para a formalização dessas regras, optou-se pela lógica de primeira ordem onde foram identificados as constantes e os predicados compondo o alfabeto da linguagem e, a partir deles, foram formalizados os axiomas.

Como base para a formalização mencionada, foram utilizados os diagramas de classes da UML que representam os pacotes do metamodelo sSPEM 2.0. Quanto ao alfabeto do diagrama de classes da UML, destaca-se que ele é composto por um conjunto de nomes de classes, um conjunto de nomes de atributos, um conjunto de nomes de associação e um conjunto de nomes de tipos de dados.

Nesta pesquisa, foram suficientes a utilização do conjunto de nomes de classes que, na linguagem formal utilizada, são os predicados definidos como Tipos e Subtipos; o conjunto de nomes de atributos que, são os predicados definidos como Propriedades; e um subconjunto dos nomes de associação definidos na UML, os quais são predicados tidos na linguagem utilizada, como Associações. Um Tipo representa um conceito em um processo de software e é representado por uma classe nos diagramas de classes que compõem o metamodelo sSPEM 2.0. Um Subtipo também representa um conceito em um processo de software, mas, em um diagrama de classes, caracteriza uma subclasse que especializa uma superclasse através do conceito de herança. As propriedades representam as características para um Tipo ou Subtipo e estão sempre associadas aos referidos conceitos. Por fim, uma Associação representa uma relação binária entre dois Tipos e/ou Subtipos.

A seguir, serão apresentadas, a linguagem formal utilizada nesta pesquisa e a formalização das regras de boa-formação para consistência de processos. Para facilitar o entendimento do capítulo, as regras estão organizadas exatamente da mesma maneira que no Capítulo 4, ou seja, divididas nos pacotes do metamodelo sSPEM 2.0. Vale lembrar que os identificadores de cada regra de boa-formação utilizados no Capítulo 4 são únicos, e, portanto, também serão empregados neste capítulo do trabalho.

6.1 Pacote *Process Structure*

O primeiro pacote do metamodelo sSPEM 2.0 a ter seus conceitos formalizados em lógica de primeira ordem foi o pacote *Process Structure*. Na Figura 55, é apresentado o diagrama de classes que representa as metaclasses desse pacote.

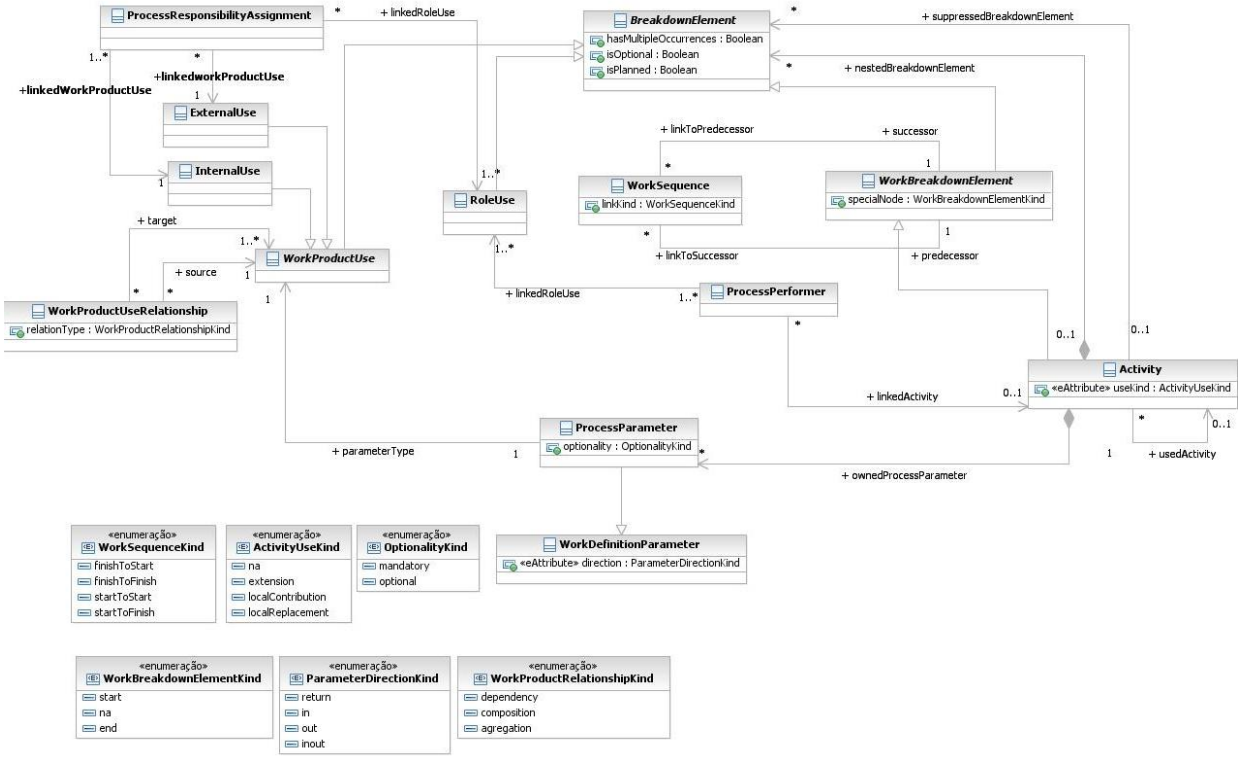

Figura 55 – Classes do pacote *Process Structure*

No contexto do diagrama de classes apresentado na Figura 55, observa-se que o elemento *WorkProductUse* se apresenta como uma metaclasse Abstrata que possui um relacionamento de generalização / especialização com as metaclasses *InternalUse* e *ExternalUse*. Em UML, uma classe abstrata é desenvolvida para representar entidades e conceitos abstratos. Assim, esse tipo de classe é sempre uma superclasse que não possui instâncias. Ela define um modelo (*template*) para um grupo de subclasses que herdam todo seu comportamento e os seus relacionamentos [Omg03].

Para formalizar a existência do elemento *WorkProductUse* e de suas especializações, foram definidos os seguintes predicados: o tipo *workProductUse(x)* indicando que *x* é um *WorkProductUse*; o sub-tipo *externalUse(x)* indicando que x é um *ExternalUse*; e o subtipo *internalUse(x)* indicando que x é um *InternalUse.* Como dito, os subtipos são criados para representar subclasses em UML que herdam o comportamento e os relacionamentos de superclasses através de uma associação chamada generalização / especialização. Como toda subclasse em UML é uma instância da

superclasse, para representar a notação de subtipo, os seguintes axiomas foram definidos:

$$\forall x(externalUse(x) \rightarrow workProductUse(x)) \quad \textbf{(A1)}$$

$$\forall x(internalUse(x) \rightarrow workProductUse(x)) \quad \textbf{(A2)}$$

Além dos predicados e axiomas acima, para a formalização das primeiras regras, foi necessário a criação de outros predicados. Nesse sentido, criou-se o tipo *processParameter(x)* que indica que x é um *ProcessParameter*. Este, por sua vez, representa um parâmetro de entrada e/ou saída para uma tarefa ou atividade em termos de um produto de trabalho (*ExternalUse* ou *InternalUse*) em um processo de software. Para indicar se o parâmetro é de entrada e/ou saída a metaclasse *ProcessParameter* possui um atributo chamado *direction,* o qual possui os valores de *In*, *Out* e *InOut*.

Na linguagem formal utilizada nesta pesquisa, o atributo *direction* é uma propriedade do tipo *ProcessParameter.* Para sua representação definiram-se os seguintes predicados: *direction(x, 'in'), direction(x, 'out')* e *direction(x, 'inout')*. Em todos os predicados, o x representa uma instância do tipo *ProcessParameter* e '*in*', '*out*' e '*inout*' se referem aos valores (constantes) de *direction* que o x pode assumir.

A metaclasse *ProcessParameter* também possui um atributo que indica qual é o *WorkProductUse (ExternalUse* ou *InternalUse)* que representa o parâmetro para a tarefa ou atividade. Tal atributo é chamado *parameterType* e é definido com o seguinte predicado: *parameterType (x, y)* onde x representa uma instância do tipo *ProcessParameter* e y representa uma instância do tipo *ExternalUse* ou do tipo *InternalUse*.

Com a utilização dos predicados e axiomas definidos acima, as primeiras regras para o elemento *WorkProductUse* foram escritas em lógica de primeira ordem e são:

Regra #1 - Os produtos de trabalho externos (ExternalUse) não podem ser produzidos em um processo de software.

$$\forall x\ (externalUse(x) \rightarrow \neg\ \exists y\ (processParameter(y) \wedge direction(y, \text{'out'}) \wedge parameterType(y, x)))$$

Regra #2 - Os produtos de trabalho externos (ExternalUse) devem ser consumidos e/ou modificados em um processo de software.

$$\forall x\ (externalUse(x) \rightarrow \exists y\ (processParameter(y) \wedge (\ direction(y, \text{'in'}) \vee direction(y, \text{'inout'})) \wedge parameterType(y, x)))$$

Regra #3 - Os produtos de trabalho internos (InternalUse) devem ser produzidos em um processo de software.

$\forall x\ (internalUse(x) \rightarrow \exists y\ (processParameter(y) \wedge direction(y, \text{'out'}) \wedge parameterType(y, x)))$

Regra #4 – Uma parâmetro de entrada e/ou saída (ProcessParameter) de uma atividade deve estar sempre associado a exatamente um produto de trabalho do tipo interno ou externo (InternalUse ou ExternalUse).

$\forall y\ (processParameter(y) \rightarrow \exists x\ ((internalUse(x) \vee externalUse(x)) \wedge parameterType(y,x)))$

A metaclasse *WorkProductUse* possui outras relações no diagrama de metaclasses da Figura 1 que necessitam ser traduzidas na linguagem formal sobre processos de software. Esse é o caso, por exemplo, das relações com as metaclasses *ProcessResponsabilityAssignment* e *WorkProductUseRelationship* que permitem, respectivamente, que papéis sejam atribuídos como responsáveis para um *WorkProductUse (ExternalUse ou InternalUse)* e que relacionamentos sejam criados entre *WorkProductUses* (*ExternalUse ou InternalUse).*

Para definir a relação de responsabilidade para um determinado *WorkProductUse,* os seguintes predicados foram criados: *processResponsabilityAssignment(x)* onde x representa uma instância da metaclasse *ProcessResponsabilityAssignment e linkedWorkProductUse (x, y)* onde x corresponde a uma instância da metaclasse *ProcessResponsabilityAssignment* e y representa uma instância da metaclasse *WorkProductUse* (*ExternalUse* ou *InternalUse)*.

Considerando a metaclasse *WorkProductUseRelationship* que, como já dito, permite determinar diferentes tipos de relacionamentos entre *WorkProductUses,* definiram-se os seguintes predicados: *workProductUseRelationship(x)* onde x representa uma instância de *WorkProductUseRelationship* e *relationType(x, 'composition')* onde x representa uma instância de *WorkProductUseRelationship* e '*composition*' o valor da propriedade (constante) *relationType.* A propriedade *relationType* também pode assumir os valores '*dependency*' e '*agregation'*. Foram necessários ainda os predicados *source(x, y)* e *target(x, y)* nos quais, em ambos os predicados x representa uma instância de *WorkProductUseRelationship* e y corresponde a uma instância de um *WorkProductUse* (*ExternalUse* ou *InternalUse*).

O *WorkProductUse* que está no *source* é o elemento que mantém relacionamento com o *WorkProductUse* que está no *target*. Considere, por exemplo, que o *InternalUse A* mantém um relacionamento do tipo *WorkProductUseRelationship* com o *ExternalUse B,*

com o valor de *relationType* igual a "*composition*". Se o *InternalUse A* está no *source* desta relação e o *ExternalUse B* está no *target,* significa dizer que o *InternalUse A* é uma composição de *ExternalUse B*. O predicado que define uma relação de composição entre dois *WorkProductUses* é o seguinte:

$\forall z,x,y$ *((externalUse(x)* $\vee$ *internalUse(x))* $\wedge$ *(externalUse(y)* $\vee$ *internalUse(y))* $\wedge$ *(workProductUseRelationship(z)* $\wedge$ *relationType(z, 'composition')* $\wedge$ *source(z, x)* $\wedge$ *target(z, y)))*

Uma vez que o predicado acima será bastante utilizado, optou-se por simplificá-lo, criando a seguinte sentença:

$\forall z,x,y$ *((externalUse(x)* $\vee$ *internalUse(x))* $\wedge$ *(externalUse(y)* $\vee$ *internalUse(y))* $\wedge$ *(workProductUseRelationship(z)* $\wedge$ *relationType(z, 'composition')* $\wedge$ *source(z, x)* $\wedge$ *target(z, y))* $\rightarrow$ *composicao(x, y))*

Agora, toda vez que for necessário estabelecer uma relação de composição entre dois *WorkProductUses,* usar-se-á o predicado *composicao(x,y)*. Considere, também, que o predicado acima existe para os outros valores de *relationType*. Dessa forma, para obter os predicados dos valores '*dependency*' e '*agregation*', basta trocar o valor de *relationType* e o nome do predicado. A relação de dependência possui um predicado adicional, pois algumas regras de boa-formação para consistência são válidas apenas para quando um *WorkProductUse* é do tipo *InternalUse*.

Dessa maneira, considere o seguinte predicado para o momento em que um *WorProductUse* (*InternalUse* ou *ExternalUse*) depende de um *InternalUse*:

$\forall z,x,y$ *((externalUse(x)* $\vee$ *internalUse(x))* $\wedge$ *internalUse(y) (workProductRelationship(z)* $\wedge$ *relationType(z, 'dependency')* $\wedge$ *source(z, x)* $\wedge$ *target(z, y))* $\rightarrow$ *dependencia-internal (x, y))*

Conforme explicado no Capítulo 4, as relações de composição, agregação e dependência são relações transitivas. Os axiomas abaixo formalizam essa propriedade para tais relações.

$\forall x,y,z$ *(composicao(x,y)* $\wedge$ *composicao(y,z)* $\rightarrow$ *composicao(x,z))* **(A3)**

$\forall x,y,z$ *(agregacao(x,y)* $\wedge$ *agregacao(y,z)* $\rightarrow$ *agregacao (x,z))* **(A4)**

$\forall x,y,z$ *(dependencia(x,y)* $\wedge$ *dependencia (y,z)* $\rightarrow$ *dependencia (x,z))* **(A5)**

Com base nas relações e predicados definidos acima, as seguintes regras de boa-formação para consistência foram definidas.

Regra #5 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode ser o "todo" em um relacionamento de composição se uma de suas "partes" já representa o seu "todo" em outro relacionamento de composição ou representa o seu "todo" pela transitividade da relação de composição.

$\forall x,y\ (composicao(x,y) \rightarrow \neg\ composicao(y,x))$

Regra #5 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode ser o "todo" em um relacionamento de agregação se uma de suas "partes" já representa o seu "todo" em outro relacionamento de agregação ou representa o seu "todo" pela transitividade da relação de agregação.

$\forall x,y\ (agregacao(x,y) \rightarrow \neg\ agregacao(y,x))$

Regra #5 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode depender de um produto de trabalho externo ou interno (ExternalUse ou InternalUse) que já é seu dependente.

$\forall x,y\ (dependencia(x,y) \rightarrow \neg\ dependencia(y,x))$

Regra #49 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode representar o "todo" e a "parte" em um relacionamento de composição.

$\forall x\ \neg\ composicao(x,x)$

Regra #49 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode representar o "todo" e a "parte" em um relacionamento de agregação.

$\forall x\ \neg\ agregacao(x,x)$

Regra #49 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode depender de si próprio.

$\forall x\ \neg\ dependencia(x,x)$

Regra #7 - Um produto de trabalho externo ou interno (ExternalUse ou InternalUse) não pode ser "parte" em mais de um relacionamento de composição.

$\forall x,y,z\ (composition(x,y) \rightarrow \neg\ composition(x,z))$

Regra #8 – Um produto de trabalho do tipo interno (InternalUse) deve possuir ao menos um relacionamento com a metaclasse ProcessResponsabilityAssignment.

$\forall x(internalUse(x) \rightarrow \exists w\ (processResponsabilityAssingment(w) \wedge linkedWorkProductUse(w,x)))$

As próximas regras de boa-formação formalizadas são relacionadas com os aspectos de duplicidade e opcionalidade e envolvem os elementos de processo *Atividade*,

Produto de Trabalho (Interno ou Externo) e *Papel*, bem como seus relacionamentos. Embora estes elementos de processo possuam muitos relacionamentos, os predicados definidos para a formalização das próximas regras de boa-formação foram apenas: o tipo *activity(x)* onde x representa uma instância da metaclasse *Activity*; o tipo *roleUse(x)* onde x corresponde a uma instância da metaclasse *RoleUse*; e o predicado *isOptional(x, 'true')* que representa uma propriedade para os elementos *Atividade*, *Produto de Trabalho (Interno ou Externo)* e *Papel.* Nesse último, x representa uma instância da metaclasse *Activity*, *RoleUse*, *InternalUse* ou *ExternalUse* e '*true'* ou *'false'* referem aos valores (constante) para a propriedade *isOptional* que o x pode assumir.

A partir dos novos predicados, as seguintes regras de boa-formação foram formalizadas:

Regra #25 – Uma atividade (Activity) opcional não deve possuir elementos obrigatórios.

$\forall x((activity(x) \wedge isOptional(x,'true')) \rightarrow \neg\exists y,z,w(internalUse(y) \wedge isOptional(y,'false')) \wedge (externalUse(z) \wedge isOptional(z,'false')) \wedge (roleUse(w) \wedge isOptional(w,'false'))))$

Regra #29 – Um produto de trabalho do tipo interno (InternalUse) obrigatório deve ser associado a pelo menos um papel (RoleUse) obrigatório.

$\forall x((internalUse(x) \wedge isOptional(x,'false')) \rightarrow \exists y,z\ ((roleUse(y) \wedge isOptional(y,'false')) \wedge (processResponsabilityAssignment(z) \wedge linkedWorkProductUse(z, x) \wedge linkedRoleUse(z, y))))$

Regra #33 – Não deve existir mais de um papel (RoleUse) com o mesmo nome em um processo de software.

$\forall x,y,n_1,n_2\ ((roleUse(x) \wedge name(x, n_1)) \wedge (roleUse(y) \wedge name(y, n_2)) \rightarrow n_1 \neq n_2)$

Regra #34 – Não deve existir mais de um produto de trabalho do tipo externo (ExternalUse) com o mesmo nome em um processo de software.

$\forall x,y,n_1,n_2\ ((externalUse(x) \wedge name(x, n_1)) \wedge (externalUse(y) \wedge name(y, n_2)) \rightarrow n_1 \neq n_2)$

Regra #34 – Não deve existir mais de um produto de trabalho do tipo interno (InternalUse) com o mesmo nome em um processo de software.

$\forall x,y,n_1,n_2\ ((internalUse(x) \wedge name(x, n_1)) \wedge (internalUse(y) \wedge name(y, n_2)) \rightarrow n_1 \neq n_2)$

Regra #36 – Um papel (RoleUse) não deve ser definido mais de uma vez como responsável pelo mesmo produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse).

$\forall x,y,z((roleUse(x) \wedge workProductUse(y) \wedge (processResponsabilityAssignment(r) \wedge linkedRoleUse(r,x) \wedge linkedWorkProductUse(r,y))) \rightarrow \neg\exists w(processResponsabilityAssignment\ (w) \wedge linkedRoleUse(w,x) \wedge linkedWorkProductUse(w,y)))$

Regra #37 – Um produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse) não deve ser definido mais de uma vez como dependente do mesmo produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse).

$\forall x,y,z((internalUse(x) \vee ExternalUse(x)) \wedge (internalUse(y) \vee ExternalUse(y)) \wedge (workProductUseRelationship(r) \wedge source(r,x) \wedge target(r,y) \wedge relationType(r,'dependency')) \rightarrow \neg\exists w(workProductUseRelationship\ (w) \wedge source(w,x) \wedge target\ (w,y) \wedge relationType(r,'dependency')))$

Regra #38 – Um produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse) não deve ser definido mais de uma vez como o todo, através de uma relação de composição, do mesmo produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse).

$\forall x,y,z((internalUse(x) \vee ExternalUse(x)) \wedge (internalUse(y) \vee ExternalUse(y)) \wedge (workProductUseRelationship(r) \wedge source(r,x) \wedge target(r,y) \wedge relationType(r,'composition')) \rightarrow \neg\exists w(workProductUseRelationship\ (w) \wedge source(w,x) \wedge target\ (w,y) \wedge relationType(r,'composition')))$

Regra #39 – Um produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse) não deve ser definido mais de uma vez como o todo, através de uma relação de agregação, do mesmo produto de trabalho do tipo externo ou interno (ExternalUse ou InternalUse).

$\forall x,y,z((internalUse(x) \vee ExternalUse(x)) \wedge (internalUse(y) \vee ExternalUse(y)) \wedge (workProductUseRelationship(r) \wedge source(r,x) \wedge target(r,y) \wedge relationType(r,'agregation')) \rightarrow \neg\exists w(workProductUseRelationship\ (w) \wedge source(w,x) \wedge target\ (w,y) \wedge relationType(r,'agregation')))$

Observando as relações do elemento *Activity* no pacote *Process Struc*ture, nota-se que esse elemento possui um relacionamento de composição com a metaclasse abstrata *BreakdownElement,* a qual tem como subclasses *ProcessParameter, ProcessPerformer, WorkProductUse, RoleUse*, entre outras. Desse modo, para representar tal relação definiu-se o predicado de associação *parte-de(x, y)* que indica que x é uma parte de y. Na relação entre *Activity* e *ProcessParameter,* por exemplo, pode-se usar esse predicado para descrever a relação de composição existente entre as referidas metaclasses. Assim, considerando o predicado *parte-de,* derivam-se os seguintes axiomas:

$\forall x \neg parte\text{-}de(x,x)$ **(A6)**

$\forall x,y\ (parte\text{-}de(x,y) \rightarrow \neg parte\text{-}de(y,x))$ **(A7)**

$\forall x,y,z\ (parte\text{-}de(x,y) \wedge parte\text{-}de(y,z) \rightarrow parte\text{-}de(x,z))$ **(A8)**

Outra relação do elemento *Activity,* herdada da metaclasse *WorkBreakdownElement*, que precisou ser formalizada foi a relação com a metaclasse *WorkSequence* que permite definir o sequenciamento entre atividades de um processo de software. Para capturar essa relação de seqüenciamento entre *atividades,* foi necessário a criação de vários predicados e axiomas que formalizam esta relação.

Dessa maneira, os primeiros predicados criados são relacionados com a metaclasse *WorkSequence* e seus atributos. Tais predicados são: *workSequence(x)*; *predecessor(x, y)*; *sucessor(x, y)* e *linkKind(x, 'FS')*. Em todos eles, o x é uma instância de *WorkSequence*. Nos predicados *predecessor* e *sucessor* o y representa uma instância de *Activity,* e, no predicado *linkKind* '*FS*' representa o valor *finishToStart*. O último predicado também pode assumir os outros valores (constantes) do atributo *linkKind* que são '*FF*' (*finishToFinish*), '*SF*' (*startToFinish*) e '*SS*' (*startToStart*). Outro predicado definido foi o predicado *specialNode(x, 'start')* onde x é uma instância de *Activity* e '*start'* denota o tipo desta *Activity*. O predicado *specialNode* pode assumir também os valores (constantes) '*end'* e '*na*'.

Embora as metaclasses e relações descritas acima, mostradas na Figura 55, sejam suficientes para expressar o conceito de precedência entre tarefas, na linguagem utilizada nesta pesquisa, foi definido um novo conjunto de conceitos e predicados. A ideia é que as mesmas informações presentes no modelo da UML possam ser expressas. Contudo, considerando a lógica de primeira ordem, os novos conceitos e predicados facilitam o entendimento das regras de boa-formação que consideram as informações de precedência.

Nesse sentido, afim de capturar o conceito de atividade predecessora e sucessora criou-se os predicados *pré-atividade* (a_1, a_2) e *pós-atividade* (a_2, a_1) indicando, respectivamente, que a_1 é uma atividade predecessora de a_2 ou, de forma inversa, que a_2 é uma atividade sucessora de a_1. Como pode ser visto, em ambos os predicados, a_1 e a_2 são instâncias de *Activity*. Considerando que as relações de precedência pré e pós-atividade são transitivas e assimétricas, os seguintes axiomas foram definidos:

$\forall(a_1, a_2)\ (pre\text{-}atividade(a_1, a_2) \leftrightarrow pos\text{-}atividade(a_2, a_1))$ **(A9)**

$\forall(a_1, a_2, a_3)\ (pre\text{-}atividade(a_1, a_2) \wedge pre\text{-}atividade(a_2, a_3) \rightarrow pre\text{-}atividade(a_1, a_3))$ **(A10)**

$\forall(a_1, a_2)$ (*pre-atividade*$(a_1, a_2) \rightarrow \neg$ *pre-atividade*(a_2, a_1)) **(A11)**

$\forall a_1 \neg$ *pre-atividade*(a_1, a_1) **(A12)**

Para formalizar os conceitos de atividade-inicio e atividade-fim, os quais foram incluídos nesta pesquisa no metamodelo sSPEM 2.0 (conforme Capítulo 4), os seguintes predicados foram definidos: *atividade-inicio(a, a_i)* e *atividade-fim(a, a_f)*, nos quais *a* representa uma instância de *Activity* e a_i e a_f identificam, respectivamente, o início e o fim dessa *Activity*. No intuito de registrar que toda atividade realmente possui um início e um fim e que o início precede o fim, considere os seguintes axiomas:

$\forall x$(*activity(x)* $\rightarrow \exists!(x1,x2)$ (*activity(x1)* $\wedge$ *atividade-inicio(x,x1)* $\wedge$ *activity(x2)* $\wedge$ *atividade-fim(x,x2)*) **(A13)**

$\forall x \exists!(x1,x2)$ (*activity(x)* $\rightarrow$ *atividade-inicio(x, x1)* $\wedge$ *atividade-fim(x, x2)* $\wedge$ *pre-atividade (x1,x2)*) **(A14)**

Utilizando os novos conceitos de atividade-início e atividade-fim, traduziram-se os valores do atributo *linkKind* através dos seguintes predicados:

$\forall x, x1, x2$ (*workSequence(x)* $\wedge$ *activity(x1)* $\wedge$ *activity(x2)* $\wedge$ *predecessor(x, x1)* $\wedge$ *sucessor(x, x2)* $\wedge$ *linkKind(x, FS)* $\rightarrow \exists!(a_1,a_2)$ (*atividade-fim(x1, a_1)* $\wedge$ *atividade-inicio(x2, a_2)* $\wedge$ *pre- atividade(a_1,a_2)*))

$\forall x, x1, x2$ (*workSequence(x)* $\wedge$ *activity(x1)* $\wedge$ *activity(x2)* $\wedge$ *predecessor(x, x1)* $\wedge$ *sucessor(x, x2)* $\wedge$ *linkKind(x, FF)* $\rightarrow \exists!(a_1,a_2)$ (*atividade-fim(x1, a_1)* $\wedge$ *atividade-fim(x2, a_2)* $\wedge$ *pre-atividade (a_1,a_2)*))

$\forall x, x1, x2$ (*workSequence(x)* $\wedge$ *activity(x1)* $\wedge$ *activity(x2)* $\wedge$ *predecessor(x, x1)* $\wedge$ *sucessor(x, x2)* $\wedge$ *linkKind(x, SS)* $\rightarrow \exists!(a_1,a_2)$ (*atividade-inicio(x1, a_1)* $\wedge$ *atividade-inicio(x2, a_2)* $\wedge$ *pre- atividade(a_1,a_2)*))

$\forall x, x1, x2$ (*workSequence(x)* $\wedge$ *activity(x1)* $\wedge$ *activity(x2)* $\wedge$ *predecessor(x, x1)* $\wedge$ *sucessor(x, x2)* $\wedge$ *linkkind(x, SF)* $\rightarrow \exists!(a_1,a_2)$ (*atividade-inicio(x1, a_1)* $\wedge$ *atividade-fim(x2, a_2)* $\wedge$ *pre- atividade(a_1,a_2)*))

Os predicados acima estabelecem precedência, considerando os inícios e fins de duas atividades. Para registrar que uma atividade precede outra atividade, sem precisar considerar seus inícios e fins, os seguintes predicados foram definidos:

$\forall x, y, x1, x2$ (*activity(x)* $\wedge$ *activity(y)* $\wedge$ *atividade-inicio(x, x1)* $\wedge$ *atividade-inicio(y, x2)* $\wedge$ *pre_atividade(x_1, x_2)* $\rightarrow$ *pre-atividade(x, y)*)

$\forall x, y, x1, x2$ (*activity(x)* $\wedge$ *activity(y)* $\wedge$ *atividade-inicio(x, x1)* $\wedge$ *atividade-fim(y, x2)* $\wedge$ *pre-atividade(x_1, x_2)* $\rightarrow$ *pre-atividade(x, y)*)

$\forall x, y, x1, x2$ (activity(x) $\wedge$ activity(y) $\wedge$ atividade-fim(x, x1) $\wedge$ atividade-inicio(y, x2) $\wedge$ pre-atividade(x_1, x_2) $\rightarrow$ pre-atividade(x, y))

$\forall x, y, x1, x2$ (activity(x) $\wedge$ activity(y) $\wedge$ atividade-fim(x, x1) $\wedge$ atividade-fim(y, x2) $\wedge$ pre-atividade(x_1, x_2) $\rightarrow$ pre- atividade(x, y))

Com base nos axiomas e predicados que consideram a precedência de atividades em um processo de software, as seguintes regras de boa-formação foram formalizadas.

Regra #12 – Um processo de software deve possuir exatamente um nodo final.

$\exists!x$ (activity(x) $\wedge$ specialNodel(x,'end'))

Regra #13 – Um processo de software deve possuir exatamente um nodo inicial.

$\exists!x$ (activity(x) $\wedge$ specialNodel(x,'start'))

Regra #14 – O nodo inicial não pode ser sucessor para nenhuma atividade no sequenciamento definido em um processo de software.

$\forall x$ ((activity(x) $\wedge$ specialNode(x,'start')) $\rightarrow \neg \exists y$ (worksequence(y) $\wedge$ sucessor(y,x)))

Regra #15 – O nodo final não pode ser predecessor para nenhuma atividade no sequenciamento definido em um processo de software.

$\forall x$ (activity(x) $\wedge$ specialNode(x,'end') $\rightarrow \neg \exists y$ (worksequence(y) $\wedge$ predecessor(y,x)))

Regra #43 – O nodo inicial deve ser definido como predecessor para pelo menos uma atividade no sequenciamento definido em um processo de software.

$\forall x$ (activity(x) $\wedge$ specialNode(x,'start') $\rightarrow \exists y$ (worksequence(y) $\wedge$ predecessor(y,x)))

Regra #44 – O nodo final deve ser definido como sucessor para pelo menos uma atividade no sequenciamento definido em um processo de software.

$\forall x$ (activity(x) $\wedge$ specialNode(x,'end') $\rightarrow \exists y$ (worksequence(y) $\wedge$ sucessor(y,x)))

Regra #45 – O sequenciamento estabelecido entre as atividades e o nodo inicial ou final em um processo de software deve ser realizado através do tipo de sequenciamento finishToStart.

$\forall x,y, z$ ((activity(x) $\wedge$ specialNode(x,'start')) $\wedge$ activity(y) $\wedge$ (workSequence(z) $\wedge$ predecessor(z, x) $\wedge$ sucessor(z,y)) $\rightarrow$ linkKind (z, 'FS'))

∀x,y, z ((activity(x) ∧ specialNode(x,'end')) ∧ activity(y) ∧ (workSequence(z) ∧ predecessor(z, y) ∧ sucessor(z,x)) → linkKind (z, 'FS'))

Regra #41 – As atividades do processo que representam os nodos inicial e final não devem ser associados com instâncias da metaclasse ProcessPerformer.

∀x(((activity(x) ∧ specialNode(x,'end')) ∨ (activity(x) ∧ specialNode(x,'start'))) → ¬∃y (processPerformer(y) ∧ linkedActivity(y, x)))

Regra #42 – As atividades do processo que representam os nodos inicial e final não devem possuir elementos.

∀x (((activity(x) ∧ specialNode(x,'end')) ∨ (activity(x) ∧ specialNode(x,'start'))) → ¬∃a, b, c, d, e ,f, g, h, i ((roleUse(a) ∧ parte-de(a, x)) ∨ (activity(b) ∧ parte-de(b, x)) ∨ (externalUse(c) ∧ parte-de(c, x)) ∨ (internalUse(d) ∧ parte-de(d, x)) ∨ (milestone(e) ∧ parte-de(e, x)) ∨ (processParameter(f) ∧ parte-de(f, x)) ∨ (processPerformer(f) ∧ parte-de(f, x)) ∨ (workProductUseRelationship(g) ∧ parte-de(g, x)) ∨ (processResponsabilityAssignment(h) ∧ parte-de(h, x)) ∨ (workSequence(i) ∧ parte-de(i, x))))

Regra #16 – O sequenciamento de atividades não deve possuir situações que representem um deadlock na execução de um processo de software.

∀(x1, x2) (activity(x1) ∧ activity(x2) ∧ pre-atividade(x1, x2) → ¬ pre-atividade(x2, x1))

Regra #17 – Não devem existir transições duplicadas entre duas atividades em um sequenciamento.

∀x, x1, x2(workSequence(x) ∧ activitvy(x1) ∧ activity(x2) ∧ predecessor(x, x1) ∧ sucessor(x, x2) ∧ linkKind(x, 'SF') →¬∃y (workSequence(y) ∧ predecessor(x, x1) ∧ sucessor(x, x2)∧ linkKind(x, 'SF')))

∀x, x1, x2 (workSequence(x) ∧ activity(x1) ∧ activiy(x2) ∧ predecessor(x, x1) ∧ sucessor(x, x2) ∧ linkKind(x, 'FF') →¬∃y (workSequence(y) ∧ predecessor(x, x1) ∧ sucessor(x, x2)∧linkKind(x, 'FF')))

∀x, x1, x2 (workSequence(x) ∧ activity(x1) ∧ activity(x2) ∧ predecessor(x, x1) ∧ sucessor(x, x2) ∧ linkKind(x, 'SS') →¬∃y (workSequence(y) ∧ predecessor(x, x1) ∧ sucessor(x, x2) ∧ linkKind(x, 'SS')))

∀x, x1, x2 (workSequence(x) ∧ activity(x1) ∧ activity(x2) ∧ predecessor(x, x1) ∧ sucessor(x, x2) ∧ linkKind(x, 'FS')→¬∃y (workSequence(y) ∧ predecessor(x, x1) ∧ sucessor(x, x2) ∧ linkKind(x, 'FS')))

Regra #18 – Todas as atividades que são sequenciadas em um processo de software deverão ter ligação com o nodo inicial e final do processo para garantir

que toda atividade é iniciada após o nodo inicial e possui seu término antes do nodo final .

∀x, y, z, w (activity(x) ∧ (workSequence(y) ∧ predecessor(y, x) ∨ sucessor(y,x)) ∧ (activity(z) ∧ specialNode(z, 'start')) ∧ (activity(w) ∧ specialNode(w, 'end')) → (pre-atividade(z,x) ∧ pos-atividade (x,w)))

A última regra de formação formalizada é relacionada com o elemento *Activity* e o mecanismo de adaptação *usedActivity*.

Regra #46 – Uma atividade não pode herdar seu próprio conteúdo.

∀x (activity(x) ∧ (useKind(x, 'extension') ∨ useKind(x, 'localContribution') ∨ useKind(x, 'localReplacement')) → ¬ usedActivity(x, x))

6.2 Pacote Process with Methods

Embora no Capítulo 4 deste trabalho, as regras de boa-formação do pacote *Method Content* tenham sido apresentadas antes das regras do pacote *Process with Methods*, optou-se, no presente capítulo, por uma organização diferente. Isso se deu com um intuito de permitir uma maior reutilização de predicados e axiomas para a descrição das regras do pacote *Method Content*, as quais serão apresentadas na próxima seção.

Para a formalização dos predicados e axiomas dos conceitos do pacote *Process with Methods,* foi utilizado o diagrama de classes mostrado na Figura 56. Basicamente, os novos predicados e axiomas definidos nesse pacote são relacionados com o elemento de processo tarefa, que é representado no diagrama da Figura 56 pela metaclasse *TaskUse*.

Para formalizar a existência do elemento tarefa, definiu-se o predicado *taskUse(x),* no qual o x representa uma instância de *TaskUse*. No pacote *Process with Methods,* novas relações são definidas para o elemento *TaskUse* como mostra o diagrama de classes da Figura 56. Nesta figura, pode-se observar que *TaskUse* possui um relacionamento de composição com o elemento *ProcessParameter* que pode ser representando pelo predicado *parte-de(x,y),* o qual já foi definido no pacote *Process Structure*.

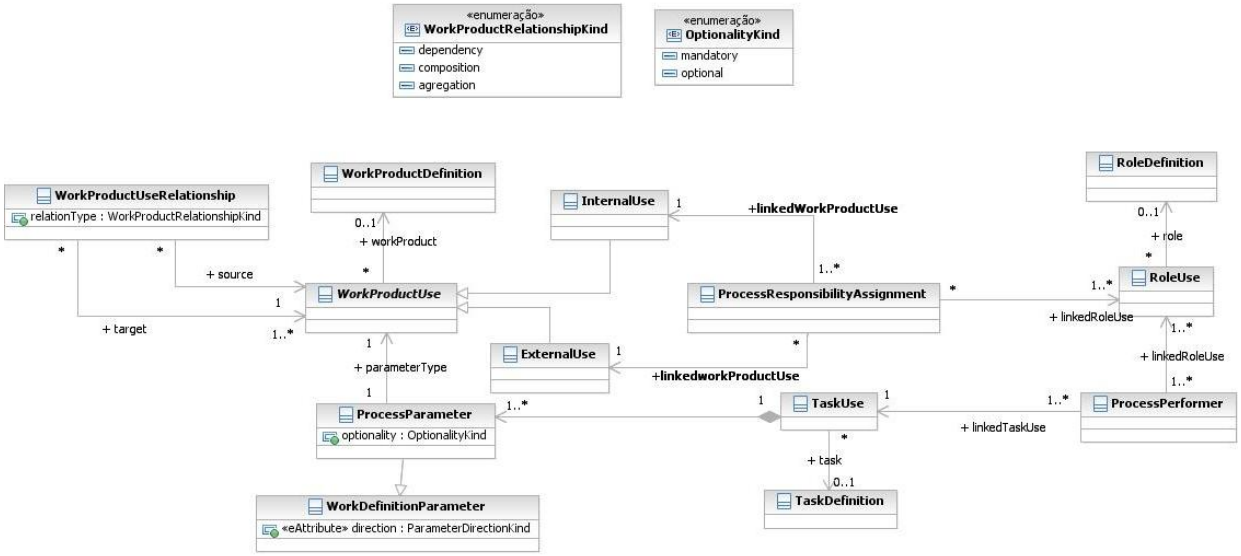

Figura 56 – Classes do pacote *Process with Methods*

A metaclasse *TaskUse* também possui relação com a metaclasse *ProcessPerformer*. Através dessa relação, será possível associar um papel para desempenhar uma tarefa. Para representar a mencionada relação, foram definidos dois novos predicados: o tipo *processperformer(x)* onde x é uma instância de *ProcessPerformer* e o predicado *linkedTaskUse(x, y)* que representa uma propriedade de *ProcessPerfomer*. Neste predicado x é uma instância de *ProcessPerformer* e y é uma instância de *TaskUse*.

Considerando os predicados acima e aqueles já definidos para o pacote *Process Structure*, formalizou-se as seguintes regras de boa-formação:

Regra # 10 – Uma tarefa (TaskUse) deve possuir ao menos um relacionamento com a metaclasse ProcessPerformer.

$\forall x\ (taskUse(x) \rightarrow \exists y\ (processPerformer(y) \wedge linkedTaskUse(y,x)))$

Regra # 48 – Um papel (RoleUse) deve possuir ao menos um relacionamento com a metaclasse ProcessPerformer.

$\forall x\ (roleUse(x) \rightarrow \exists y\ (processPerformer(y) \wedge linkedRoleUse(y,x)))$

Regra # 9 – Uma tarefa (TaskUse) deve modificar ao menos um produto de trabalho do tipo interno ou externo (InternalUse ou ExternalUse) e/ou produzir ao menos um produto de trabalho do tipo interno (InternalUse).

$\forall x\ (taskUse(x) \rightarrow \exists y,z\ (((processParameter(y) \wedge direction(y, \text{'inout'})) \wedge (externalUse(z) \vee internalUse(z))) \vee ((processParameter(y) \wedge direction(y, \text{'out'})) \wedge internalUse(z)) \wedge parte\text{-}de(y,x)))$

Regra # 21 – Uma atividade (Activity) deve possuir ao menos uma tarefa (TaskUse).

$\forall x\ (activity(x) \rightarrow \exists t\ (taskUse(t) \wedge parte\text{-}de(t, x)))$

Algumas regras que envolvem o elemento *TaskUse* e o elemento *WorkProductUse* (*InternalUse* e *ExternalUse*) também foram formalizadas. Contudo, para a formalização destas novas regras alguns predicados novos tornam-se necessários. O primeiro deles é o predicado que representa a produção de um *InternalUse* por uma *TaskUse*. O predicado que representa esta sentença é o seguinte:

$\forall x,y,z\ (taskUse\ (x) \wedge internalUse(z) \wedge (processParameter(y) \wedge direction(y,'out') \wedge parametertype(y, z)) \wedge parte\text{-}de(y, x) \rightarrow tarefa\text{-}produz\ (x, z))$

Vale lembrar ainda que só *WorkProductUse* do tipo *InternalUse* pode ser produzido em um processo de software. Dessa forma, esse predicado só considera o referido tipo de elemento. Além do predicado para produção de *InternalUse,* têm-se os predicados de modificação e consumo de produtos de trabalho por parte de uma *TaskUse*, os quais consideram ambos os elementos *InternalUse* e *ExternalUse*:

$\forall\ x,y,z\ (taskUse(x) \wedge (internalUse(z) \vee externalUse(z)) \wedge (processParameter(y) \wedge direction(y,'inout') \wedge parameterType(y, z)) \wedge parte\text{-}de(y, x) \rightarrow tarefa\text{-}modifica\ (x, z))$

$\forall x,y,z\ (taskUse(x) \wedge (internalUse(z) \vee externalUse(z)) \wedge (processParameter(y) \wedge direction(y,'in') \wedge parameterType(y, z)) \wedge parte\text{-}de(y, x) \rightarrow tarefa\text{-}consome\ (x, z))$

Por fim, o último predicado criado estabelece o consumo de elementos do tipo *InternalUse* nas *TaskUses.*

$\forall x,y,z\ (taskUse\ (x) \wedge internalUse(z) \wedge (processParameter(y) \wedge direction(y,'in') \wedge parametertype(y, z)) \wedge parte\text{-}de(y, x) \rightarrow tarefa\text{-}consome\text{-}internal(x, z))$

Este último predicado foi criado, uma vez que algumas regras de boa-formação para consistência são definidas, considerando apenas o consumo de *InternalUse*.

Todavia, antes de apresentar as novas regras de boa-formação, que são baseadas nos novos predicados e axiomas, torna-se necessário definir que alguns dos predicados acima, mais especificamente os predicados *tarefa-produz (x, y)*, *tarefa-consome(x,y)* e *tarefa-modifica(x,y)*, também existem para a metaclasse *Activity*, uma vez que, como explicado, essa metaclasse possui relação com a metaclasse *ProcessParameter*. Assim, considera-se que, para obter os predicados *atividade-produz (x, y)*, *atividade-consome(x,y)* e *atividade-modifica(x,y)* basta que x seja representado por uma instância de *Activity*.

As próximas regras de boa-formação formalizadas foram:

Regra #22 – Os parâmetros de entrada e/ou saída (ProcessParameter) de uma atividade (Activity) devem ser compatíveis com os parâmetros de entrada e/ou saída (ProcessParameter) de suas tarefas (TaskUse).

$\forall x,y$ (atividade-consome(x,y) $\rightarrow \exists t$ (tarefa-consome(t,y) $\wedge$ parte-de(t,x))

$\forall x,y$ (atividade-modifica(x,y) $\rightarrow \exists t$ (tarefa-modifica (t,y) $\wedge$ parte-de(t,x))

$\forall x,y$ (atividade-produz(x,y) $\rightarrow \exists t$ (tarefa-produz(t,y) $\wedge$ parte-de(t,x))

Regra #11 – Todas as dependências de um produto de trabalho do tipo interno (InternalUse) devem estar conectadas como entrada nas suas tarefas (TaskUse) de produção.

$\forall x,y,t$(internalUse(x)$\wedge$(internalUse(y) $\vee$ (extenalUse(y)) $\wedge$ dependencia(x,y) $\wedge$ tarefa-produz(t,x) $\rightarrow$ tarefa-consome(t,y))

Regra #24 – Toda tarefa (TaskUse) que produz um produto de trabalho do tipo interno (InternalUse) deverá estar associada com ao menos um papel (RoleUse) que é responsável por este produto de trabalho.

$\forall x,y$(tarefa-produz (x, y) $\rightarrow \exists r, w, z$ (roleUse(r) $\wedge$ (processPerfomer(z) $\wedge$ linkedTaskUse(z, x)$\wedge$ linkedRoleUse(z,r)) $\wedge$ (processResponsabilityAssingment(w) $\wedge$ linkedRoleUse(w,r)$\wedge$ linkedWorkProductUse(w,y))))

Regra #23 – Um papel (RoleUse) obrigatório deve ser associado a pelo menos uma tarefa (TaskUse) obrigatória.

$\forall x$ ((roleUse(x) $\wedge$ isOptional(x, 'false')) $\rightarrow \exists y,z$ ((taskUse(y) $\wedge$ isOptional(y, 'false')) $\wedge$ (processPerformer(z) $\wedge$ linkedTaskUse(z, y) $\wedge$ linkedRoleUse(z, x))))

Regra #25 – Uma atividade (Activity) opcional não deve possuir elementos obrigatórios.

$\forall x$((activity(x) $\wedge$ isOptional(x,'true)) $\rightarrow \neg\exists y,z,w$((taskUse(y) $\wedge$ isOptional(y, 'false')) $\wedge$ ((internalUse(z) $\vee$ externalUse(z)) $\wedge$ isOptional(z, 'false')) $\wedge$ (roleUse(w) $\wedge$ isOptional(w, 'false'))))

Regra #26 – Uma atividade (Activity) obrigatória deve possuir ao menos uma tarefa (TaskUse) obrigatória.

$\forall x$((activity(x) $\wedge$ isOptional(x, 'false')) $\rightarrow \exists y$ (taskUse(y) $\wedge$ isOptional(y, 'false')))

Regra #27 – Um produto de trabalho do tipo externo (ExternalUse) obrigatório deve ser consumido e/ou modificado por pelo menos uma tarefa (TaskUse) obrigatória.

∀x (((externalUse(x) ∧ isOptional(x, 'false')) → ∃y,z ((taskUse(y) ∧ isOptional(y, 'false')) ∧ (processParameter(z) ∧ (direction(z,'in') ∨ direction(z, 'inout')) ∧ parameterType(z, x)) ∧ parte-de(z,y)))

Regra #28 – Um produto de trabalho do tipo interno (InternalUse) obrigatório deve ser produzido por pelo menos uma tarefa (TaskUse) obrigatória.

∀x (((internalUse(x) ∧ isOptional(x, 'false')) → ∃y,z ((taskUse(y) ∧ isOptional(y, 'false')) ∧ (processParameter(z) ∧ direction(z,'out') ∧ parameterType(z, x)) ∧ parte-de(z,y)))

Regra #30 – Uma tarefa (TaskUse) obrigatória deve modificar pelo menos um produto de trabalho do tipo externo (ExternalUse) obrigatório e/ou produzir ao menos um produto de trabalho do tipo interno (InternalUse) obrigatório.

∀x ((taskUse(x) ∧ isOptional(x, 'false')) → ∃ y,z (((extenalUse(y) ∧ isOptional(y, 'false')) ∧ (processParameter(z) ∧ direction(z,'inout') ∧ parameterType(z, y)) ∧ parte-de(z,x)) ∨ ((internalUse(y) ∧ isOptional(y, 'false')) ∧ (processParameter(z) ∧ direction(z,'out') ∧ parameterType(z, y)) ∧ parte-de(z,x))))

Regra #31 – Toda tarefa (TaskUse) obrigatória que produz um produto de trabalho do tipo interno (InternalUse) obrigatório deverá estar associada com ao menos um papel (RoleUse) obrigatório que é responsável por este produto de trabalho.

∀ x,y ((tarefa-produz (x, y) ∧ isOptional(x, 'false')) → ∃r, w, z ((roleUse(r) ∧ isOptional(r, 'false)) ∧ (processPerfomer(z)∧linkedTaskUse(z,x)∧linkedRoleUse(z,r))∧ (processResponsabilityAssingment(w) ∧ linkedRoleUse(w,r)∧ linkedWorkProductUse(w,y))))

Regra #32 – Um tarefa (TaskUse) obrigatória deve ser associada a pelo menos um papel (RoleUse) obrigatório.

∀x ((taskUse(x) ∧ isOptional(x, 'false')) → ∃ y, z ((roleUse(y) ∧ isOptional(y, 'false')) ∧ (processPerformer(z) ∧ linkedTaskUse(z, x) ∧ linkedRoleUse(z, y))))

Regra #35 – Um papel (RoleUse) não deve ser definido mais de uma vez como executor da mesma tarefa (TaskUse).

∀x,y,z (taskUse(x) ∧ roleUse(y) ∧ (processPerformer(z) ∧ linkedTaskUse(z, x) ∧ linkedRoleUse(z, y)) → ¬∃w (processPerfomer(w) ∧ linkedTaskUse(w, x) ∧ linkedRoleUse(w, y)))

Regra #40 - Uma tarefa (TaskUse) não deve possuir dois ou mais parâmetros de entrada e/ou saída (ProcessParameter) iguais.

$\forall x,y,z((taskUse(x) \wedge (internalUse(z) \vee externalUse(z)) \wedge (processParameter(y) \wedge direction(y,'out') \wedge parameterType(y,z)) \wedge parte\text{-}de(y,x)) \rightarrow \neg\exists w\ (processParameter(w) \wedge direction(w,'out') \wedge parameterType(w,z)) \wedge part\text{-}of(w,x)))$

$\forall x,y,z((taskUse(x) \wedge (internalUse(z) \vee externalUse(z)) \wedge (processParameter(y) \wedge direction(y,'inout') \wedge parameterType(y,z)) \wedge parte\text{-}de(y,x)) \rightarrow \neg\exists w\ (processParameter(w) \wedge direction(w,'inout') \wedge parameterType(w,z)) \wedge part\text{-}of(w,x)))$

$\forall x,y,z((taskUse(x) \wedge (internalUse(z) \vee externalUse(z)) \wedge (processParameter(y) \wedge direction(y,'in') \wedge parameterType(y,z)) \wedge parte\text{-}de(y,x)) \rightarrow \neg\exists w\ (processParameter(w) \wedge direction(w,'in') \wedge parameterType(w,z)) \wedge part\text{-}of(w,x)))$

Uma vez que existem regras de boa-formação relacionadas com o sequenciamento do elemento *tarefa,* vários outros predicados e axiomas foram necessários para formalizá-las. Contudo, como, no *pacote Process Structure,* já foram demonstradas a formalização de predicados e axiomas bastante similares aos utilizados neste ponto do estudo, optou-se por apresentar a formalização das regras de boa-formação específicas para a definição de um sequenciamento consistente entre tarefas no Apêndice D. Portanto, as regras de boa-formação que se encontram neste apêndice são: *Regra 6, Regra #12, Regra #13, Regra #14, Regra #15, Regra #16, Regra #17, Regra #18, Regra #41, #Regra #42, Regra #43, Regra #44*, *Regra #45* e *Regra #50*.

Considerando todos os axiomas e predicados definidos anteriormente, assim como aqueles definidos no Apêndice D, as últimas regras de boa-formação puderam ser formalizadas para pacote *Process with Methods* e são:

Regra # 19 – Todas as dependências de um produto de trabalho do tipo interno (InternalUse) devem ser produzidas antes deste produto de trabalho em um processo de software.

$\forall x,y\ (dependencia\text{-}internal(x,y) \rightarrow \exists t1, t2\ (tarefa\text{-}produz(t1, x) \wedge tarefa\text{-}produz(t2, y) \wedge pre\text{-}tarefa(t2,t1)))$

Regra # 20 – Todos os produtos de trabalho do tipo interno (InternalUse) devem ser produzidos antes de serem consumidos em um processo de software.

$\forall t1, x\ (tarefa\text{-}consome\text{-}internal(t1,x) \rightarrow \exists t2\ (tarefa\text{-}produz(t2, x) \wedge pre\text{-}tarefa(t2,t1)$

6.3 Pacote Method Content

A definição dos predicados e axiomas para a formalização das regras do pacote *Method Content* foi realizada da mesma forma que no pacote *Process Structure* e *Process with Methods*. Assim, considera-se que cada metaclasse mostrada no diagrama da Figura 57 (diagrama do pacote *Method Content*) representa um novo predicado.

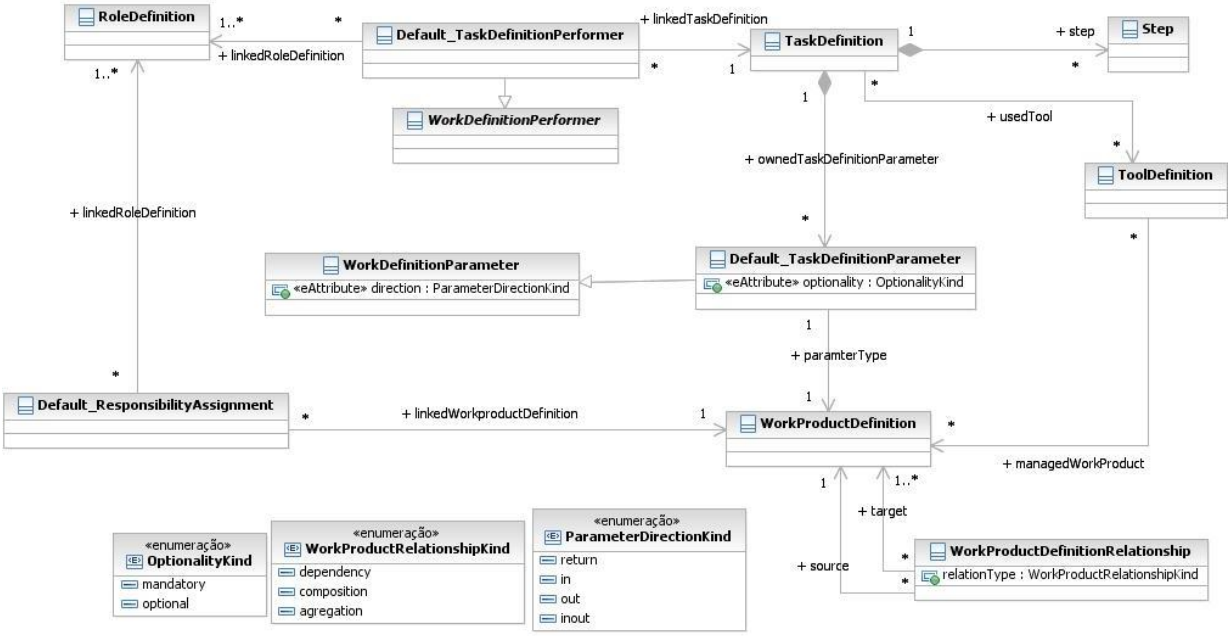

Figura 57 – Classes do pacote *Method Content*

Além dos mencionados predicados, muitos outros que representam as propriedades das metaclasses também foram criados. Por fim, alguns dos axiomas e predicados definidos para o pacote *Process Structure* e *Process with Methods* são válidos também no pacote *Method Content*. Contudo, para estes axiomas e predicados, devem ser consideradas as metaclasses do pacote *Method Content* e alguns deles necessitam ser diferenciados em termos de suas nomenclaturas. Para o caso, por exemplo, do predicado *dependencia(x,y)* que no pacote *Process Structure,* é definido com as metaclasses *WorkProductUseRelationship*, *ExternalUse* e *InternalUse,* considera-se, que, no pacote *Method Content,* esse predicado se chama *dependenciaMC(x,y)* e se utiliza das metaclasses *WorkProductDefinitionRelationship* e *WorkProductDefinition*. Apenas para facilitar o entendimento, a seguir, mostra-se como é escrito o predicado *dependenciaMC(x,y)* no pacote *Method Content*:

$\forall z,x,y(workProductDefinition(x) \wedge workProductDefinition(y) \wedge (workProductDefinitionRelationship(z) \wedge relationType(z, \text{'dependency'}) \wedge source(z, x) \wedge target(z, y)) \rightarrow dependenciaMC(x, y))$

As regras de boa-formação formalizadas para o pacote *Method Content* são as seguintes:

Regra #4 – Um parâmetro de entrada e/ou saída (Default_TaskDefinitionParameter) de uma tarefa deve estar sempre associado a exatamente um produto de trabalho (WorkProductDefinition).

$\forall y\ (default\_TaskDefinitionParameter(y) \rightarrow \exists x\ (workProductDefinition(x) \wedge parameterType(y,x)))$

Regra #5 - Um produto de trabalho (WorkProductDefinition) não pode ser o "todo" em um relacionamento de composição se uma de suas "partes" já representa o seu "todo" em outro relacionamento de composição ou representa o seu "todo" pela transitividade da relação de composição.

$\forall x,y\ (composicaoMC(x,y) \rightarrow \neg\ composicaoMC(y,x))$

Regra #5 - Um produto de trabalho (WorkProductDefinition) não pode ser o "todo" em um relacionamento de agregação se uma de suas "partes" já representa o seu "todo" em outro relacionamento de agregação ou representa o seu "todo" pela transitividade da relação de agregação.

$\forall x,y\ (agregacaoMC(x,y) \rightarrow \neg\ agregacaoMC(y,x))$

Regra #5 - Um produto de trabalho (WorkProductDefinition) não pode depender de um produto de trabalho (WorkProductDefinition) que já é seu dependente.

$\forall x,y\ (dependenciaMC(x,y) \rightarrow \neg\ dependenciaMC(y,x))$

Regra #49 - Um produto de trabalho (WorkProductDefinition) não pode representar o "todo" e a "parte" em um relacionamento de composição.

$\forall x\ \neg\ composicaoMC(x,x)$

Regra #49 - Um produto de trabalho (WorkProductDefinition) não pode representar o "todo" e a "parte" em um relacionamento de agregação.

$\forall x\ \neg\ agregacaoMC(x,x)$

Regra #49 - Um produto de trabalho (WorkProductDefinition) não pode depender de si próprio.

$\forall x\ \neg\ dependenciaMC(x,x)$

Regra #7 - Um produto produto de trabalho (WorkProductDefinition) não pode ser "parte" em mais de um relacionamento de composição.

$\forall x,y,z\ (composition(x,y) \rightarrow \neg\ composition(x,z))$

Regra #11 – Todas as dependências de um produto de trabalho (WorkProductDefinition) devem estar conectadas como entrada nas suas tarefas (TaskDefinition) de produção.

$\forall$ x,y,t (workProductUse(x) $\wedge$ dependenciaMC(x,y) $\wedge$ tarefa-produzMC(t,x) $\rightarrow$ tarefa-consomeMC(t,y))

Regra #33 – Não deve existir mais de um papel (RoleDefinition) com o mesmo nome no repositório (Method Content).

$\forall x,y,n_1,n_2$ ((roleDefinition(x) $\wedge$ name(x, n_1)) $\wedge$ (roleDefinition (y) $\wedge$ name(y, n_2)) $\rightarrow n_1 \neq n_2$)

Regra #34 – Não deve existir mais de um produto de trabalho (WorkProductDefinition) com o mesmo nome no repositório (Method Content).

$\forall x,y,n_1,n_2$ ((workProductDefinition(x) $\wedge$ name(x, n_1)) $\wedge$ (workProductDefinition(y) $\wedge$ name(y, n_2)) $\rightarrow n_1 \neq n_2$)

Regra #35 – Um papel (RoleDefinition) não deve ser definido mais de uma vez como executor da mesma tarefa (TaskDefinition).

$\forall x,y,z$ (taskDefinition(x) $\wedge$ roleDefinition(y) $\wedge$ (defaultTaskDefinitionPerformer(z) $\wedge$ linkedTaskDefinition(z, x) $\wedge$ linkedRoleDefinition(z, y)) $\rightarrow \neg\exists w$ (defaultTaskDefinitionPerformer(w) $\wedge$ linkedTaskDefinition(w, x) $\wedge$ linkedRoleDefinition(w, y)))

Regra #36 – Um papel (RoleDefinition) não deve ser definido mais de uma vez como responsável pelo mesmo produto de trabalho (WorkProductDefinition).

$\forall x,y,z$((roleDefinition(x) $\wedge$ workProductDefinition(y) $\wedge$ (Defatult_ResponsabilityAssignment(r) $\wedge$ linkedRoleDefinition(r,x)$\wedge$linkedWorkProductDefinition(r,y)))$\rightarrow\neg\exists w$(Defatult_ResponsabilityAssignment (w) $\wedge$ linkedRoleDefinition(w,x) $\wedge$ linkedWorkProductDefinition(w,y)))

Regra #37 – Um produto de trabalho (WorkProductDefinition) não deve ser definido mais de uma vez como dependente do mesmo produto de trabalho (WorkProductDefinition).

$\forall x,y,z$(workProductDefinition(x)$\wedge$ workProductDefinition(y) $\wedge$ (workProductDefinitionRelationship(r) $\wedge$ source(r,x)$\wedge$ target(r,y)$\wedge$ relationType(r,'dependency'))$\rightarrow \neg\exists w$(workProductDefinitionRelationship (w) $\wedge$ source(w,x) $\wedge$ target (w,y) $\wedge$ relationType(r,'dependency')))

Regra #38 – Um produto de trabalho (WorkProductDefinition) não deve ser definido mais de uma vez como o todo, através de uma relação de composição, do mesmo produto de trabalho (WorkProductDefinition).

∀x,y,z(workProductDefinition(x)∧ workProductDefinition(y) ∧ (workProductDefinitionRelationship(r) ∧source(r,x)∧ target(r,y)∧ relationType(r,'composition'))→ ¬∃w(workProductDefinitionRelationship (w) ∧ source(w,x) ∧ target (w,y) ∧ relationType(r,'composition')))

Regra #39 – Um produto de trabalho (WorkProductDefinition) não deve ser definido mais de uma vez como o todo, através de uma relação de agregação, do mesmo produto de trabalho (WorkProductDefinition).

∀x,y,z(workProductDefinition(x)∧ workProductDefinition(y) ∧ (workProductDefinitionRelationship(r) ∧source(r,x)∧ target(r,y)∧ relationType(r,'agregation'))→ ¬∃w(workProductDefinitionRelationship (w) ∧ source(w,x) ∧ target (w,y) ∧ relationType(r,'agregation')))

Regra #40 - Uma tarefa (TaskDefinition) não deve possuir dois ou mais parâmetros de entrada e/ou saída (Default_TaskDefinitionParameter) iguais.

∀x,y,z(taskDefinition(x) ∧ workProductDefinition(z) ∧ (default_TaskDefinitionParameter(y) ∧ direction(y,'out')∧ parameterType(y,z)) ∧ part-of(y,x) → ¬∃w (default_TaskDefinitionParameter(w) ∧ direction(w,'out') ∧ parameterType(w,z)) ∧ part-of(w,x))

6.4 Pacote Method Plugin

A única regra de boa-formação formalizada considerando os elementos do pacote *Method Plugin* foi a *Regra #47*, a qual é relacionada com o mecanismo de adaptação *Variability*. Os predicados criados para este pacote são relacionados com as metaclasses e relacionamentos exibidos no diagrama de classes da Figura 58.

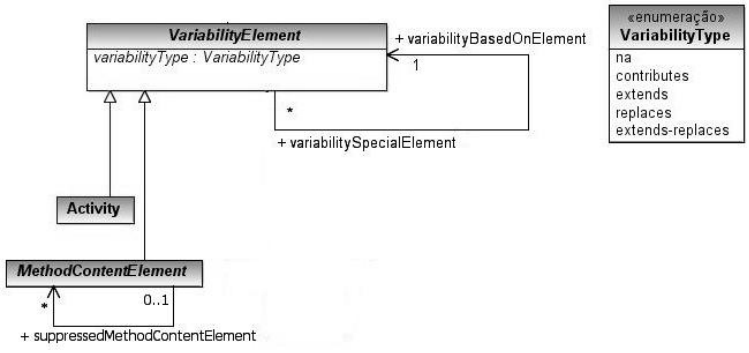

Figura 58 – Classes do Pacote *Method Plugin*

Os predicados definidos para formalizar a *Regra #47* foram os seguintes: *methodContentElement(x)* onde x é uma instância da metaclasse *MethodContentElement*; *variabilityBasedOnElement(x, y)* onde x e y podem ser representandos por instâncias da metaclasse *Activity* ou da metaclasse *MethodContentElement;* e *variabilityType (x,*

‘contributes’) onde x pode ser representando por uma instância da metaclasse *Activity* ou da metaclasse *MethodContentElement* e ‘*contributes’* refere ao valor que a propriedade *variabilityType* pode assumir. *VariabilityType* também assume os valores *‘na’, ‘extends’, ‘replaces’* e ‘*extends-replaces’*.

Com base nos predicados acima, a formalização da *Regra #47* foi definida e é apresentada a seguir:

Regra #47 – Elementos do repositório de conteúdo (Method Content) e atividades (Activity) não podem variar seu próprio conteúdo.

$\forall x$ *(activity(x)* $\wedge$ *(variabilityType(x, ‘contributes’)* $\vee$ *variabilityType(x, ‘replaces’)* $\vee$ *variabilityType (x, ‘extends’)* $\vee$ *variabilityType(x, ‘extends-replaces’))* $\rightarrow \neg$ *variabilityBasedOnElement(x, x))*

$\forall x$ *(methodContentElement(x)* $\wedge$ *(variabilityType(x, ‘contributes’)* $\vee$ *variabilityType(x, ‘replaces’)* $\vee$ *variabilityType (x, ‘extends’)* $\vee$ *variabilityType(x, ‘extends-replaces’))* $\rightarrow \neg$ *variabilityBasedOnElement(x, x))*

6.5 Considerações Finais

Este capítulo descreveu um conjunto de predicados e axiomas na linguagem de lógica de primeira ordem, bem como apresentou a formalização das regras de boa-formação nessa linguagem.

O próximo capítulo deste trabalho, apresenta uma avaliação do metamodelo sSPEM 2.0 e das regras de boa-formação, que foi conduzida utilizando-se do processo OpenUP.

7 AVALIAÇÃO

A avaliação da solução proposta nesta tese foi feita com base em testes analíticos verificando-se os resultados produzidos pela aplicação das regras de boa-formação para consistência sobre cenários simulados.

Para realização desta avaliação foi desenvolvido um protótipo de ferramenta que suporta automaticamente o uso do metamodelo sSPEM 2.0 e das regras de boa-formação para consistência. Nas seções seguintes, serão apresentados cinco cenários baseados no processo de software *OpenUP* [Ope10]. O *OpenUP* é uma parte do *Eclipse Process Framework* - EPF que representa um *framework* de processo *open source* desenvolvido dentro da *Eclipse Foundation*. Este processo é uma versão simplificada do RUP que aplica uma abordagem iterativa e incremental dentro de um ciclo de vida estruturado. *O OpenUP* foi escolhido para realização dessa avaliação porque esse processo está em conformidade com o metamodelo sSPEM 2.0. Além disso, por fazer parte do EPF, esse processo vem sendo utilizado pela comunidade de software livre, assumindo certa relevância como processo de engenharia de software.

O restante deste capítulo é organizado da seguinte forma: na Seção 7.1 o protótipo desenvolvido é resumidamente descrito; na Seção 7.2 os cenários de testes são descritos e as regras de boa-formação para consistência são utilizadas, sendo os resultados desta utilização descritos de forma analítica; e, por fim, a Seção 7.3 apresenta as conclusões do capítulo.

7.1 Protótipo

Tendo em vista que a quantidade de metaclasses e regras de boa-formação para consistência do metamodelo sSPEM 2.0 é elevada e que processos de software, geralmente, possuem uma grande quantidade de informações, considera-se que a falta de suporte automatizado torna o uso da solução proposta por esta pesquisa bastante difícil. Isso porque, todas as análises implícitas nas regras de boa formação para consistência, tanto na atividade de definição dos processos quanto nas atividades de adaptação desses processos, constituiriam uma tarefa exaustiva com maior probabilidade de erros. Baseado nesse contexto, foi desenvolvido um protótipo de ferramenta para auxiliar a utilização do metamodelo sSPEM 2.0 e das regras de boa-formação para consistência. O protótipo, que é denominado *sSPEM Tool*, foi desenvolvido utilizando

como principais tecnologias o *Eclipse Modeling Framework* - EMF e o *IBM Rational Software Modeler* - RSM.

7.1.1 Tecnologias utilizadas

7.1.1.1 Eclipse Modeling Framework (EMF)

O *Eclipse Modeling Framework* (EMF) [Ecl08] é um arcabouço para desenvolvimento de modelos baseados no MOF, com suporte à geração de código através de linguagem Java. Além disso, o EMF provê persistência dos dados utilizando XMI, linguagem padrão para definição de troca de dados da UML. Quanto à geração do código, o EMF cria a base do modelo, constituída de Java Interfaces e sua implementação, utilizando Java Classes.

Já com relação a geração da Interface Gráfica, destaca-se que no EMF ela é feita baseada no padrão *Adapter* [Gam93], possibilitando visualização e edição do modelo através de um editor básico. Os modelos definidos pelo EMF são baseados no *ECore*. Este pode ser considerado uma implementação simples do MOF, sendo utilizado para definir a sintaxe e a semântica dos modelos manipulados pelo EMF. Isso implica na capacidade de construir elementos como, por exemplo, classes, interfaces e associações [Moo04].

Basicamente, os modelos criados a partir do EMF são formados por *EClasses* (elementos equivalentes às classes definidas no MOF), as quais podem ter *EAttributes* (elementos equivalentes as propriedades definidas no MOF) e, por fim, é possível estabelecer relacionamento entre as *EClasses* (esses relacionamentos são equivalentes à associações definidas no MOF).

Destaca-se também que o *plugin* EMF possui um *framework* de validação que checa a integridade dos modelos instanciados nesse *plugin*.

7.1.1.2 IBM Rational Software Modeler (RSM)

O *IBM Rational Software Modeler* (RSM) é uma ferramenta CASE desenvolvida pela *IBM Rational Software* que foi construída para facilitar a modelagem de sistemas Orientados a Objetos utilizando UML. O foco dessa ferramenta é sua capacidade de permitir a modelagem visual dos conceitos definidos pela *Model Driven Development* (MDD) [Atk03].

Dessa forma, a ferramenta em questão possibilita: (1) modelagem específica de domínio utilizando UML; (2) criação de diagramas simples, com base em uma modelagem visual e assistente de criação; (3) suporte à customização e às transformações de modelos; (4) criação de UML *Profiles*; (5) gera documentos e relatórios a partir dos modelos UML; e (6) suporte a desenvolvimento descentralizado, facilitando a decomposição, comparação e *merge* do modelo. Além disso, essa ferramenta possui também suporte para construção de modelos do tipo *Ecore*, sendo compatível com o EMF. Devido a esse fato, o desenvolvimento do modelo feito no RSM pode ser importado pelo EMF, permitindo, então, a geração de código e implementação do modelo. Tal característica facilita os testes sobre a modelagem, uma vez que pode ser feita automaticamente.

7.1.2 Visão Geral de *sSPEM Tool*

O protótipo *sSPEM Tool* implementa funcionalidades para a definição de processos de software a partir do metamodelo sSPEM 2.0 e das regras de boa-formação para consistência. Também permite a adaptação desses processos através dos mecanismos de adaptação *usedActivity* e *supressedBreakdownElement*. Fundamentamente, para definição e adaptação dos processos, os quatro principais pacotes do metamodelo sSPEM 2.0 foram implementados: *Method Content*, *Process Structure*, *Process With Methods* and *Method Plugin*.

A implementação do *Method Plugin* permite a criação de bibliotecas (*Method Library*). Essas bibliotecas são formadas essencialmente por conteúdo e processos implementados, respectivamente, pelos pacotes *Method Content* e *Process Structure*. Por fim, a implementação do pacote *Process with Methods* permite que os elementos definidos no *Method Content* sejam utilizados em um ou mais processos de software - *Process Structure*. É importante considerar que a utilização de qualquer elemento do repositório de conteúdo (*Method Content*) em um processo de software acontece conforme foi explicado na Seção 5.5 deste estudo.

Para a verificação de consistência, tanto do repositório de conteúdo quanto dos processos de software, foi implementado uma funcionalidade de validação em *sSPEM Tool*. Para fazer isso, o *framework* de validação do *plugin* EMF com a inclusão de todas as regras de boa-formação definidas nesta pesquisa.

Por fim, as funcionalidades específicas para adaptação de processos foram implementadas com base nos mecanismos *usedActivity* e *supressedBreakdownElement*,

os quais permitem que processos de software sejam criados, reutilizando partes de um ou mais processos já definidos.

7.1.3 Arquitetura de sSPEM Tool

A arquitetura de *sSPEM Tool* é dividida em camadas funcionais de acordo com a Figura 59.

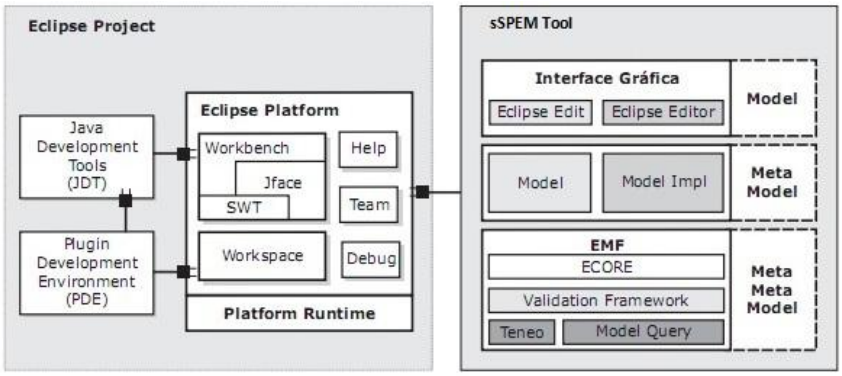

Figura 59 - Arquitetura de *sSPEM Tool*

Meta-Metamodelagem

O *Eclipse Modeling Framework* provê funcionalidades de infraestrutura reutilizáveis e, por se constituir basicamente do *ECore*, fornece diversas funcionalidades de modelagem, tais como: persistência em XMI ou Objeto Relacional, validação de integridade de modelo e gerência dos dados do modelo com utilização de transação.

Metamodelagem

A camada de modelo possui toda a implementação do metamodelo sSPEM 2.0 que foi feita com a utilização de linguagem Java. Nessa camada estão os contratos e restrições, feitos a partir de Java Interfaces.

Interface Gráfica

A interação entre a ferramenta e o seu usuário é realizada através desta camada. Toda a interface gráfica utiliza *SWT* e baseia-se nos conceitos de *Views*, *Perspectives* e *Wizards*, assim como nos encontrados na plataforma Eclipse. Isso ocorre pelo fato da ferramenta *sSPEM Tool* funcionar como um *plugin* para o Eclipse, estendendo suas funcionalidades.

7.1.4 Funcionalidades

As principais funcionalidades do *sSPEM Tool* são a definição e a adaptação de processos de desenvolvimento de software. Adicionalmente, uma funcionalidade para validação de consistência dos processos de software, também encontra-se disponível neste protótipo.

7.2 Cenários de Teste

Nesta seção são apresentados os cenários desta avaliação. Como já exposto, todos os cenários utilizam informações do processo *OpenUP*. Contudo, outras informações são introduzidas nesse processo para contemplar os novos elementos do metamodelo sSPEM 2.0. Tais informações são relacionadas com: (1) opcionalidade de elementos; (2) produtos de trabalho internos *x* externos; e (3) dependências entre produtos de trabalho. Além das informações acima, são incluídas algumas situações-problema que representam inconsistências em um processo de software. O objetivo das inserções é demonstrar que as regras de boa-formação para consistência identificam tais inconsistências.

Todas as informações acima e situações-problema são incluídas nos cinco cenários a seguir. Comentários explicando cada etapa desta avaliação são apresentados para facilitar a compreensão.

Durante a avaliação a identificação das regras de boa-formação será realizada pelos seus respectivos identificadores, conforme apresentado no Capítulo 4. Tendo em vista que *sSPEM Tool* foi implementado utilizando-se da língua inglesa, os identificadores das regras de boa-formação tiveram a palavra *Regra* traduzida para o inglês. Assim, por exemplo, a regra de boa-formação *Regra* #1 é identificada em *sSPEM Tool* como *Rule #1*.

7.2.1 Inclusão do *OpenUP* em *sSPEM Tool*

Para iniciar a criação dos cenários de avaliação foi necessário a inclusão do processo *OpenUP* em *sSPEM Tool*. Para realizar essa inclusão, os passos definidos no guia para definição de um processo de software que reutiliza conteúdo de um repositório (*Method Content*) foram seguidos (conforme Capítulo 5).

Os referidos passos são: (1) criar uma nova biblioteca (*Method Library*); (2) criar um *Method Content* para a biblioteca e (3) criar um *ProcessPackage* (definindo o atributo *isAuthoring* igual a *true*). A Tabela 16 mostra os tipos e quantidade de elementos criados para o *Method Content* e o *ProcessPackage* da biblioteca em questão. A tabela também expõe as metaclasses do metamodelo sSPEM 2.0 utilizadas para criação desses elementos. Essas metaclasses foram escolhidas, respeitando a especificação textual do SPEM 2.0 [Omg07a].

Tabela 16 - Elementos criados no *Method Content* e *Process Structure* na criação do processo *OpenUP*

Biblioteca (*Method Library*)			
Method Content		***Process Package***	
Elemento de Processo	Classe do Metamodelo	Elemento de Processo	Classe do Metamodelo
19 Tarefas	*TaskDefinition*	1 Processo	*Activity*
17 Produtos de Trabalho	*WorkProductDefinition*	4 Fases	*Activity*
7 Papéis	*RoleDefinition*	4 Iterações	*Activity*
-	-	19 Atividades	*Activity*
-	-	57 Tarefas	*TaskUse*
-	-	17 Produtos de Trabalho	*InternalUse*
-	-	7 Papéis	*RoleUse*

Embora não esteja explícito na Tabela 16, destaca-se que os elementos em negrito (localizados nas primeiras linhas) dessa tabela foram inicialmente definidos através de uma única instância das metaclasses *MethodLibrary*, *MethodContent* e *ProcessPackage*. Apenas para o caso da metaclasse *ProcessPackage,* outras instâncias desse elemento foram criadas durante a presente avaliação. Isso foi feito, pois, de acordo com o guia para definição e adaptação de processos, cada processo de software definido a partir do metamodelo sSPEM 2.0 deve ser representado por uma instância da metaclasse *ProcessPackage*.

Além disso, é definido, no mesmo guia, que, para cada instância de *ProcessPackage,* deverá ser informado, através do atributo *isAuthoring,* se o processo de software sendo criado está na fase de definição (*isAuthoring = 'true'*) ou irá sofrer atividades de adaptação (*isAuthoring =' false'*).

Vale destacar também que, para inclusão do processo *OpenUP*, todas as tarefas, produtos de trabalho e papéis definidos no *Process Package* foram criados e associados aos elementos do *Method Content*, ou seja, o conteúdo definido para a biblioteca foi utilizado na criação do processo. O número elevado de tarefas no *Process Package* se dá devido ao fato de uma mesma tarefa ser executada várias vezes no ciclo de vida do desenvolvimento.

Além dos elementos relacionados na Tabela 16, outros foram criados tanto no *Method Content* quanto no *Process Package* durante a inclusão do *OpenUp*. Estes elementos representam os relacionamentos do processo. Várias instâncias, por exemplo, da metaclasse *ProcessPerformer* e *Default_ProcessPerformer,* foram criadas para, respectivamente, no *Process Package* e *Method Content*, representar a associação entre as tarefas e os papéis. Outro exemplo são as instâncias da metaclasse *WorkSequence* criadas no *Process Package* , as quais foram utilizadas para definir o sequenciamento entre as fases do *OpenUp* e entre algumas das atividades que compõem as iterações desse processo.

A Figura 60 mostra o processo *OpenUP* incluído em *sSPEM Tool*. Como pode ser visto, a GUI é dividida em várias janelas:

- em **a)** está a janela *Package Hierarchi View* que provê uma iteração com os projetos existentes e seus conteúdos internos, apresentados de forma hierárquica.

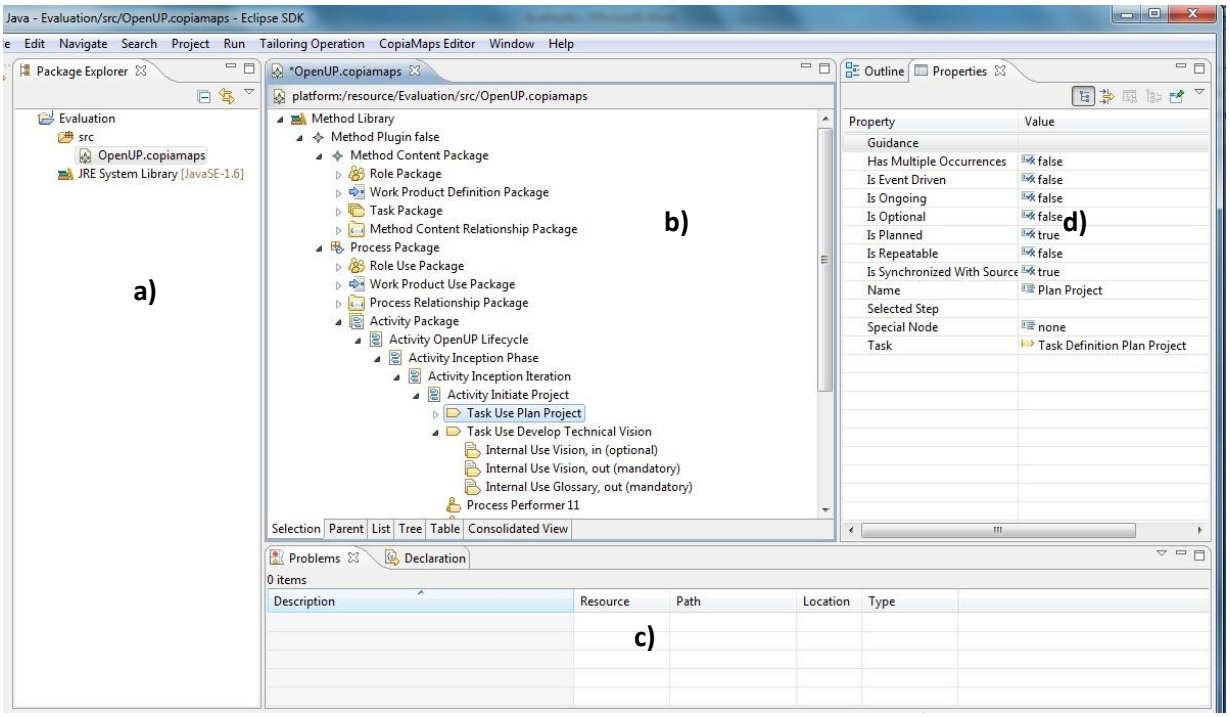

Figura 60 - *OpenUP* incluído em *sSPEM Tool*

- em **b)** está a janela que disponibiliza as funcionalidades da biblioteca. A biblioteca é aberta através de uma *View* amigável, permitindo o manuseio da biblioteca a partir de seis modos diferentes, inclusive em árvore, constituindo a linguagem abstrata, entretanto, sem uma notação em diagramas. Na Figura 60, a forma de visualização selecionada é a *Selection*. Esta é a visão mais utilizada em *sSPEM Tool*. Nela, os elementos de processo podem ser instanciados pelo botão direito do mouse. Por exemplo, para incluir uma nova

Tarefa no *Method Content* basta clicar com o botão direito do mouse sobre o pacote de tarefas (*Task Package*) e selecionar a opção *New Child – Task.*

- em **c)** é mostrada a janela *Problems View* que permite ver os erros encontrados na biblioteca durante a definição e adaptação dos processos. Essa janela também disponibiliza a localização dos problemas encontrados durante a validação do modelo. Além disso, a mesma janela é uma das formas em *sSPEM Tool* de visualização do resultado da validação realizada com as regras de boa-formação para consistência do metamodelo sSPEM 2.0. Outras formas de visualização serão exploradas em outros pontos desta avaliação.
- por fim, em **d)** está a *Properties View*. Essa janela permite a iteração com os atributos e relacionamentos existentes entre os elementos do modelo. Também nessa janela é possível, por exemplo, configurar que um elemento de processo *Use* está associado a um elemento *Definition* do *Method Content*, ou seja, através da janela *Properties View* elementos definidos como conteúdo reutilizável na biblioteca podem ser utilizados em um processo de software.

O objetivo da apresentação da Figura 60 foi mostrar uma visão geral das telas de *sSPEM Tool* e, também o processo, *OpenUP* incluído nesse protótipo. Contudo, neste momento, nenhum procedimento de validação foi ainda realizado.

O próximo passo na inclusão do *OpenUP* é justamente executar o *framework* de validação para entender seu funcionamento. Para realizar esse procedimento, é necessário selecionar o elemento que se deseja validar e acessar a opção *Validate* do botão direito do mouse.

A Figura 61 mostra a validação do papel *Tester*. Como pode ser visto, nessa figura, nenhuma inconsistência foi encontrada e o elemento foi validado sem a indicação de erros. No exemplo exibido na Figura 61, o *framework* de validação avaliou realmente apenas o elemento *papel*. Isso ocorreu pois esse tipo de elemento não é composto por nenhum outro elemento. Como já explicado durante este estudo, o *framework* de validação tem um funcionamento em cascata, ou seja, ele procura por inconsistências existentes no interior de um elemento selecionado, portanto, seus relacionamentos fortes, ou seja, do tipo composição, serão validados. Quanto mais alta for a hierarquia do elemento validado, mais inconsistências existentes em seus elementos filhos serão reveladas.

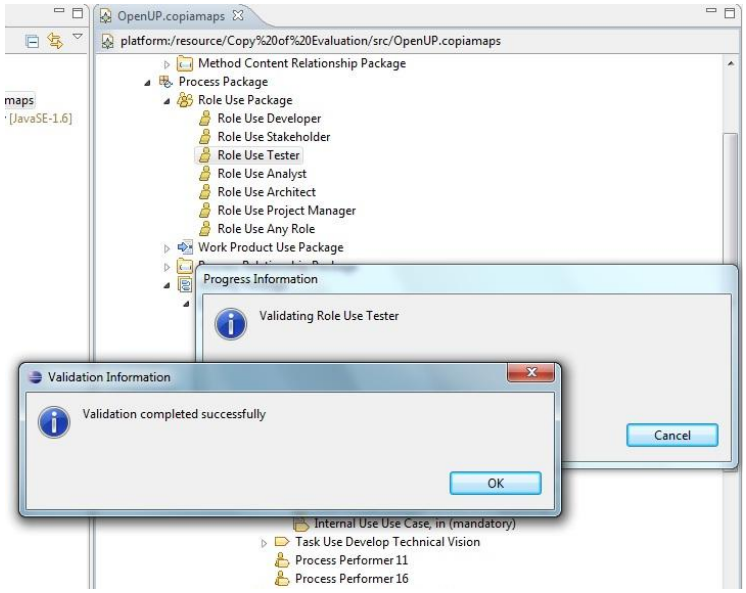

Figura 61 - Exemplo de validação em *sSPEM Tool*

Para demonstrar um exemplo mais completo de validação e mostrar um resultado que indique erros, a Figura 62 exibe um novo exemplo de validação, agora na atividade *Testing Validate.* Essa atividade não pertence ao *OpenUP* e foi criada apenas para demonstrar com mais clareza como funciona o *framework* de validação. Logo após a validação ter sido executada, este elemento e todo seu conteúdo foram apagados em *sSPEM Tool*.

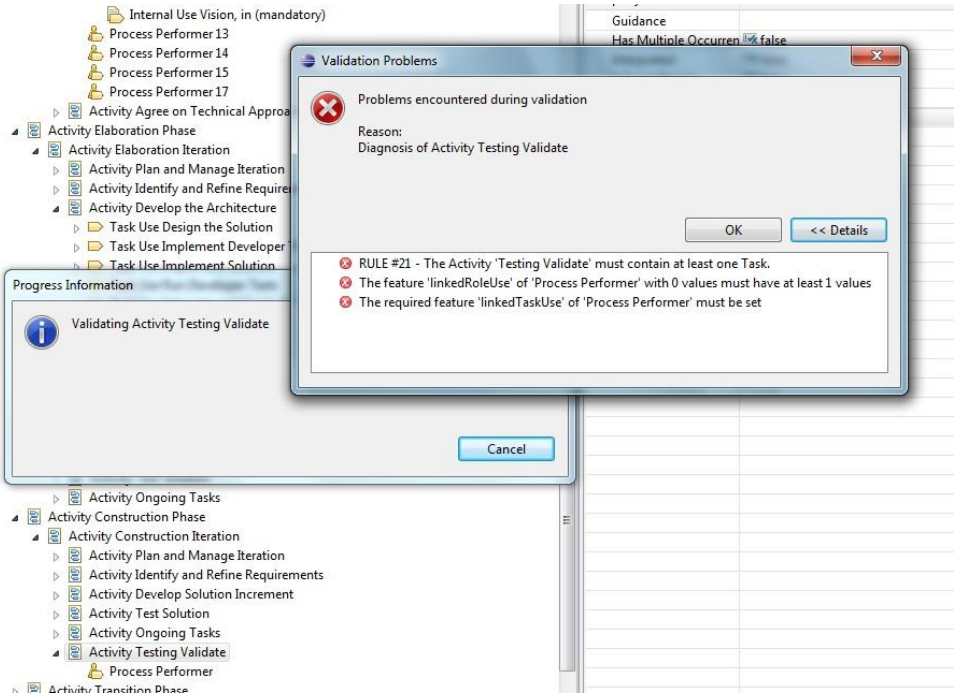

Figura 62 - Validação da *Activity Testing Validate*

A Figura 62 mostra ainda que, para a atividade *Testing Validate,* foi criado apenas um elemento do tipo *ProcessPerformer*. Dessa forma, tanto a atividade *Testing Validate* quanto o *ProcessPerformer* foram validados. A janela *Validation Problems* relata os

problemas relacionados com tais elementos. O resultado mostra a *Rule #21* e dois erros os quais não possuem a identificação proposta para as regras desta tese.

Em verdade, os dois últimos erros listados na janela em questão foram erros detectados no próprio metamodelo, ou seja, são erros provenientes de restrições incluídas utilizando a própria UML. Como se sabe, a criação da biblioteca, na verdade, é uma instância do metamodelo sSPEM 20 apresentado no Capítulo 4. Por isso, ao criar uma biblioteca deve-se obedecer às restrições e regras existentes no metamodelo.

Como pode ser observado na Figura 62, os problemas de validação estão dispostos através de mensagens que possuem um formato típico. Contudo, embora as mensagens sejam informativas, o formato em que se apresentam não traduz a localização exata do problema. Dessa forma, é necessário completar a quantidade de informações. Para isso, basta completar a validação, clicando em *ok.* Nesse momento, os erros são movidos para janela *Problems View*, conforme mostrado na Figura 63. Nessa janela, cada elemento com problema pode ser localizado facilmente através de um duplo clique sobre cada erro. Além disso, a janela *Problems View* contabiliza o número de problemas encontrados durante a validação.

Uma vez finalizada a inclusão do *OpenUp* em *sSPEM Tool* e explicado o funcionamento do *framework* de validação, parte-se para a avaliação dos cenários de testes.

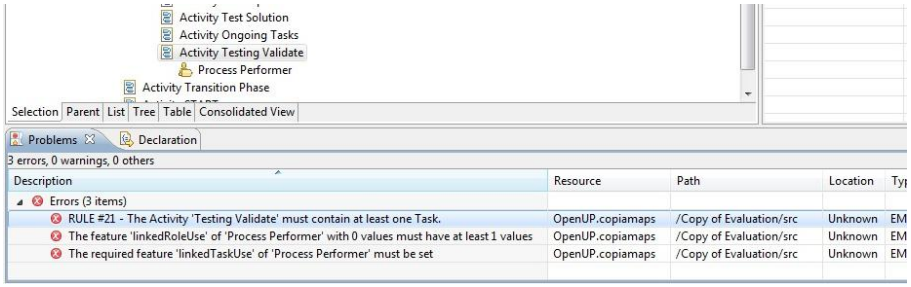

Figura 63 - Janela *Problems View* em *sSPEM Tool*

7.2.2 Cenário 1

O primeiro cenário utilizado nesta avaliação é o processo *OpenUP* na sua forma original. Para esse cenário, nenhuma informação específica do metamodelo sSPEM 2.0 foi utilizada. O objetivo aqui foi verificar se o processo avaliado, quando não submetido a nenhuma mudança, é consistente.

É importante considerar que nem todas as regras de boa-formação para consistência definidas nesta pesquisa podem ser avaliadas nesse cenário, uma vez que como dito acima, nenhuma informação específica do metamodelo sSPEM 2.0 foi incluída até o presente momento no processo *OpenUP*. Assim, sabe-se que alguns tipos de inconsistências não poderão ser identificados e, por conseqüência, algumas premissas não poderão ser analisadas.

Baseado no contexto acima, as premissas e regras de boa-formação consideradas para este cenário são: **P #3** (*Rule #3*); **P #4** (*Rule #5*, *#7* e *#49*); **P #6** (*Rule #8* e *#24*); **P #7** (*Rule #4, #9, #10* e *#21*); **P #9** (*Rule #20*); **P #10, P #11 e P #12** (*Rule #6, #12, #13, #14, #15, #16, #17, #18, #41, #42, #43*, *#44, #45* e *#50*); **P #13** (*Rule #22*); **P #14** (*Rule #48*); **P #15, P #16, P #17, P #18, P #19** e **P #20** (*Rule #23, #25, #26, #27, #28, #29, #30, #31 e #32*); e **P #21 e P #22** (*Rule #33, #34, #35, #36, #37, #38, #39* e *#40*). Ao final desta seção todas as premissas e regras de boa-formação avaliadas neste cenário são relacionadas com objetivo de verificar quais delas foram atendidas e quais não foram atendidas (gerando inconsistências no processo).

O primeiro passo para avaliação do Cenário 1 envolve e execução do *framework* de validação sobre esse processo. Na prática isto implica na validação do elemento *Process Package*, uma vez que todos os demais elementos do processo estão a ele vinculados por um relacionamento de composição. A Figura 64 exibe os erros retornados pelo protótipo na janela *Problems View*. A seguir, segue-se uma interpretação dos resultados desta validação.

Observa-se, na Figura 64 (janela *Properties View*) que a quantidade de erros retornados pelo protótipo durante a primeira avaliação foi igual a 19 e a quantidade de *warnings* igual a 7. Analisando, inicialmente, os erros retornados, verifica-se que grande parte deles (17 do total de erros), é relacionada com a regra de boa-formação *Rule #20*. A regra de boa-formação em questão estabelece que todo produto de trabalho considerado interno (*InternalUse*) deverá ser produzido por, pelo menos, uma tarefa no processo de software antes de ser consumido. Além disso, é preciso que a tarefa que gera o produto de trabalho tenha caminho definido, através das estruturas de sequenciamento até as tarefas que consomem esse produto de trabalho.

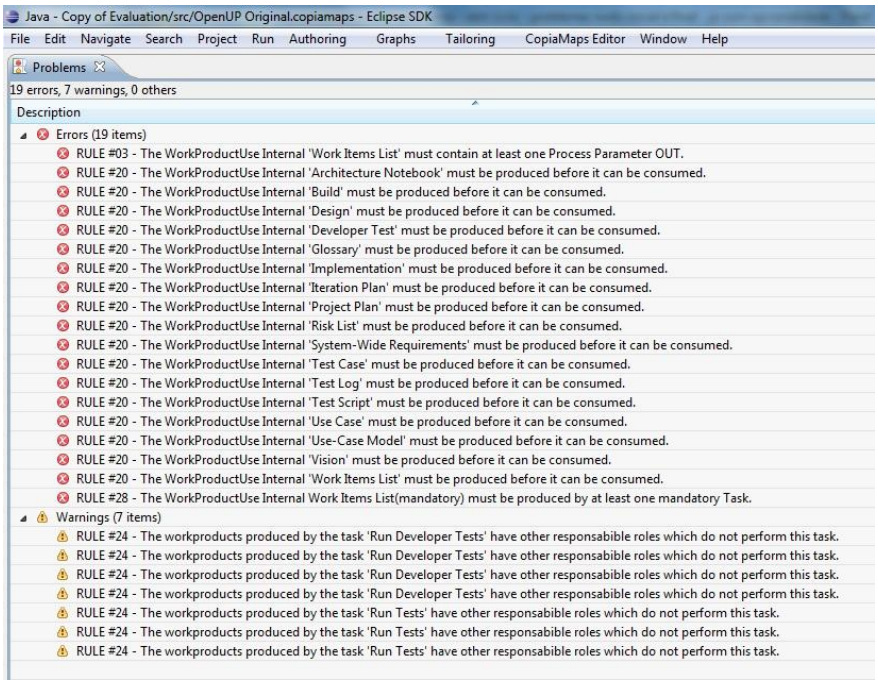

Figura 64 - Validação do processo *OpenUP* original

No *OpenUp,* todos os produtos de trabalho são internos e apresentam inconsistências relacionadas com o caminho entre as tarefas que os produzem e as tarefas que os consomem. Isso ocorre por duas razões: (1) o sequenciamento entre tarefas não é completo no *OpenUP*, ou seja, nem todas as tarefas deste processo são seqüenciadas e (2) muitos produtos de trabalho são realmente consumidos antes de serem produzidos.

Para entender melhor as inconsistências relacionadas acima, é apresentado, na Figura 65, um detalhamento das atividades e tarefas de uma das iterações do processo *OpenUp*.

A figura exibe na parte superior a iteração *Inception* que é a primeira iteração executada no processo. Nessa área da Figura 65, as atividades dessa iteração, as tarefas pertencentes a cada atividade e as informações de sequenciamento que existem entre as atividades também são mostradas. No centro e parte inferior da Figura 65, respectivamente, as tarefas pertencentes à atividade *Initiate Project* e *Plan and Manage Iteration* são detalhadas em termos de entradas, saídas (produtos de trabalho) e papéis. Por restrições de espaço, alguns papéis que participam da execução de algumas tarefas foram omitidos. O papel exibido foi sempre o que possui alguma responsabilidade na tarefa sobre algum produto de trabalho e/ou que é considerado no *OpenUP* como principal executor da tarefa.

Analisando apenas as informações do cenário acima, faz-se possível identificar que existe sequenciamento entre 3 atividades e que uma quarta atividade (*Plan and Manage Iteration*) não apresenta sequenciamento, podendo ser executada a qualquer ponto na iteração. Para facilitar a visualização desse fluxo, um diagrama de atividades da UML é exibido na Figura 66.

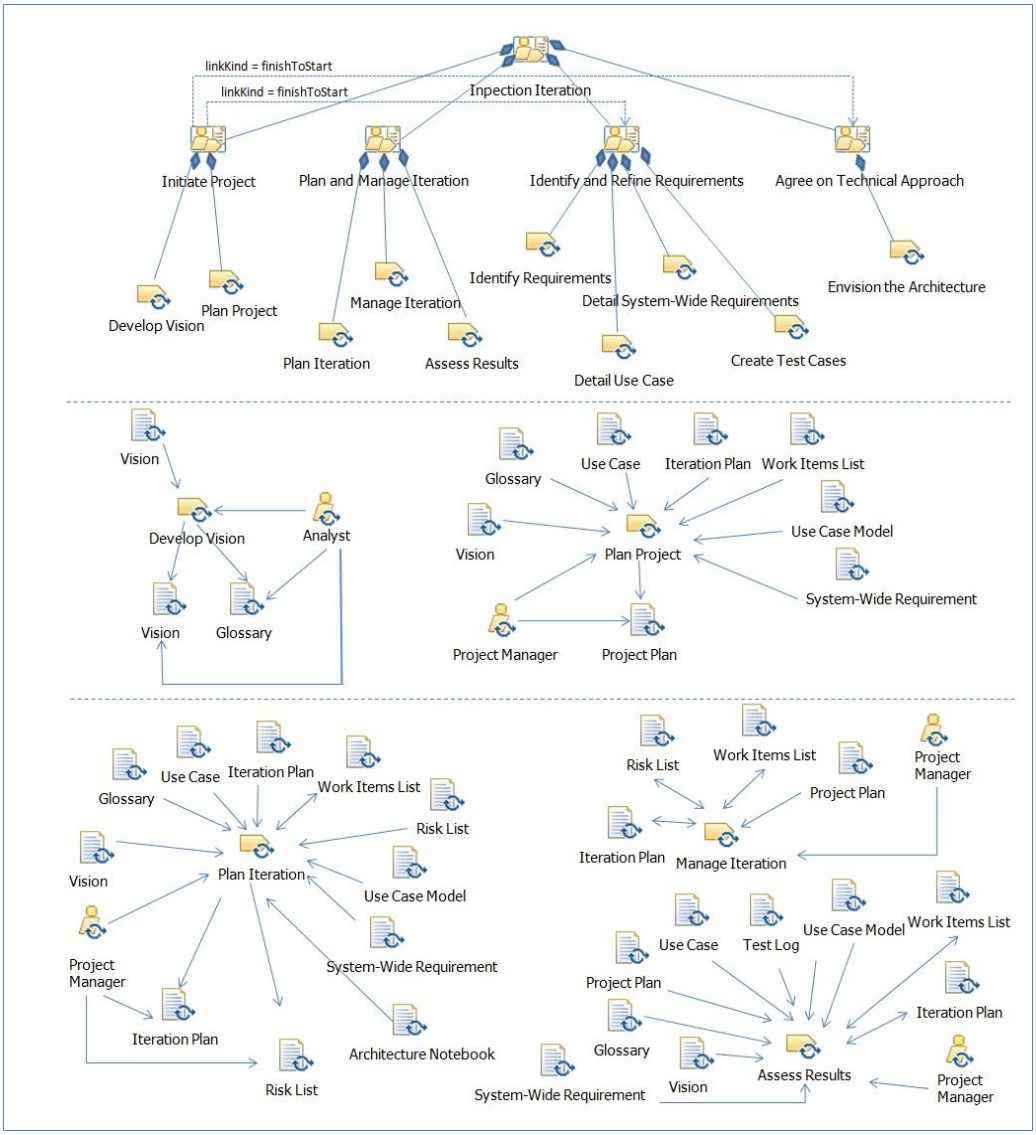

Figura 65 - Detalhamento da Iteração *Inception Iteration* do *OpenUP*

Considerando que Inception Iteration é a primeira iteração do processo OpenUP, observa-se que as primeiras tarefas a serem executadas no processo são as tarefas da atividade Initiate Project e Plan and Manage Iteration. Também se nota que essas tarefas já consomem vários produtos de trabalho que ainda não foram produzidos no processo, o que gerou muitas das inconsistências relacionadas com a regra de boa-formação *Rule #20*.

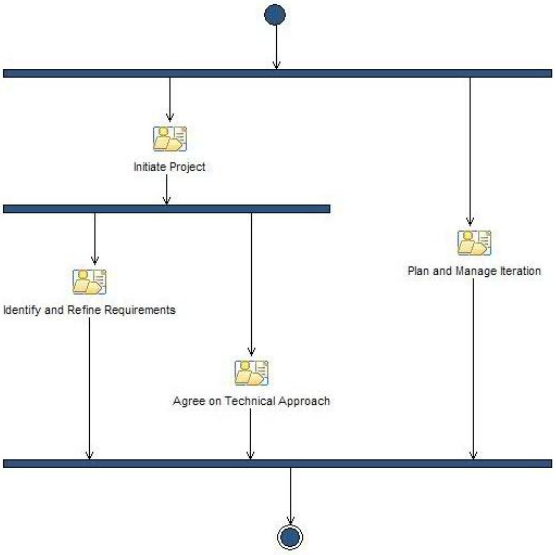

Figura 66 - Diagrama de atividades da Iteração *Inception Iteration*

Especificamente, dos 17 erros retornados por essa regra, 10 deles estão relacionados com as tarefas das atividades *Initiate Project* e *Plan and Manage Iteration.* Existem casos onde o produto de trabalho foi produzido (produtos de trabalho *Iteration Plan*, *Vision* e *Glossary*), mas não existe caminho entre a tarefa produtora e a tarefa consumidora; e, existem casos onde o produto de trabalho realmente ainda não foi produzido. Como exemplo para este segundo caso, verifica-se os produtos de trabalho *Use Case, Use Case Model, System-Wide Requirement* e *Work Items List* que são consumidos pela tarefa *Plan Project,* mas não são produzidos por nenhuma das tarefas iniciais do processo.

Prosseguindo com a avaliação das inconsistências, verifica-se que outros dois erros relacionados com as regras de boa-formação *Rule #28* e *Rule #*3 também foram apresentados. Ambos os erros são apontados para o produto de trabalho *Work Items List* e indicam, respectivamente, que esse produto de trabalho não é produzido por nenhuma tarefa no processo e não está associado a nenhuma tarefa obrigatória. Em verdade, os dois erros foram apontados pelo fato do produto de trabalho *Work Items List* não ser produzido por nenhuma tarefa no processo *OpenUP*, mas, mais especificamente, a regra *Rule #3* identificou que a tarefa produtora desse produto de trabalho precisa ser obrigatória no processo, já que tal produto de trabalho foi definido como obrigatório. Isso aconteceu porque, embora esse cenário não contemple as informações incluídas no metamodelo sSPEM 2.0 (como, por exemplo, as opcionalidades dos elementos para o processo), elas já existem em *sSPEM Tool* e, para alguns casos, são de cadastramento

obrigatório, como é o caso de informações sobre opcionalidade. Assim, todos os elementos do *OpenUP* foram incluídos como obrigatórios no protótipo.

Além dos erros retonados pela validação feita no *OpenUP,* verifica-se também que 7 *warnings* foram identificados. *Warnings* não representam inconsistências em um processo de software e não podem causar danos à execução das tarefas. No caso específico da avaliação deste cenário, todos os *warnings* identificados estão relacionados com a relação *tarefas x produtos de trabalhos x papéis*. O que ocorre é que, em todos os casos, mais de um responsável foi definido para os produtos de trabalho e existem casos onde nem todos os responsáveis por um determinado produto de trabalho foram associados às suas tarefas de produção. Como a regra de boa-formação *Rule #24* estabelece que, pelo menos, um responsável pelo produto de trabalho deve estar associado a todas as suas tarefas de produção a condição foi atendida. Contudo, os alertas existem e são de caráter informativo.

Na Tabela 17 é apresentado o resultado da avaliação indicando quais regras de boa-formação foram atendidas e não atendidas neste Cenário.

Tabela 17 - Premissas e regras analisadas no Cenário 1

Premissas e Regras Atendidas		**Premissas e Regras Não Atendidas**	
Premissa	**Regra de Boa-Formação**	**Premissa**	**Regra de Boa-Formação**
P #4	*Rule #5, #7 e #49*	**P #3**	*Rule #3*
P #6	*Rule #8 e #24*	**P #9**	*Rule #20*
P #7	*Rule #4, #9, #10 e #21*	**P #15** **P #16** **P #17** **P #18** **P #19** **P #20**	*Rule #28*
P #10 **P #11** **P #12**	*Rule #6, #12, #13, #14, #15, #16, #17, #18, #41, #42, #43, #44, #45 e #50*	**-**	-
P #13	*Rule #22*	**-**	-
P #14	*Rule #48*	**-**	-
P #15 **P #16** **P #17** **P #18** **P #19** **P #20**	*Rule #23, #25, #26, #27, #29, #30, #31 e #32*	**-**	-
P #21 **P #22**	*Rule #33, #34, #35, #36, #37, #38, #39 e #40*	**-**	-

7.2.3 Cenário 2

O Cenário 2 avalia os aspectos de dependência entre os produtos de trabalho e as informações sobre opcionalidade em um processo de software. As premissas e regras

de boa-formação avaliadas nesse cenário são: **P #1, P #2** e **P #3** (*Rule #3*); **P #4** (*Rule #5*, *#7 #49*); **P #5** (*Rule #5*); **P #6** (*Rule #8* e *#24*); **P #7** (*Rule #4, #9, #10* e *#21*); **P #8** (*Rule #11, #19*); **P #9** (*Rule #19* e *#20*); **P #10, P #11 e P #12** (*Rule #6, #12, #13, #14, #15, #16, #17, #18, #41, #42, #43*, *#44, #45 e #50*); **P #13** (*Rule #22*); **P #14** (*Rule #48*); **P #15, P #16, P #17, P #18, P #19** e **P #20** (*Rule #23, #25, #26, #27, #28, #29, #30, #31 e #32*); e **P #21 e P #22** (*Rule #33, #34, #35, #36, #37, #38, #39* e *#40*).

No Cenário 2 novas informações foram incluídas no processo *OpenUP,* tendo como base o metamodelo sSPEM 2.0. Tais informações referem-se a dependências entre os produtos de trabalho e opcionalidade dos elementos deste processo.

Para incluir as dependências entre os produtos de trabalho do *OpenUP,* uma análise sobre a descrição de todas as tarefas do processo foi realizada. Uma vez que este processo não mapeia esse tipo de informação, esta foi a única forma de identificar as dependências existentes sobre os produtos de trabalho.

Para entender melhor como foi realizado o processo de mapeamento de dependências entre produtos de trabalho, pode-se tomar como exemplo a dependência encontrada entre os produtos de trabalho *Test Case x Test Script*. Nessa relação, foi definido que o produto de trabalho *Test Script* depende do produto de trabalho *Test Case*. Isso foi feito, pois, observando a descrição de ambos produtos de trabalho e também de suas tarefas produtoras, constatou-se que não é possível produzir *Test Script* no *OpenUP* sem produzir o produto de trabalho *Test Case*.

A Tabela 18 indica a relação de dependências mapeadas para todos os produtos de trabalho do *OpenUP*. Na coluna da esquerda é relacionado um produto de trabalho e, na coluna da direita, foram relacionadas suas dependências.

Tabela 18 - Relação de dependências entre os produtos de trabalho do *OpenUP*

Produto de Trabalho	Dependência(s)
Test Case	*Use Case*
Build	*Implementation*
Test Script	*Test Case*
Architecture Notebook	*System-Wide Requirements*
Design	*System-Wide Requirements* e *Use Case*
Implementation	*Desing, Use Case e System-Wide Requirements*
Test Log	*Build, Test Script* e *Implementation*

Depois de incluído as dependências acima, as informações sobre opcionadade precisaram também ser incluídas. Mais uma vez, um estudo sobre toda descrição do processo foi realizada. Neste momento, verificou-se que todas as tarefas e todos os

produtos de trabalho são obrigatórios no *OpenUP*, assim sendo, torná-los opcionais levaria a várias mudanças nesse processo. Então, para resolver essa situação viabilizando a avaliação das premissas e regras de boa-formação sobre opcionalidade, novas atividades, tarefas, papéis e produtos de trabalho foram incluídos. Desse modo, foi possível incluir algumas situações-problema no processo e avaliar os resultados do *framework* de validação.

A nova atividade incluída é composta de 3 novas tarefas, 2 novos produtos de trabalho e 2 novos papéis e refere-se a uma atividade de verificação e validação dos requisitos no processo. Embora o *OpenUP* não possua essa atividade, ela é bastante comum nos processos de desenvolvimento de software, tais como o RUP e o OPEN. Tal atividade foi incluída em todas as iterações onde existem atividades de documentação de requisitos. A Figura 67 detalha a atividade incluída em termos de tarefas e produtos de trabalho (consumidos, modificados e produzidos). Abaixo, a Figura 68 mostra os novos elementos incluídos em *sSPEM Tool*. É possível ver na Figura 68 que a atividade *Verify and Validate Requirements* foi incluída nas fases *Inception*, *Elaboration* e *Construction*, respectivamente, nas suas iterações *Inception Iteration*, *Elaboration Iteration* e *Construction Iteration.*

Figura 67 - Detalhamento da atividade incluída no *OpenUP*

Os únicos elementos que não foram mostrados na Figura 67 foram os novos papéis que desempenham as novas tarefas (iss porque eles foram incluídos no pacote de papéis). Esses papéis são *Client* e *Reviewer* que desempenham, respectivamente, as tarefas *Validade Requirements* e *Verify Requirements*. A tarefa *Validade Requirements* é também desempenhada pelo papel *Analyst* que é quem desempenha a nova tarefa *Ajust*

Requirements. O responsável pelos novos produtos de trabalho *Requirements Acceptance* e *Requirements Verification* é o papel *Analyst*.

Deve-se notar que o responsável pelo produto de trabalho *Requirements Verification* não é o papel que desempenha a tarefa de produção (*Verify Requirements*) desse produto de trabalho. Essa situação-problema foi incluída, propositalmente, no *sSPEM Tool* para verificar o resultado de sua avaliação no *framework* de validação.

Outra situação-problema incluída nos novos elementos foi referente à sua opcionalidade no processo de software. Foram definidos como opcionais o produto de trabalho *Requirements Acceptance* e o papel *Reviewer*. Isso introduz 2 novas inconsistências no processo que são explicadas a seguir, durante a análise dos resultados deste cenário.

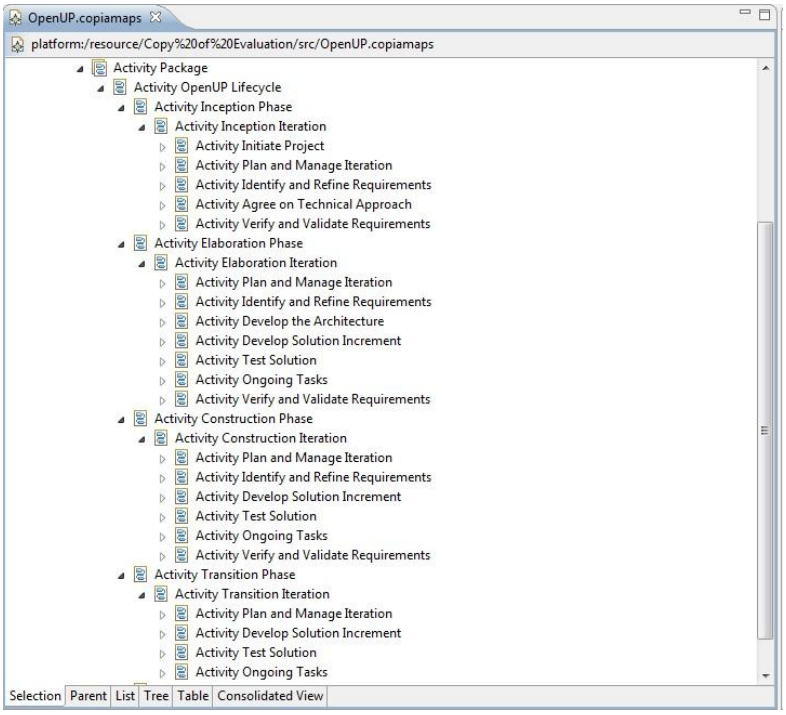

Figura 68 - Inclusão da atividade *Verify and Validate Requirements* em *sSPEM Tool*

Uma vez incluídas todas as informações necessárias no Cenário 2, o próximo passo desta avaliação foi executar o *framework* de validação sobre o elemento *Process Package*. O resultado disso foi o retorno de 41 erros e 7 *warnings*. A saída exibida na janela *Problems View* pode ser vista na Figura 69.

Visualiza-se que os 19 erros e 7 *warnigns* encontrados no Cenário 1 foram novamente retornados pelo *framework* de validação. Isso já era esperado uma vez que nenhum dos erros foi consertado até o momento, tendo sido apenas incluídas novas

informações. Os novos erros totalizam 22 novas inconsistências para o processo *OpenUP*.

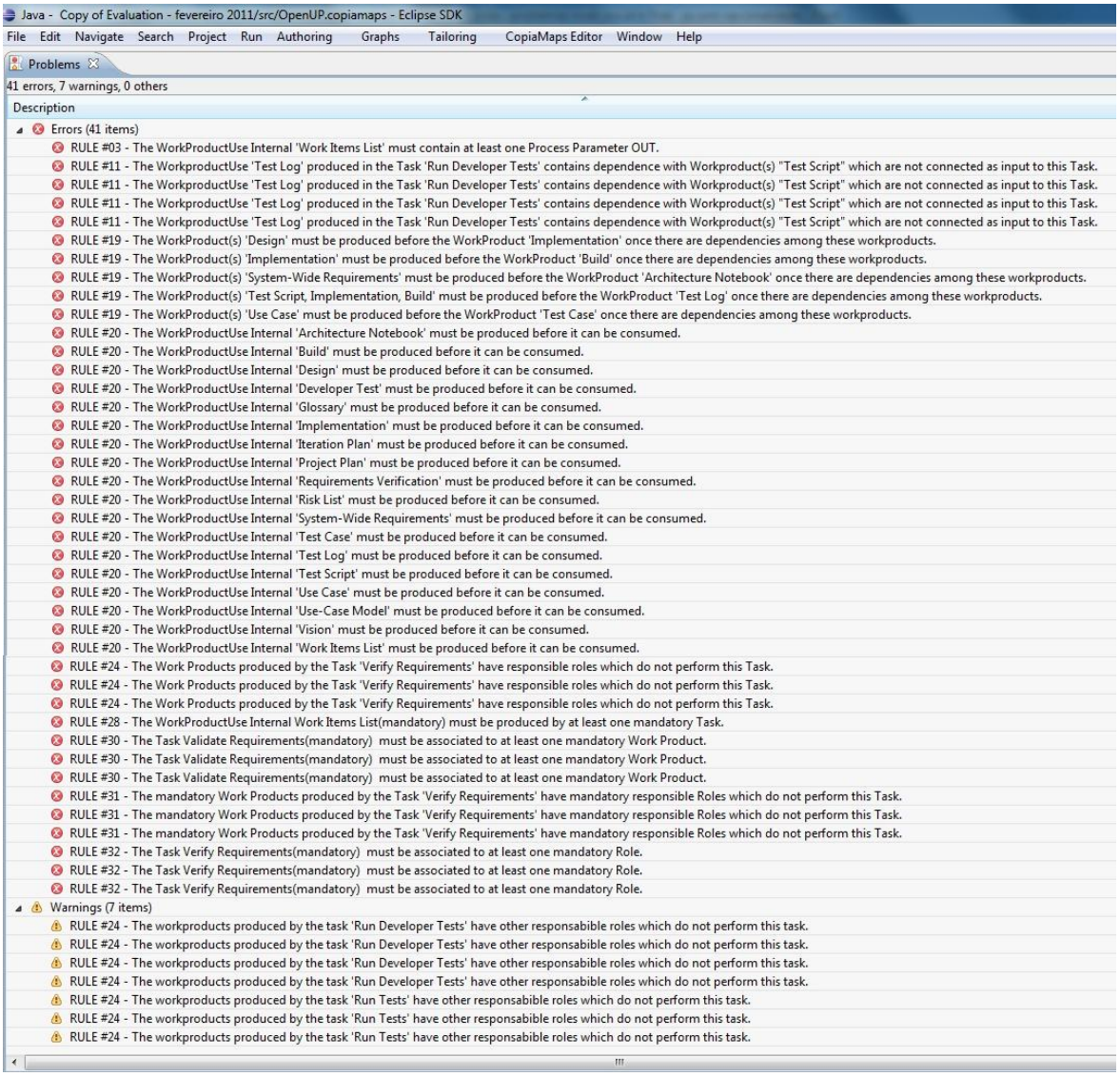

Figura 69 - Validação do processo *OpenUP* com dependências entre produtos de trabalho

Em análise aos novos erros gerados, verificou-se, pelos resultados retornados, que as inconsistências relacionadas com informações sobre dependência entre produtos de trabalho foram as inconsistências que violaram as regras de boa-formação *Rule #11* e *Rule #19.* Estas regras estabelecem, respectivamente, que as todas as dependências de um produto de trabalho devem estar conectadas como entrada nas suas tarefas de produção e que todas estas dependências devem ser produzidas antes do produto de trabalho em questão. Além disso, a *Rule #19* verifica ainda se existe caminho entre as tarefas que produziram a dependência e a(s) tarefa(s) que a consome(m).

Para a *Rule #11,* foram mostrados 4 erros para o mesmo produto de trabalho e para a mesma tarefa. Isso ocorreu, pois a tarefa em questão (*Run Developer Tests*) é

executada em 4 pontos diferentes do processo e não possui como entrada o produto de trabalho *Test Script* que é necessário à produção do produto de trabalho *Test Log* (ver dependências na Tabela 18). Contudo, embora o erro esteja replicado, para consertá-lo, basta corrigir essa tarefa no *Method Content*. Neste momento, todas as instâncias dessa tarefa no processo de software, seriam atualizadas. Em verdade, qualquer erro relacionado com tarefas que possuem mais de uma instância no processo de software é replicado no resultado de validação. Como dito acima, não é necessário consertar um erro por vez, mas sim consertar o erro no *Method Content*.

Os erros retornados para a *Rule #19* já eram esperados nesta avaliação. A razão para ocorrência desse tipo de erro no *OpenUP* está relacionado ao mesmo motivo da ocorrência dos erros relacionados com a *Rule #20*. Como já mencionado no Cenário 1, isso ocorre porque (1) o sequenciamento entre tarefas não é completo no *OpenUP*, ou seja, nem todas as tarefas deste processo são seqüenciadas e (2) muitos produtos de trabalho são realmente consumidos antes de serem produzidos.

Outros erros retornados no resultado desta avaliação são todos resultantes dos novos elementos (atividades, tarefas, produtos de trabalho e papéis) incluídos neste cenário. Especificamente, em análise ao erro relacionado com a regra de boa-formação *Rule #24,* é possível verificar que este erro foi retornado pois o papel que é responsável pelo produto de trabalho (*Requirements Verification*) não é o mesmo que desempenha suas tarefas de produção no processo. Cabe lembrar que, para este cenário, foi incluído o papel *Analyst* como responsável pelo produto de trabalho *Requirements Verification* e definido o papel *Reviewer* como desempenhador das tarefas que produzem este produto de trabalho (*Verify Requirements*). Como esse fato viola a regra de boa-formação *Rule #24,* o erro foi retornado 3 vezes no *framework* de validação, indicando que a tarefa é executada em 3 pontos diferentes do processo.

Por fim, os erros relacionados com as regras de boa-formação *Rule #30*, *Rule #31* e *Rule #32* são todos relacionados com aspectos de opcionalidade dos elementos no processo de software. Esses erros ocorreram, pois o papel *Reviewer* e o produto de trabalho *Requirements Acceptance* foram definidos como opcionais e todos os outros elementos, os quais são relacionados com ambos, papel e produto de trabalho, são obrigatórios.

O erro relacionado com a regra de boa-formação *Rule #30* ocorreu pois a tarefa obrigatória *Validade Requirements* produz apenas o produto de trabalho opcional

Requirements Acceptance. Como a Premissa P #18 estabelece que toda tarefa obrigatória deverá estar a associada a, pelo menos, um produto de trabalho obrigatório, bem como ser executada por, pelo menos, um papel obrigatório (através das regras de boa-formação *Rule #30, #31 e #32*) e a Premissa P #7 estabelece que toda tarefa de um projeto de software deve ser desempenhada por um papel e produzir um resultado de valor observável em termos de produtos de trabalho (através das regras de boa-formação *Rule #4, #9, #10 e #21*), esse cenário desrespeita diretamente a Premissa P #18 e indiretamente a Premissa P #7. Diz-se que a Premissa P #7 é desrespeitada indiretamente porque o produto de trabalho *Requirements Acceptance* poderia ser excluído do processo e, nesse caso, violaria a regra de boa-formação *Rule #9*, desrespeitando, assim, diretamente a premissa P #7.

As últimas regras de boa-formação violadas neste Cenário foram as regras *Rule #31* e *Rule #32*. Relacionado à regra *Rule #31,* isso ocorreu, pois o papel obrigatório *Analyst* que é responsável pelo produto de trabalho obrigatório *Requirements Verification* não está associado à tarefa obrigatória de produção desse produto de trabalho (*Verify Requirements*). Embora essa regra seja similar a regra *Rule #24* explicada acima, ela trata especificamente sobre a opcionalidade dos elementos. Por fim, a regra de boa-formação *Rule #32* foi violada, pois o papel opcional *Reviewer* foi definido como único executor da tarefa obrigatória *Verify Requirements.* Os fatos acima desrespeitam diretamente a Premissa P #18 e Premissa P #6 e indiretamente a Premissa P #7. Vale destacar que os erros estão duplicados no resultado da avaliação, uma vez que as tarefas em questão são desempenhadas em vários pontos do processo *OpenUP*.

A Tabela 19 mostra os resultados da avaliação do Cenário 2 em termos de quais regras de boa-formação e premissas puderam ou não ser atendidas atendidas.

Tabela 19 - Premissas e regras analisadas no Cenário 2

Premissas e Regras Atendidas		**Premissas e Regras Não Atendidas**	
Premissa	**Regra de Boa-Formação**	**Premissa**	**Regra de Boa-Formação**
P #4	*Rule #5, #7 e #49*	**P #1** **P #2** **P #3**	*Rule #3*
P #6	*Rule #8*	**P #9**	*Rule #19 e #20*
P #7	*Rule #4, #9, #10 e #21*	**P #15** **P #16** **P #17** **P #18** **P #19** **P #20**	*Rule #28, #30, #31 e #32*
P #10 **P #11** **P #12**	*Rule #6, #12, #13, #14, #15, #16, #17, #18, #41, #42, #43, #44, #45 e #50*	**P #6**	*Rule #24*
P #13	*Rule #22*	**P #8**	*Rule #11* e *#19*

P #14	*Rule #48*	-	-
P #21 **P #22**	*Rule #33, #34, #35, #36, #37, #38, #39 e #40*	-	-
P #5	*Rule #5*	-	-
P #15 **P #16** **P #17** **P #18** **P #19** **P #20**	*Rule #23, #25, #26, #27 e #29*	-	-

7.2.4 Cenário 3

Neste cenário dois aspectos distintos são avaliados: (1) o uso de produtos de trabalho do tipo externo no processo e (2) as informações sobre sequenciamento entre unidades de trabalho. Assim, as seguintes premissas e regras de boa-formação são analisadas: **P #1, P #2** e **P #3** (*Rule #1, #2,* e *#3*); **P #4** (*Rule #5*, *#7, #49*); **P #5** (*Rule #5*); **P #6** (*Rule #8* e *#24*); **P #7** (*Rule #4, #9, #10* e *#21*); **P #8** (*Rule #11 e #19*); **P #9** (*Rule #19* e *#20*); **P #10, P #11 e P #12** (*Rule #6, #12, #13, #14, #15, #16, #17, #18, #41, #42, #43*, *#44, #45 e #50*); **P #13** (*Rule #22*); **P #14** (*Rule #48*); **P #15, P #16, P #17, P #18, P #19** e **P #20** (*Rule #23, #25, #26, #27, #28, #29, #30, #31 e #32*); **P #21** e **P #22** (*Rule #33, #34, #35, #36, #37, #38, #39* e *#40*).

Para facilitar a compreensão, este cenário é executado apenas sobre a iteração *Inception Iteration* do *OpenUP*. Contudo, para o Cenário 3 algumas alterações foram realizadas nessa iteração, uma vez que para esse cenário está sendo considerado a definição de um processo para um projeto de manutenção de software evolutiva (tipo de processo que altera, inclui e/ou exclui requisitos de um sistema [Iee98]). Isso foi realizado, uma vez que a inclusão de produtos de trabalho do tipo externo se aplica muito bem a este tipo de processo.

A Figura 70 exibe o detalhamento das tarefas em termos de entradas e saídas de produtos de trabalho e papéis que são avaliadas no Cenário 3. Para ver as atividades e informações de sequenciamento da iteração *Inception Iteration,* as quais não foram modificadas, pode-se consultar a Figura 65.

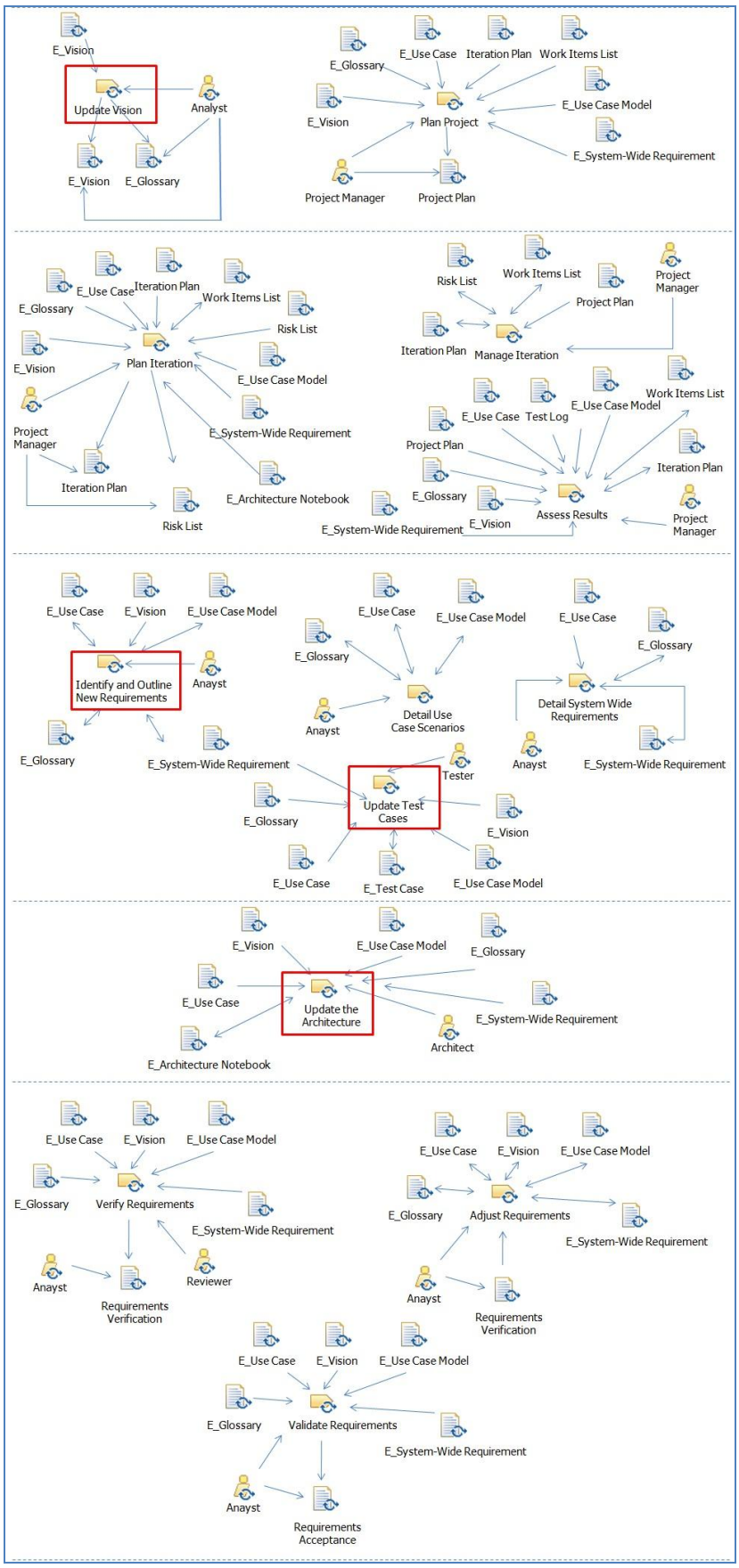

Figura 70 - Detalhamento das tarefas do Cenário 3

Observa-se que muitos produtos de trabalho foram definidos como produtos de trabalho externos (identificados na Figura 70 com a letra *E* no início do nome). Por se tratar de um processo para um projeto de manutenção de software, foi considerado para o Cenário 3 que todos os produtos de trabalho de *Requisitos*, *Análise*, *Implementação* e *Testes* já estavam prontos e sofrerão apenas modificações no novo projeto de software. Essa é a razão para que tais produtos de trabalho tenham sido definidos como externos (*ExternalUse*). Outro ponto importante a considerar é que algumas tarefas (destacadas por um retângulo na Figura 70) foram alteradas na iteração *Inception Iteration* para contemplar apenas a modificação dos produtos de trabalho externos.

Quanto à opcionalidade dos elementos do processo, considerou-se que a modificação de todos os produtos de trabalho de *Requisitos*, *Análise*, *Implementação* e *Testes* é opcional no processo criado. Em projetos de software, por exemplo, que não possuem nenhuma alteração, inclusão e exclusão de termos de negócio, não será necessário a atualização do produto de trabalho *Glossary*. Já em projetos que apenas corrijem erros no código não serão necessárias alterações em nenhum produto de trabalho de *Requisitos* e *Análise*. Como consequência da definição de produtos de trabalho opcionais, para não causar inconsistências em tal processo, todas as tarefas de modificação e papéis associados a essas tarefas também foram definidos como opcionais. Essa definição foi realizada, pois, embora se saiba que em um processo de manutenção de software evolutiva exista alteração, inclusão e/ou exclusão de requisitos, por não conhecer tais requisitos, não é possível identificar durante a definição de um processo de software quais produtos de trabalho sofrerão alteração.

Ainda relacionado ao aspecto de opcionalidade, o produto de trabalho *Requirements Verification* e as tarefas de verificação e ajuste dos requisitos *Verify Requirements* e *Ajust Requirements* também foram redefinidas como opcionais. Todas as outras tarefas e os outros produtos de trabalho relacionados com planejamento de projeto e validação dos requisitos por parte do cliente continuaram sendo considerados obrigatórios (conforme havia sido definido no Cenário 1).

Após todas as definições acima e criação das novas tarefas no *Method Content,* o novo processo foi definido na área de processos em *sSPEM Tool*. Neste momento, todos os produtos de trabalho, tarefas e papéis foram utilizados a partir *Method Content*. Ainda, todos os produtos de trabalho de *Requisitos*, *Análise*, *Implementação* e *Testes* foram definidos como externos. A Figura 71 exibe a iteração *Inception Iteration* incluída em *sSPEM Tool*.

Como já exposto, o processo criado para este cenário é composto apenas por uma iteração. Até este momento, nenhuma informação sobre sequenciamento foi incluída. A ideia foi de, antes de incluir essas informações, executar o *framework* de validação e verificar se inconsistências relacionadas com aspectos de sequenciamento são identificados. O resultado dessa validação é exibido na Figura 72.

Avaliando os resultados retornados pelo *framework* de validação, verifica-se que várias inconsistências já explicadas nos cenários anteriores aparecem novamente. É o caso das inconsistências relacionadas com as regras de boa-formação *Rule #3, #20, #24, #28, #30* e *#32*. A única diferença encontrada no resultado deste cenário relacionado com estas regras é a sua quantidade.

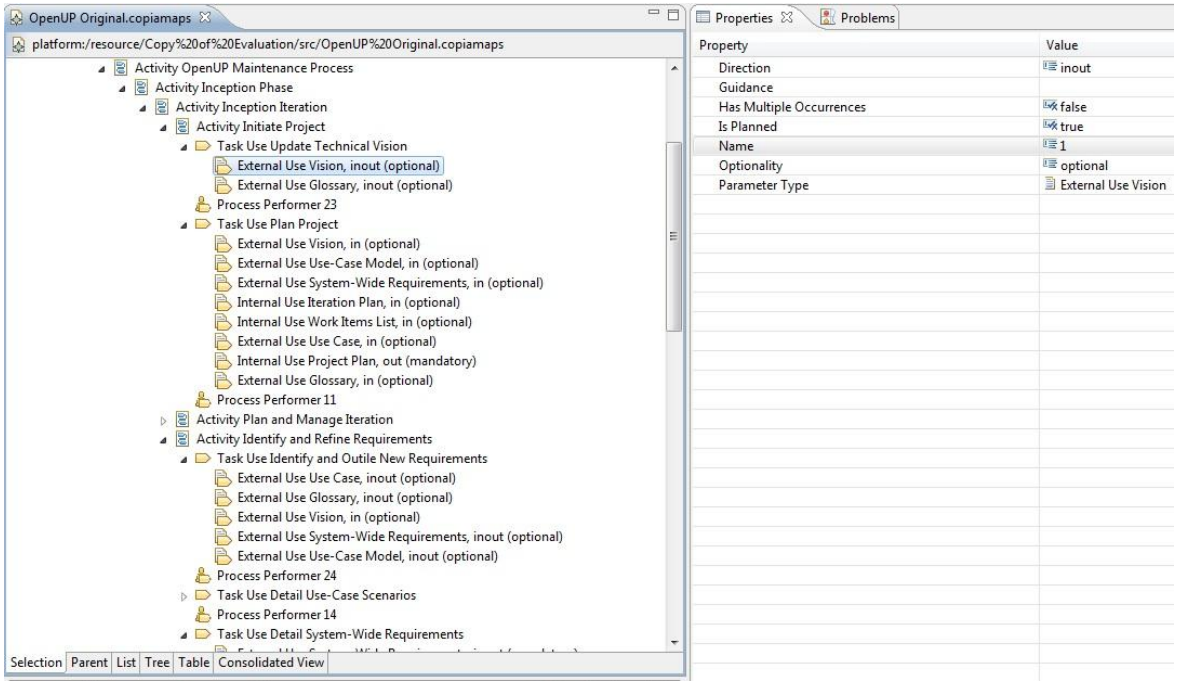

Figura 71 - Inclusão do Cenário 3 em *sSPEM Tool*

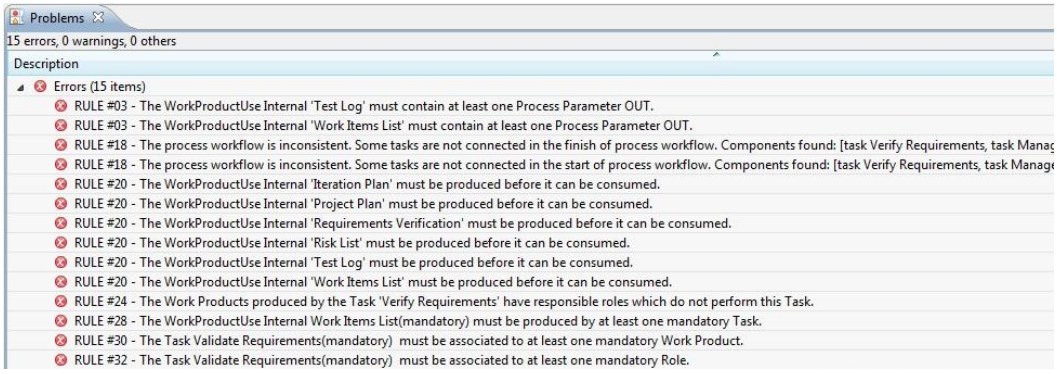

Figura 72 - Primeira validação do Cenário 3 (sem informações novas de sequenciamento)

No caso das inconsistências retornadas para as regras de boa-formação *Rule #20, #24, #30* e *#32,* verifica-se que o número de inconsistências diminuiu. Isso ocorreu

porque este cenário não possui execução das mesmas tarefas em pontos diferentes do processo e porque a quantidade de produtos de trabalho utilizados é menor do que a quantidade dos outros cenários. Já a regra de boa-formação *Rule #3* aparece também para o produto de trabalho *Test Log*. Isso ocorre, pois esse produto de trabalho não é produzido em nenhum ponto do processo.

A nova regra de boa-formação resultante no *framework* de validação é a regra *Rule #18*. Ela aparece duas vezes no *framework* de validação e indica que existem inconsistências no fluxo do processo, uma vez que existem tarefas no processo desconectadas do nodo inicial e/ou do nodo final. No caso específico desta avaliação, uma vez que nenhum sequenciamento foi ainda definido, a regra retorna duas vezes indicando que todas as tarefas estão desconectadas do início e fim do processo.

Ainda que tenham sido criados vários produtos de trabalho do tipo externo para este cenário, verifica-se que nenhuma inconsistência foi encontrada relacionada com esse aspecto. Isso ocorre pois todos os produtos de trabalho foram conectados como entrada e/ou conectados como saída (para modificação) em alguma tarefa e também porque nenhuma tarefa foi conectada como produtora para nenhum desses produtos de trabalho, o que indica que eles foram definidos de forma consistente.

Para finalizar a avaliação do Cenário 3 outras regras de boa-formação para consistência relacionadas com sequenciamento foram testadas. Para fazer isso, inicialmente, um fluxo foi definido para as tarefas que fazem parte da iteração *Inception Iteration.* Optou-se por definir o fluxo entre as tarefas para que inconsistências relacionadas com os produtos de trabalho comecem a ser consertadas. O fluxo definido para a iteração *Inception Iteration* é mostrado no na Figura 73. Essa figura mostra um módulo em *sSPEM Tool* desenvolvido para verificar informações sobre sequenciamento graficamente. Deve-se notar que a Figura 73 apresenta, pela primeira vez, o sequenciamento em um grafo, ou seja, nessa figura, é apresentada a solução de sequenciamento proposta nesta tese (conforme Capítulo 4).

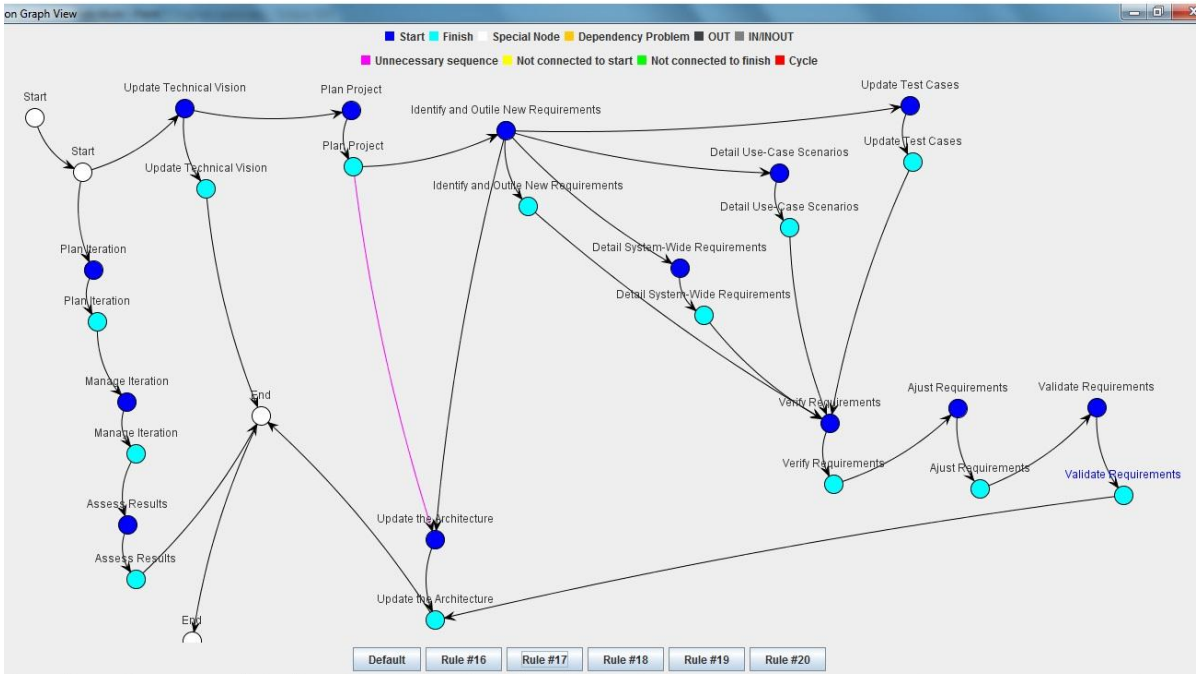

Figura 73 - Interface em *sSPEM Tool* mostrando o detalhamento do fluxo definido para a Iteração *Inception Iteration*

Também na Figura 73, há os nodos, em azul mais escuro, que representam o início das tarefas; e os nodos, em azul mais claro, que representam o fim das tarefas. O sequenciamento definido para as tarefas respeita a definição original de sequenciamento do *OpenUP* para a iteração em questão (ver Figura 66). A única diferença foi a definição de sequenciamento interno para as tarefas das atividades e a criação das tarefas que representam o nodo inicial e final do processo. Outra alteração realizada foi a inclusão de mais uma pré-condição para a tarefa *Update the Architecture* poder ser executada. Agora, além de finalizar a tarefa *Plan Project,* é necessário que a tarefa *Identify and Outline new Requirements* comece para a execução dessa tarefa. O final da tarefa *Update the Architecture* também fica condicionada ao término de todas as tarefas de requisitos.

Como visto na imagem exibida na Figura 73, todas as novas informações de sequenciamento foram incluídas em *sSPEM Tool*. Depois disso, o *framework* de validação foi novamente executado, retornando o resultado apresentado na Figura 74.

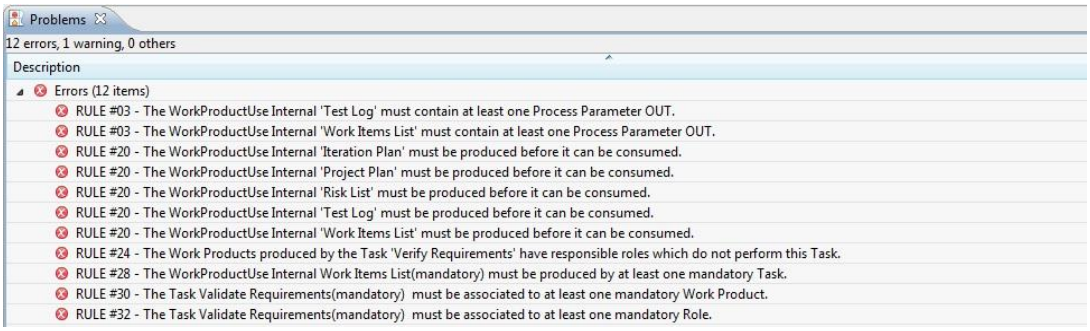

Figura 74 - Validação do Cenário 3 após inclusão das novas informações de sequenciamento

Os resultados acima mostram que os erros referentes à regra de boa-formação *Rule #18* foram consertados. Além disso, um dos erros relacionados com a *Rule #20* foi reparado durante a inclusão das informações de sequenciamento. Um novo *warning* também aparece no resultado da avaliação. Ele indica um caminho desnecessário entre as tarefas *Plan Project* e *Update the Architecture*. Isso acontece porque, no sequenciamento, é estabelecido que a tarefa *Update the Architecture* só pode começar após o final da tarefa *Plan Project* e o início da tarefa *Identify and Outline new Requirements.* Uma vez que a tarefa *Identify and Outline new Requirements* só pode começar após o final da tarefa *Plan Project,* o sequenciamento estabelecido entre a tarefa *Plan Project* e *Update the Architecture* não tem sentido nenhum.

Na Figura 73, o *warning* é destacado através da cor rosa indicada na aresta que é desnecessária. Em verdade, qualquer erro relacionado por sequenciamento pode ser acessado pelos botões disponíveis na interface de *sSPEM Tool* (conforme visto na Figura 73).

Como o objetivo do Cenário 3 também é avaliar as regras de boa-formação relacionadas com sequenciamento, neste momento, algumas situações-problema foram incluídas no processo sendo avaliado. As situações-problema referem-se à criação de um novo nodo inicial (duplicando o início do processo), exclusão do nodo final do processo (processo não possui mais fim) e inclusão de alguns ciclos entre tarefas. A Figura 75 exibe como fica o novo Cenário.

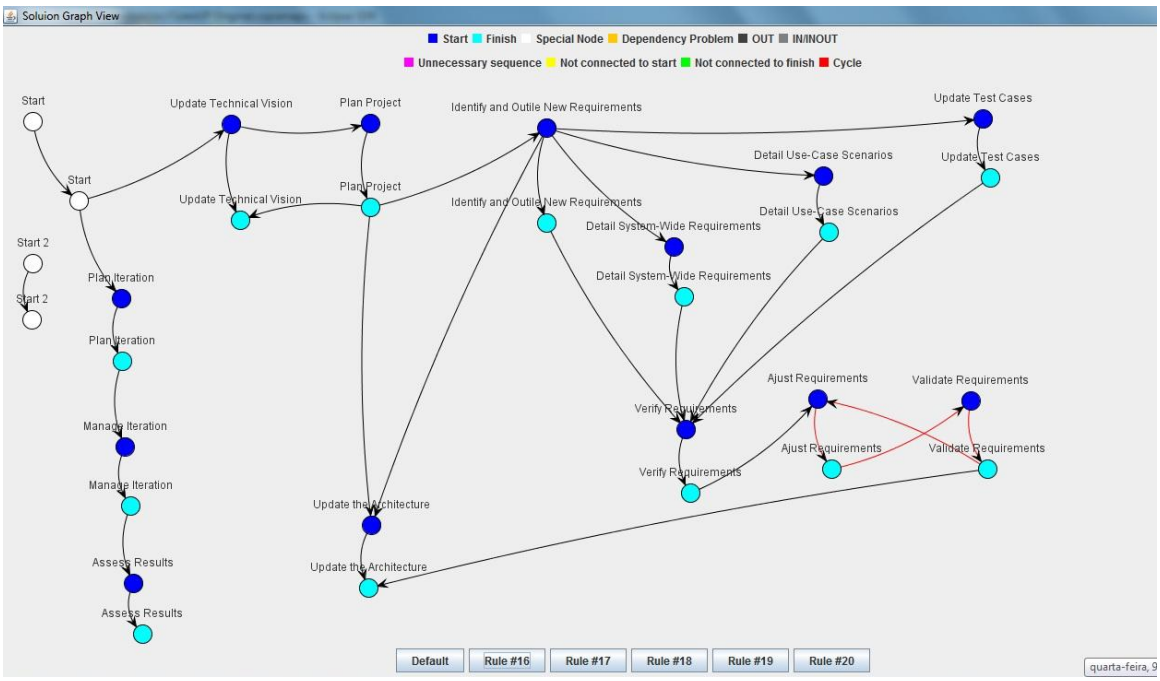

Figura 75 - Interface em *sSPEM Tool* mostrando as situações-problema de sequenciamento incluídas no Cenário 3 – destaque para a validação de ciclos

Na figura, é possível ver que existem dois nodos iniciais no processo e nenhum nodo final. Também na Figura 75, já foi acionado o botão "*Rule #16*" que testa se existem ciclos no grafo, e, como consequência, algumas arestas aparecem destacadas em vermelho mostrando exatamente onde houve formação de ciclo. Nota-se que esse ciclo foi incluído neste momento da avaliação, pois esse caminho não existia no grafo anterior (ver Figura 73). Deve-se notar também que parece haver a existência de outro ciclo entre as tarefas *Update Technical Vision* e *Plan Project*.

Contudo, observando-se em detalhe a Figura 75, verifica-se que não há formação de ciclo nessa situação. Embora se tenha incluído uma sequencia que parte da tarefa *Update Technical Vision* para a tarefa *Plan Project* e outra sequencia ao contrario, ou seja, que parte da tarefa *Plan Project* para a tarefa *Update Technical Vision,* não houve a geração de ciclo. O que ocorre é que, com esse novo sequenciamento, a execução da tarefa *Plan Project* fica dentro do tempo da execução da tarefa *Update Technical Vision,* isto é, a tarefa *Plan Project* só inicia depois do início da tarefa *Update Technical Vision* e esta tarefa, por sua vez, só pode terminar após o fim da tarefa *Plan Project.*

Tem de se perceber ainda na Figura 75 que situações-problema relacionadas com os nodos iniciais e finais ainda não apareceram. Isso ocorre, pois, na implementação de *sSPEM Tool* foi definido que, quando existem problemas de ciclo em um processo (*Rule #16*), as regras *Rule #17* (relacionada com transições duplicadas e/ou desnecessárias), *Rule #18* (relacionada com tarefas x nodo final e/ou inicial), *Rule #19* (relacionado com produção de dependências de produtos de trabalho) e *Rule #20* (relacionado com consumo de produtos de trabalho antes de sua produção) não são executadas. A Figura 76 mostra o resultado do *framework* de validação neste ponto da avaliação.

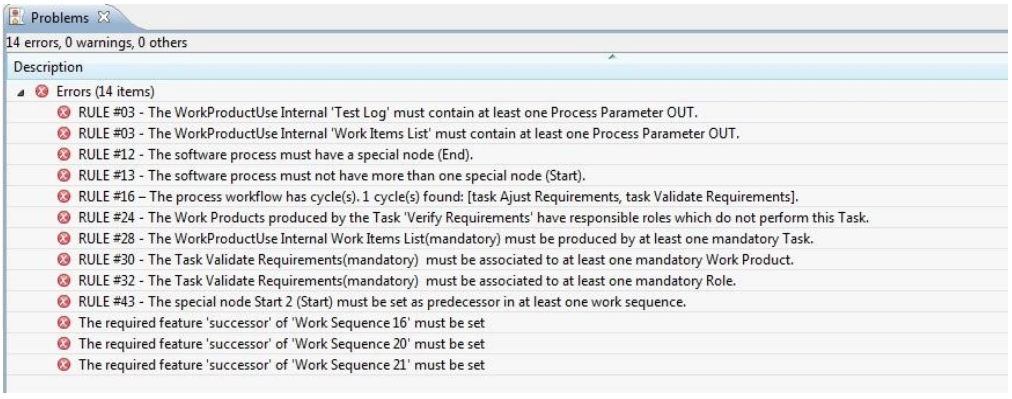

Figura 76 - Validação do Cenário 3 após a inclusão das situações-problema de sequenciamento

Note que a regra *Rule #20,* que aparecia anteriormente (na última validação - Figura 74), não é mostrada. Contudo, embora isso tenha ocorrido, o número de erros desde a última validação aumentou, pois existem, agora, novos erros. Tais novos erros *Rule #12* (obrigatoriedade de um único nodo final no processo), *Rule #13* (obrigatoriedade de um único nodo inicial no processo), *Rule #16* (processo não deve possuir ciclos) e *Rule #43* (todo nodo inicial deve ser predecessor para, pelo menos, uma tarefa ou atividade) já eram esperados e estão relacionados com as novas situações-problema incluídas no processo. Os outros 3 erros que aparecem na Figura 76 e não possuem identificação foram indicados pela própria validação do EMF. Eles indicam alguma violação em multiplicidades do modelo. Especificamente neste caso, os erros apareceram porque o nodo final foi apagado do processo deixando algumas *Work Sequence* sem sucessor (atributo obrigatório para a metaclasse *WorkSequence* no metamodelo sSPEM 2.0).

Prosseguindo a avaliação, uma vez que a regra de ciclo já foi demonstrada, a transição (que formava ciclo) *finishToStart* entre as tarefas *Validade Requirements* e *Ajust Requirements* foi apagada, conforme mostrado na Figura 77.

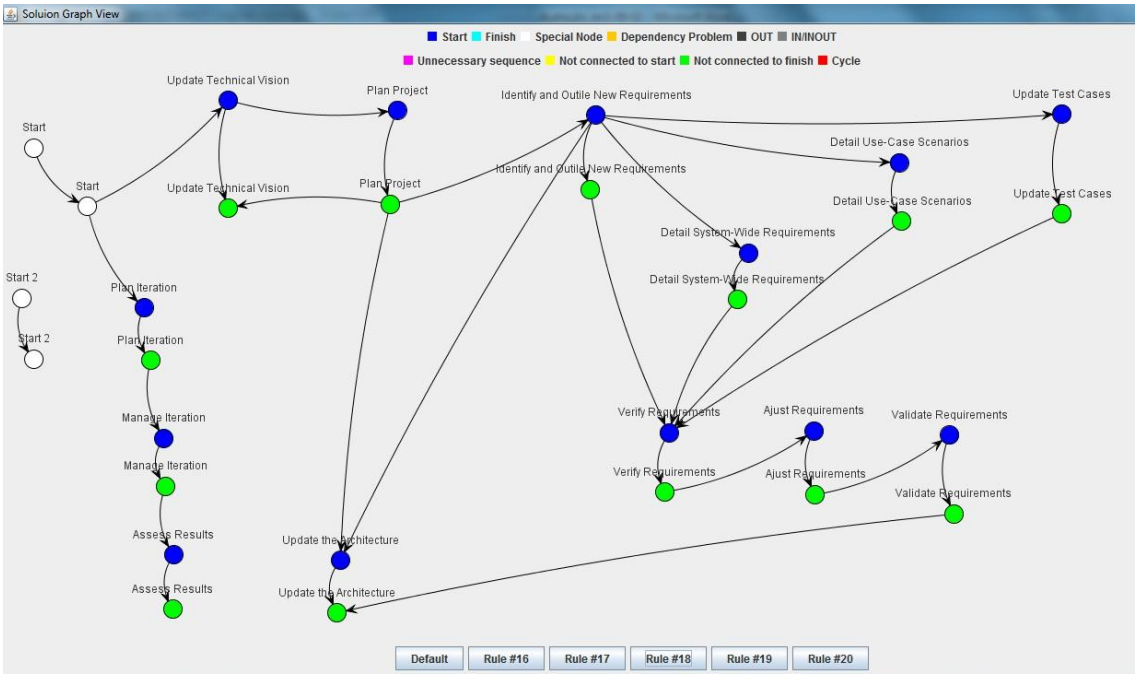

Figura 77 - Interface em *sSPEM Tool* mostrando as situações-problema de sequenciamento incluídas no Cenário 3 – destaque para a validação de nodos de início de fim do processo

A Figura 77 mostra a validação de uma nova regra (*Rule #18*) relacionada com sequenciamento. Nessa figura, os nodos em verde são os nodos que apresentam problemas (conforme legenda da Figura 77), pois não possuem conexão com o final do

processo. Ainda que exista problema de duplicação do nodo inicial do processo, a regra *Rule #18* não aparece para esse tipo de nodo, pois todas as tarefas do processo estão devidamente conectadas com, ao menos, um início no processo, fato que atende a regra *Rule #18*. Para complementar os erros que foram apresentados de forma visual, a Figura 78 mostra o resultado do *framework* de validação para o novo Cenário (neste momento, sem ciclos).

O resultado da validação mostrado na Figura 78 exibe que o processo não tem mais problemas de ciclo, uma vez que não existe mais retorno para a regra *Rule #16*. A Figura 78 também mostra que os problemas relacionados com a regra *Rule #20* voltaram a ser exibidos. Isso ocorre, pois como dito anteriormente, quando não existem problemas de ciclo as regras *Rule #17, Rule # 18, Rule 19* e *Rule #20* são normalmente avaliadas em *sSPEM Tool*. Outra diferença encontrada referente à última validação foi o novo erro (*Rule #18*) exibido como resultado.

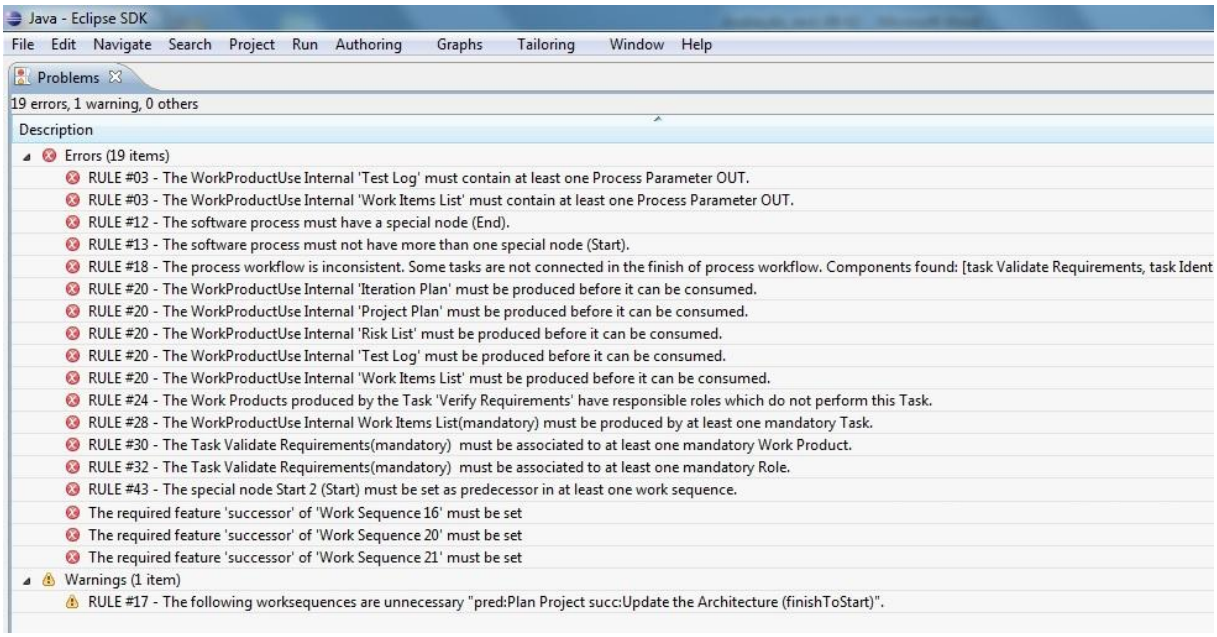

Figura 78 - Validação do Cenário 3 após a inclusão das situações-problema de sequenciamento - sem formação de ciclos

Avaliadas as inconsistências relacionadas com sequenciamento de tarefas e tendo sido demonstrado a utilização de produtos de trabalho do tipo externo, considerou-se, neste momento, encerrada a avaliação do Cenário 3. A Tabela 20 apresenta o resultado da avaliação deste cenário.

Tabela 20 - Premissas e regras analisadas no Cenário 3

Premissas e Regras Atendidas		**Premissas e Regras Não Atendidas**	
Premissa	**Regra de Boa-Formação**	**Premissa**	**Regra de Boa-Formação**
P #1 **P #2** **P #3**	*Rule #1 e #2*	**P #1** **P #2** **P #3**	*Rule #3*

P #4	*Rule #5, #7 e #49*	**P #9**	*Rule #20*
P #6	*Rule #8*	**P #6**	*Rule #24*
P #7	*Rule #4, #9, #10 e #21*	**P #10** **P #11** **P #12**	*Rule #12, #13, #16, #18 e #43*
P #8	*Rule #11 e #19*	**P #15** **P #16** **P #17** **P #18** **P #19** **P #20**	*Rule #28, #30 e #32*
P #9	*Rule #19*	-	-
P #10 **P #11** **P #12**	*Rule #6, #14, #15, #17, #41, #42, #44, #45 e #50*	-	-
P #13	*Rule #22*	-	-
P #14	*Rule #48*	-	-
P #21 **P #22**	*Rule #33, #34, #35, #36, #37, #38, #39 e #40*	-	-
P #5	*Rule #5*	-	-
P #15 **P #16** **P #17** **P #18** **P #19** **P #20**	*Rule #23, #25, #26, #27, #29* e *#31*	-	-

7.2.5 Cenário 4

Neste cenário as regras de boa-formação específicas para adaptação de processo são analisadas, sendo utilizado para tanto os mecanimos *usedActivity* e *supressedBreakdownElement*. Aqui, o ambiente utilizado foi o mesmo do Cenário 3, ou seja, as atividades e tarefas da iteração *Inception Iteration*. É importante citar, entretanto, que o Cenário considerado neste ponto da avaliação é o mesmo apresentado na Figura 73 e os erros considerados são os exibidos na Figura 74. Em outras palavras, para esse cenário utiliza-se o Cenário 3 antes da inclusão dos problemas de sequenciamento (nodo inicial duplicado, exclusão do nodo final e inclusão de ciclos), bem como avalia-se as mesmas premissas e regras de boa-formação.

Uma vez que o processo do Cenário 3 contém várias inconsistências (conforme Figura 74) e que um novo processo será gerado através dos mecanismos de adaptação, o primeiro passo desta avaliação foi a modificação de algumas informações do processo utilizado nesse cenário com objetivo de consertar as suas inconsistências.

As primeiras informações modificadas corrigem os erros da regra *Rule #20*. Neste momento, essa regra aparece *5 vezes* no resultado da avaliação (conforme Figura 74) pois 5 produtos de trabalho são consumidos antes de serem produzidos no processo do Cenário 4. Analisando esses produtos de trabalho, verifica-se que dois deles, *Iteration*

Plan e *List Risk,* são consumidos e produzidos pela mesma tarefa. Como o consumo destes produtos de trabalho é opcional, optou-se por excluí-los das suas tarefas produtoras. Após essas exclusões, o resultado do *framework* de validação retornou 11 erros. Em verdade, o problema relacionado com o produto de trabalho *List Risk* foi solucionado pois este produto de trabalho não é mais consumido no processo do Cenário 4. Contudo, os erros relacionados com a *Rule #20* ainda se apresentam para os seguintes produtos de trabalho: *Iteration Plan, Project Plan, Work Items List* e *Test Log.*

A solução para consertar os erros citados acima é a criação de caminhos entre as tarefas que produzem os produtos de trabalho em questão e todas as tarefas que consomem eles. Contudo, nota-se, no *framework* de validação, através do erro *Rule #3*, que os produtos de trabalho *Work Items List* e *Test Log* não possuem tarefa de produção e, dessa forma, torna-se necessário a criação de novos elementos para o processo do Cenário 4.

Antes de continuar tratando os erros *Rule #20,* optou-se por analisar os produtos de trabalho *Work Items List* e *Test Log*. Inicialmente, verificou-se que o produto de trabalho *Test Log* é entrada opcional para apenas uma tarefa (*Assess Results*) no Cenário 4. Dessa forma, decidiu-se excluir os seguintes elementos do processo: o produto de trabalho *Test Log*, o relacionamento que definia os papéis *Tester* e *Developer* como responsáveis por este produto de trabalho e também o parâmetro de entrada que continha o produto de trabalho *Test Log* da tarefa *Assess Results*. Neste momento, executou-se o *framework* de validação para verificar se as ações de exclusão não haviam gerado novos erros. O resultado da nova validação é mostrado na Figura 79.

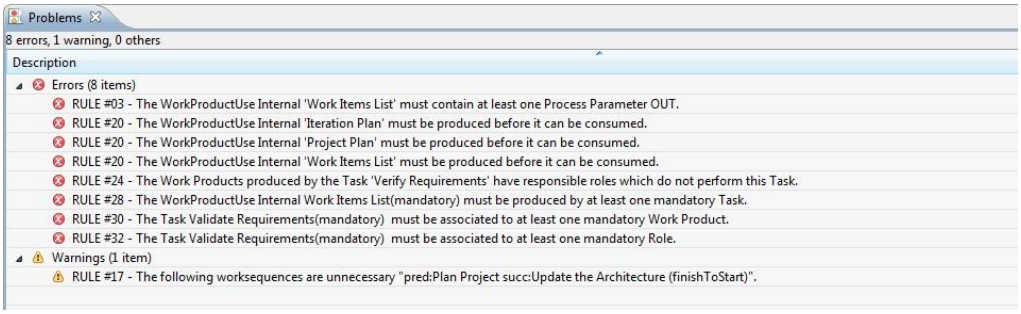

Figura 79 - Primeira validação do Cenário 4 – após a criação de caminhos entre tarefas e exclusão do produto de trabalho *Test Log*

Observa-se, na figura acima, que a quantidade de erros diminuiu, não existindo mais referências para o produto de trabalho *Test Log*. Especificamente os erros *Rule #3 e #20* deixaram de existir no processo.

Para consertar os problemas relacionados com o produto de trabalho *Work Items List (Rule #3, #20* e *#28)*, os quais são originários (conforme Cenário 1) do processo *OpenUP*, uma análise foi realizada sobre a descrição desse produto de trabalho e sobre as tarefas do processo. Verificou-se, assim, que o objetivo do *Work Items List* é listar todo o trabalho, geralmente expresso em termos de produtos de trabalho, que deve ser realizado em um projeto de software, podendo isso ser produzido como parte do produto de trabalho *Iteration Plan* (informações extraídas da seção *Representation Options* do produto de trabalho *Work Items List* no *OpenUP*).

Dessa maneira, com base na informação exposta acima, optou-se, nesta avaliação por produzir o produto de trabalho *Work Items List* na tarefa *Plan Iteration*. Originalmente, tal produto de trabalho era apenas modificado por essa tarefa. Após as alterações do processo do Cenário 4, a execução do *framework* de validação mostrou os seguintes resultados (Figura 80).

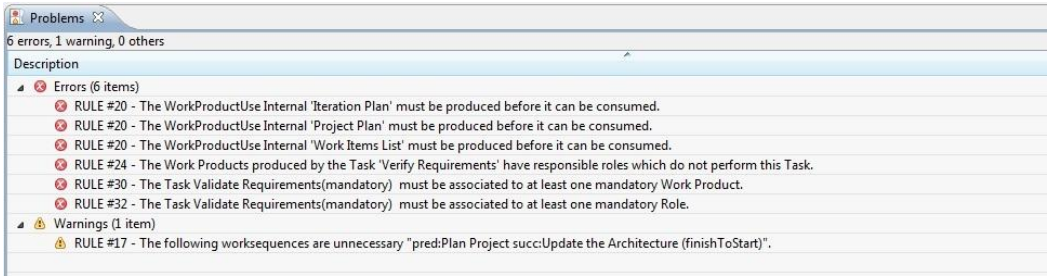

Figura 80 - Validação do Cenário 4 após consertar erros do produto de trabalho *Work Items List*

Nota-se que os erros *Rule #3 e #28* não apareceram no resultado da validação. Isso ocorreu, pois, agora, o produto de trabalho *Work Items List* é produzido por uma tarefa obrigatória, fato que atende ambas as regras de boa-formação em questão.

Seguindo a avaliação, os erros da regra *Rule #20* foram consertados. Como dito anteriormente, para fazer isto, é necessário criar caminho entre as tarefas que produzem e as tarefas que consomem os produtos de trabalho em questão. Neste momento, apenas um sequenciamento do tipo *startToStart* entre as tarefas *Plan Project* e *Plan Iteration* foi incluído. Isso foi realizado porque, analisando as informações de sequenciamento estabelecidas no Cenário 3 e também a descrição das tarefas do *OpenUP,* verificou-se que é necessário que a tarefa de planejamento de projeto tenha começado para que a tarefa de planejamento de iteração possa ser iniciada.

Além da inclusão do sequenciamento mencionado, os parâmetros de entrada das tarefas do processo foram analisados. Isto foi feito, pois se constatou que algumas tarefas

estavam erradas pois consumiam produtos de trabalho produzidos por uma tarefa posterior. Especificamente neste cenário, observou-se que os produtos de trabalho *Iteration Plan* e o *Work Items Li*st produzidos na tarefa *Plan Iteration* eram consumidos pela tarefa *Plan Project* (tarefa que é anterior a tarefa *Plan Iteration*). Como esses parâmetros de entrada eram opcionais, neste momento, decidiu-se por excluir essas informações da tarefa *Plan Project*.

A Figura 81 mostra o resultado do *framework* de validação neste ponto da avaliação. Observa-se, na Figura acima, que todos os erros relacionados com a regra *Rule #20* não são exibidos. O restante dos erros (*Rule #24, #30* e *#32*) mostrados são provenientes dos novos elementos incluídos no Cenário 2 desta avaliação. Como já detalhado na execução do Cenário 2, alguns erros foram incluídos propositalmente, como por exemplo, a inclusão de elementos opcionais que conflitam com elementos obrigatórios.

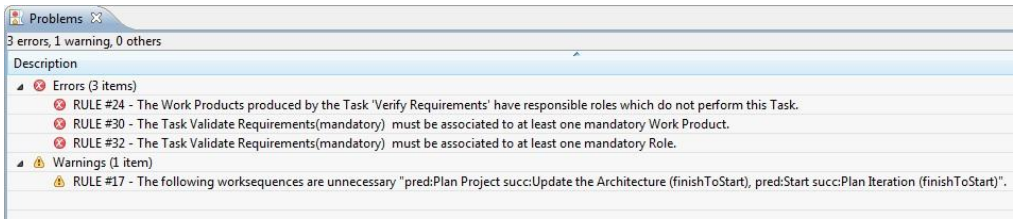

Figura 81 - Validação do Cenário 4 – após a criação de caminhos entre tarefas e exclusão do parâmetros de entrada para a tarefa *Plan Project*

Desse modo, para consertar os últimos erros, o papel *Analyst* e o produto de trabalho *Requirements Acceptance* foram definidos como obrigatórios (atendendo as regras *Rule #30* e *#32*) e o papel *Analyst,* que é responsável pelo produto de trabalho *Requirements Verification,* foi incluído como desempenhador da tarefa *Verify Requirements* (atendendo as regras *Rule #24*).

Por fim, apenas para eliminar o *warning* que aparece no *framework* de validação, duas transições desnecessárias foram excluídas. Uma das transições (conforme mostrado na Figura 73) diz respeito a uma transição incluída no Cenário 3 (entre as tarefas *Plan Project* e *Update the Architecture*). A outra transição excluída foi a transição do tipo *finishToStart* entre o nodo inicial e a tarefa *Plan Iteration*. Uma vez que o início da tarefa *Plan Iteration* foi condicionado neste cenário ao início da tarefa *Plan Project,* essa transição tornou-se desnecessária.

Neste ponto da avaliação do Cenário 4, considera-se que todas as inconsistências foram consertadas no processo. Dessa forma, o *framework* de validação foi novamente executado para confirmar que nenhum erro é encontrado no processo. O resultado, que é visto na Figura 82, retorna 0 erros e 0 *warning,* demonstrando que o

processo encontra-se 100% consistente com as regras de boa-formação propostas nesta tese.

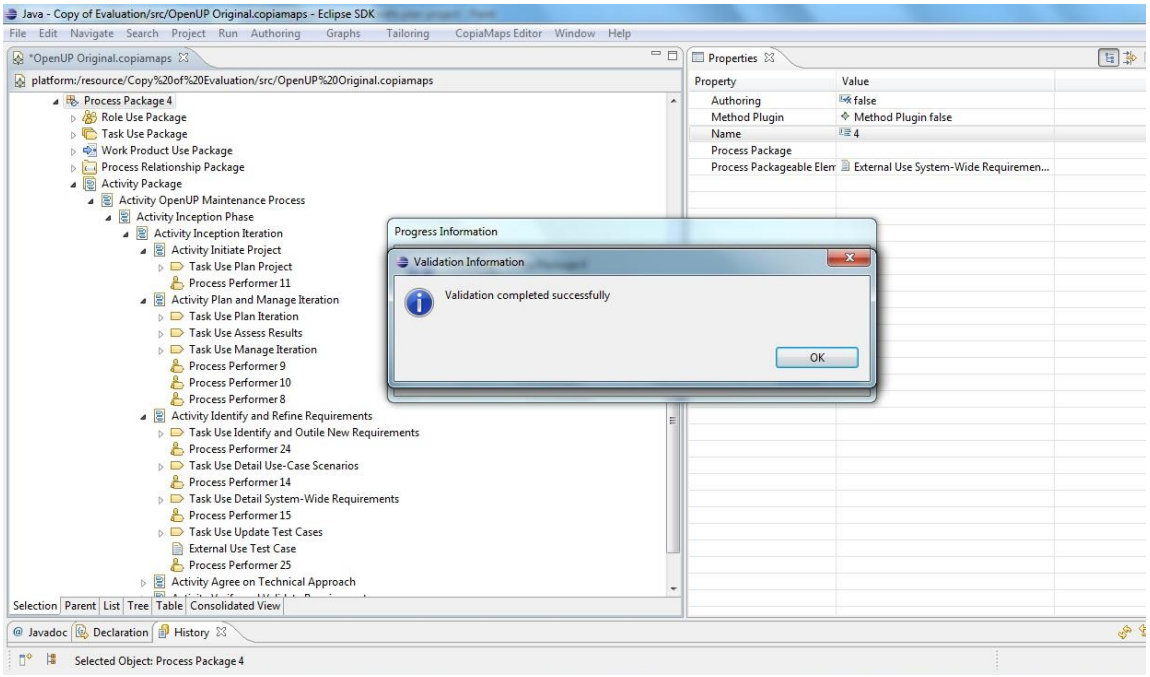

Figura 82 - Validação do Cenário 4 – processo 100% consistente

Após consertar todos os erros provenientes do Cenário 3, os primeiros passos para criação de um processo de software através dos mecanimos de adaptação do metamodelo sSPEM 2.0 foram realizados. Isto envolveu a definição de uma instância da metaclasse *ProcessPackage* (com valor de atributo *isAuthoring* = *false*) como parte da instância de *MethodPlugin*, a criação de uma instância da metaclasse *ActivityPackage* e a criação de uma instância da metaclasse *Activity*. Tais instâncias tornam-se necessárias, pois conforme explicado no guia para definição e adaptação de processos apresentado na no Capítulo 5 deste estudo, todo novo processo deverá ser representado por uma nova instância de *ProcessPackage*. Além disso, o estabelecimento de uma instância de *ActivityPackage* é obrigatória, uma vez que nenhum elemento pode ser criado fora dos pacotes. Já o estabelecimento da instância de *Activity* é necessário para representar o processo de software sendo criado.

Na primeira etapa da avaliação do Cenário 4, optou-se por avaliar o tipo de herança *extension* do mecanimo *usedActivity* e o mecanismo *supressedBreakdownElement*. Para esse cenário, foi considerado que um novo processo necessitava ser montado para um determinado projeto de manutenção de software bastante simples. Assim, a ideia foi herdar todo conteúdo do processo *OpenUp Maintenance Process* (criado no Cenário 3) através do mecanismo *extension* e após isto

excluir alguns elementos desse processo através do mecanismo *supressedBreakdownElement.*

Como uma instância da metaclasse *Activity* já foi criada no cenário em questão para representar o processo sendo adaptado, então, para herdar o conteúdo do processo *OpenUp Maintenance Process,* bastou relacionar a nova instância de *Activity* com a instância de *Activity* que representa o processo *OpenUp Maintenance Process* pelo relacionamento *usedActivity* com valor igual a *extension*. Nesse momento, todo conteúdo do processo *OpenUp Maintenance Process* foi herdado e o mecanismo *supressedBreakdownElement* foi liberado em *sSPEM Tool* para realizar as exclusões necessárias. Vale destacar que que como todo processo *OpenUp Maintenance Process* foi herdado, o processo sendo adaptado não possui inconsistências, uma vez que nesse momento da avaliação ele representa uma cópia exata do processo *OpenUp Maintenance Process* (que é 100% consistente).

Para justificar as exclusões de elementos e relações através do mecanismo *supressedBreakdownElement,* foi considerado algumas características do projeto de manutenção para o qual está sendo criado um novo processo. Inicialmente, o projeto de manutenção em questão foi considerado como uma manutenção evolutiva que não inclui e/ou exclui funcionalidades, necessidades e atores do sistema. A ideia é que apenas alguns casos de uso fossem modificados para contemplar algumas alterações de funcionalidades do sistema. Com base nesses fatos, considerou-se que nenhuma alteração seria necessária nos produtos de trabalho *Vision* e *Glossary*. Além disso, para esse tipo de projeto, ponderou-se que não seria precisa a revisão de requisitos (tarefa *Verify Requirements* e produto de trabalho *Requirements Verification*).

Partindo do contexto acima, as primeiras exclusões realizadas neste cenário foram os 2 produtos de trabalho citados acima (*Vision* e *Glossary*). Contudo, apenas para demonstrar com mais detalhes a análise de impacto realizada durante a exclusão de um elemento apenas um produto de trabalho foi excluído por vez, iniciando-se as operações de exclusão pelo produto de trabalho *Vision*. Neste momento, como *sSPEM Tool* implementa uma funcionalidade de análise de impacto, os elementos afetados pela operação de exclusão foram mostrados, conforme pode ser visto na Figura 83.

A Figura 83 mostra somente parte dos elementos afetados pela exclusão do produto de trabalho *Vision*. Contudo, através desta figura é possível observar que, em *sSPEM Tool*, os elementos afetados por uma operação de exclusão são destacados com ícones que indicam novas exclusões ou ícones de alerta. Isso ocorre porque, devido à

exclusão de um elemento, outros elementos são afetados. Especificamente no caso do produto de trabalho *Vision,* observa-se que alguns elementos necessitam ser apagados (os que possuem o ícone ✘) e alguns necessitam ser apenas alterados (os que possuem o ícone ⚠). Especificamente, além do produto de trabalho *Vision* alguns relacionamentos são apagados e alguns elementos sofrem alterações como, por exemplo, a tarefa *Update Vision* que não possuiria mais este produto de trabalho como parâmetro de entrada. Vale destacar que a tarefa *Update Vision* não foi excluída pois o produto de trabalho *Vision* foi definido como um parâmetro opcional para esta tarefa.

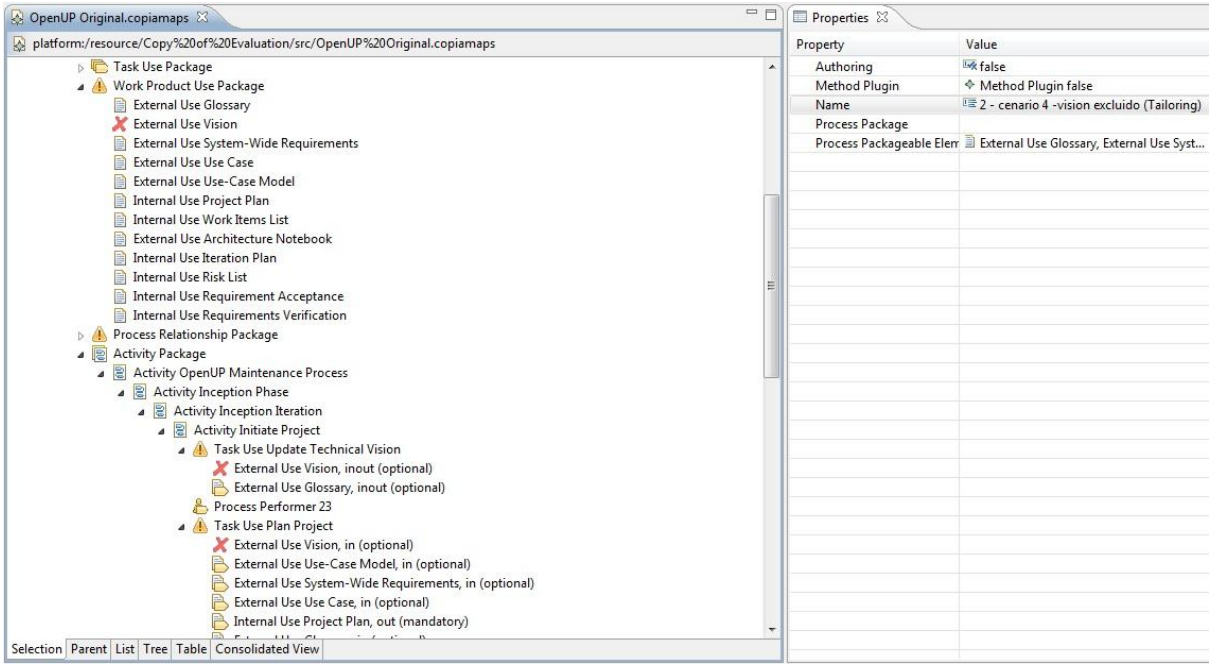

Figura 83 - Elementos afetados pela exclusão do produto de trabalho *Vision*

Depois de confirmada a exclusão do produto de trabalho *Vision,* as próximas operações realizadas em *sSPEM Tool* foram a exclusão do produto de trabalho *Glossary,* e, na sequencia, da tarefa *Verify Requirements* (ambos considerados opcionais). As Figuras 84 e 85 mostram, respectivamente, parte dos elementos afetados durante essas operações.

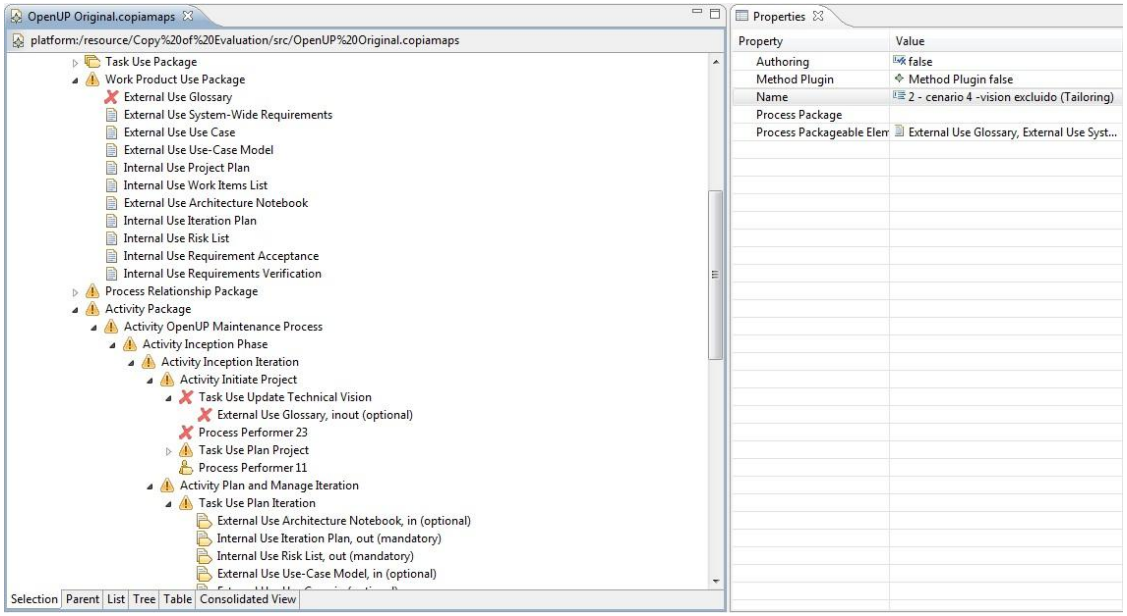

Figura 84 - Elementos afetados pela exclusão do produto de trabalho Glossary

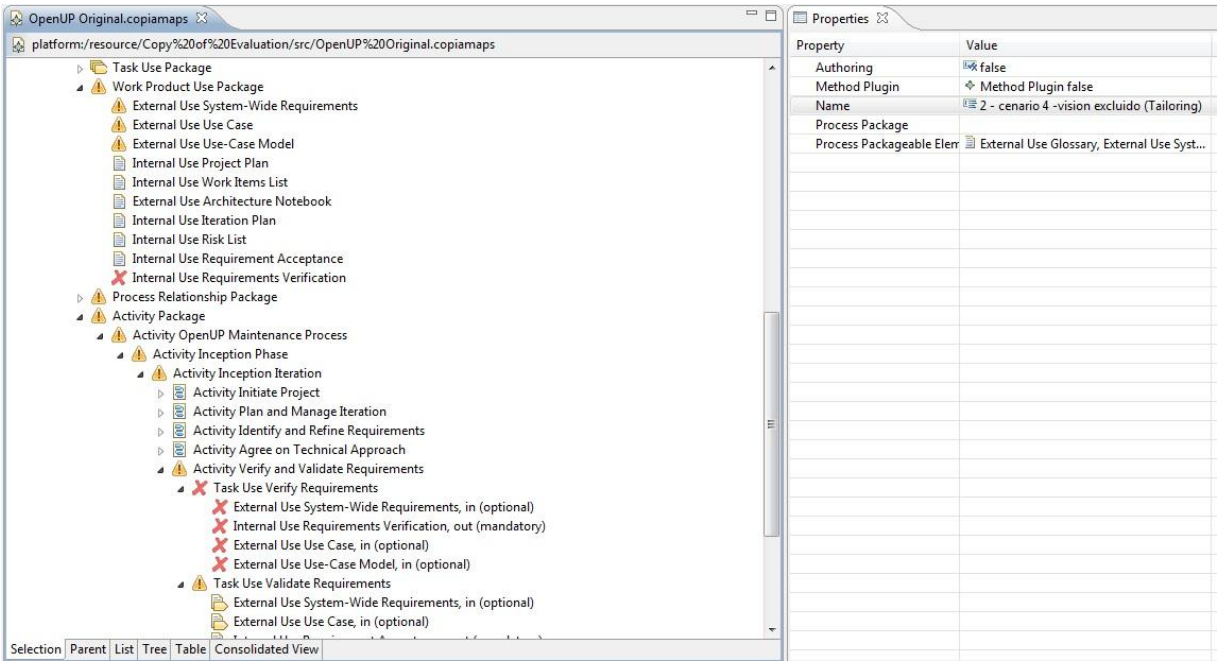

Figura 85 - Elementos afetados pela exclusão da tarefa *Verify Requirements*

Após a confirmação de todas as exclusões acima, considera-se que o novo processo, agora, adaptado, foi gerado. Como esse novo processo não possuía nenhuma inconsistência antes das exclusões acima e considerando que o mecanismo supressedBreakdownElement possui uma análise de impacto que objetiva respeitar as regras de boa-formação para consistência definidas nesta pesquisa, assume-se que o novo processo permaneceu com a quantidade de erros igual a 0. Para confirmar este fato, neste momento, o framework de validação foi executado novamente e seu resultado é exibido na Figura 86.

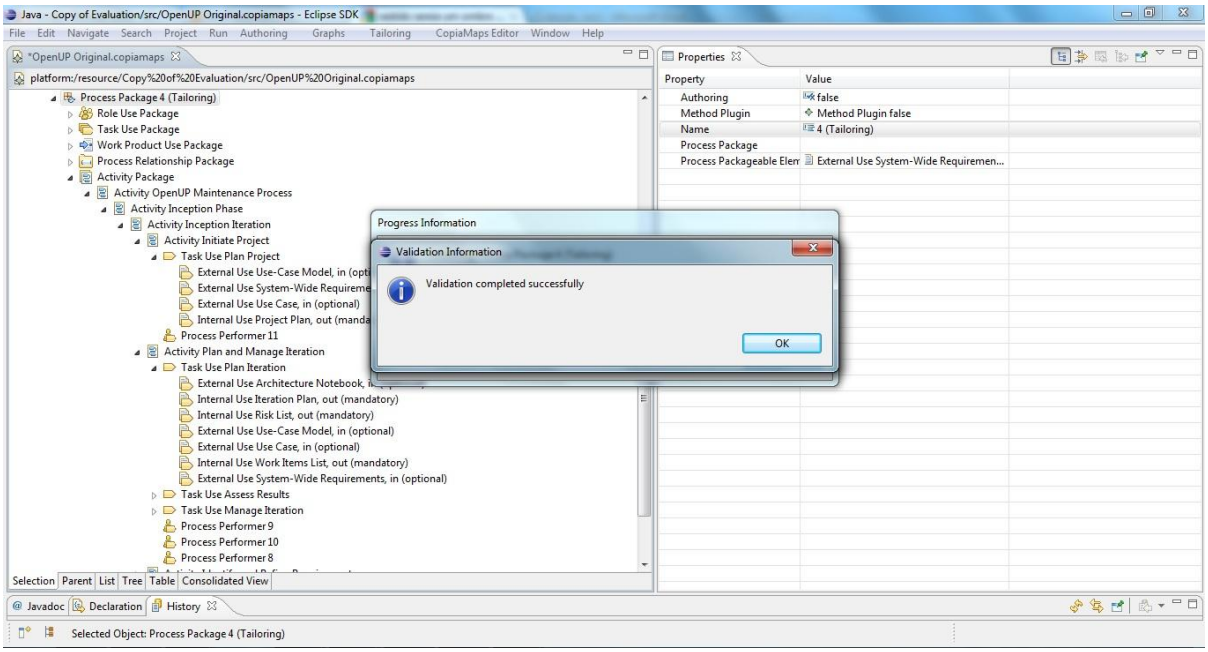

Figura 86 - Validação do Cenário 4 após operações de exclusão – processo 100% consistente

A Figura acima mostra que nenhuma das operações de exclusão causou inconsistências ao novo processo. Contudo, é possível ainda incluir novos elementos e relacionamentos nesse processo através das operações de inclusão.

Nesta avaliação, para iniciar a execução das operações de inclusão, incluiu-se propositalmente informações já existentes no processo. O objetivo é mostrar que as regras de boa-formação relacionadas com aspecto de duplicidade indicarão problemas nas operações de inclusão. Todas as inclusões de elementos e relacionamentos a seguir foram incluídos sem a utilização do *Method Content*, ou seja, foram criados diretamente no processo (utilizando-se somente de classes *Use*).

Os primeiros elementos adicionados ao processo foram os produtos de trabalho *Requirement Acceptance* e *System-Wide Requirements* e o papel *Analyst*. Como o objetivo de tais inclusões foi a duplicação, os produtos de trabalho incluídos foram definidos, respectivamente, como interno e externo. Ainda, em um primeiro momento, todos os elementos adicionados não foram conectados a nenhum outro elemento do processo.

Para verificar quais inconsistências as inclusões acima causaram no processo deste cenário, o *framework* de validação foi acionado. O resultado é exibido na Figura 87.

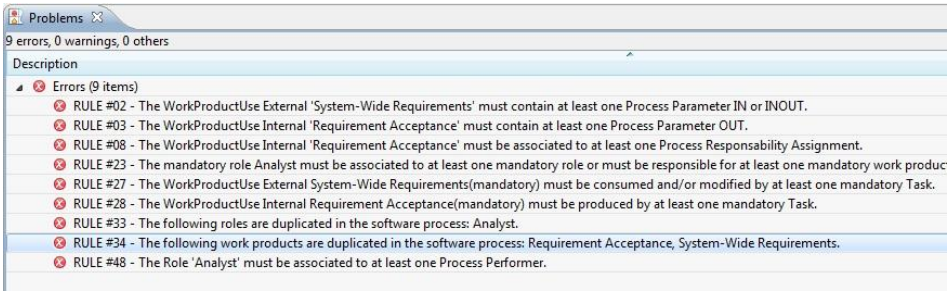

Figura 87 - Validação do Cenário 4 após primeiras operações de inclusão de elementos

O resultado acima mostra que várias inconsistências foram geradas pelas inclusões realizadas. Além de problemas relacionados com duplicidade (*Rule #33* e *Rule 34*), os quais já eram esperados, outros erros foram exibidos.

Os dois primeiros erros (*Rule #02* e *#03*) da Figura 87 foram exibidos porque produtos de trabalho necessitam ser consumidos, produzidos e/ou modificados em pelo menos uma tarefa de um processo de software. Especificamente nesse exemplo, o produto de trabalho externo *System-Wide Requirements* necessita ser consumido e/ou modificado por, pelo menos, uma tarefa e o produto de trabalho interno *Requirement Acceptance* necessita ser produzido também por pelo menos uma tarefa.

Os erros *Rule #23, #27* e *#28* são relacionados com o aspecto de opcionalidade. Tais erros ocorreram, pois todos elementos incluídos foram definidos como obrigatório e, dessa forma, necessitam estar com conectados com alguns outros elementos obrigatórios no processo. Já os erros *Rule #8* e *Rule #48* são relacionados com a necessidade dos elementos incluídos *Requirement Acceptance* (definido como um produto de trabalho do tipo interno) e *Analyst*, relacionarem-se, respectivamente, com, ao menos, uma instância da metaclasse *ProcessResponsabilityAssignment* e com, ao menos, uma instância da metaclasse *ProcessPerformer*.

Para finalizar as operações de inclusão, dois novos relacionamentos foram criados no processo deste Cenário. O objetivo de tais inclusões foi novamente avaliar o aspecto de duplicidade. Contudo, neste ponto da avaliação, as novas inclusões violam as regras de duplicação de relacionamentos. Os relacionamentos incluídos foram: um novo *ProcessPerformer* definindo mais uma vez o papel *Tester* como desempenhador da tarefa *Update Test Cases* e um novo *ProcessResponsabilityAssignment* definindo novamente que o papel *Project Manager* é responsável pelo produto de trabalho *Project Plan*. As inclusões desses relacionamentos violam as regras *Rule #35* e *#36*, conforme mostra a Figura 88 que exibe o resultado da execução do *framework* de validação.

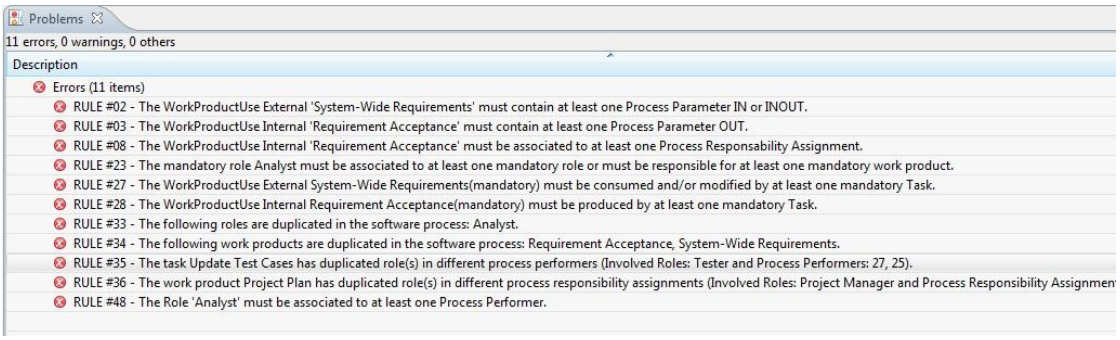

Figura 88 - Validação do Cenário 4 após primeiras operações de inclusão de relacionamentos

Para finalizar a primeira etapa da avaliação deste cenário os erros acima foram corrigidos. Para isso, foi necessária apenas a exclusão dos elementos relacionados com tais erros. O que não causou nenhum impacto ao processo, uma vez que, como já explicado acima, todos esses elementos já existiam.

A segunda e última etapa da avaliação do Cenário 4 envolve outro exemplo de utilização dos mecanimos para adaptação do metamodelo sSPEM 2.0. O objetivo, neste ponto da avaliação, é herdar apenas uma pequena parte (1 atividade) de um dos processos disponíveis em *sSPEM Tool* para avaliar se a herança de elementos a partir de processos diferentes e realizada em partes, acontece de forma consistente.

Para iniciar esta etapa da avaliação do Cenário 4 os primeiros passos foram a criação de uma instância da metaclasse *ProcessPackage* (com valor de atributo *isAuthoring* = *false*) como parte da instância de *MethodPlugin*, a criação de uma instância da metaclasse *ActivityPackage* e a criação de uma instância da metaclasse *Activity*. Depois disso, o próximo passo foi a escolha da parte e do processo ao qual seria aplicado o mecanismo de adaptação.

Nesse momento, optou-se por herdar uma parte de um processo que não contenha inconsistências e assim, utilizou-se o processo já usado no início da avaliação, ou seja, o Cenário 4 antes das operações de exclusão (conforme Figura 82). A atividade escolhida para ser herdada foi a atividade *Agree on Technical Approach*, que é mostrada na Figura 89. A motivação para escolha dessa atividade é fato que suas tarefas modificam o produto de trabalho *Architecture Notebook* que possui dependências com o produto de trabalho *System-Wide Requirements.* Além disso, suas tarefas consomem os produtos de trabalho *Use-Case Model* e *Use-Case*, os quais mantém relação de composição no processo *OpenUP,* indicando que o produto de trabalho *Use-Case Model* é o todo do produto de trabalho *Use-Case*.

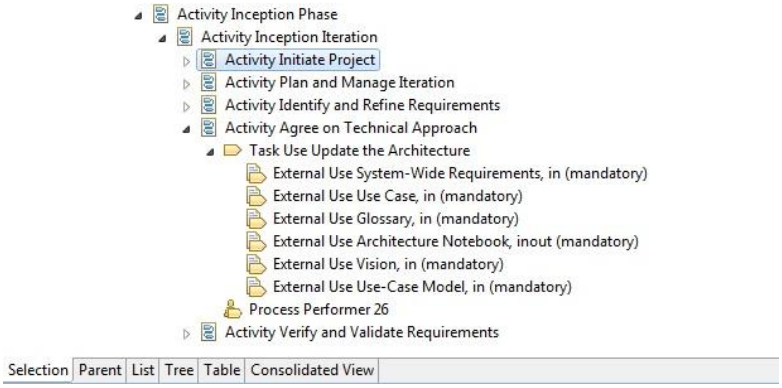

Figura 89 – Entradas e Saídas da Atividade *Agree on Technical Approach*

Antes de herdar a atividade em questão, uma modificação foi realizada propositalmente nos parâmetros de entrada e saída da tarefa *Update the Architecture*. Foram excluídos os parâmetros de entrada para os produtos de trabalho *System-Wide Requirements* e *Use-Case,* indicando que a tarefa *Update the Architecture* não consome mais estes produtos de trabalho no processo *OpenUP*. Após estas exclusões, a atividade *Agree on Technical Approach* foi herdada através do mecanismo *usedActivity* (*extension*) para o novo processo criado em *sSPEM Tool*. O resultado desta herança é mostrado na Figura 90.

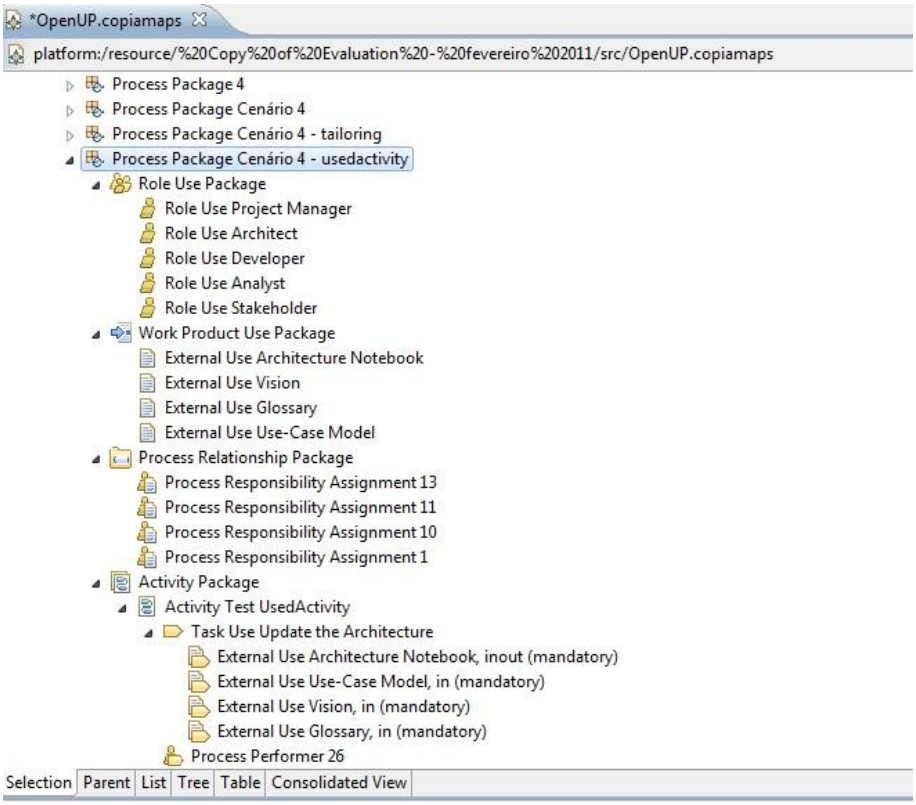

Figura 90 - Resultado da Herança da Atividade *Agree on Technical Approach*

A Figura 90 mostra que todos os produtos de trabalho e papéis envolvidos na atividade *Agree on Technical Approach* foram herdados, bem como os relacionamentos que definem as responsabilidades para os produtos de trabalho. Isso ocorreu devido às modificações realizadas nesta pesquisa para a semântica do mecanismo de adaptação *extension* realizadas com objetivo de herdar conteúdos consistentes para processos diferentes. Em verdade, as heranças foram possíveis, pois os elementos de processo utilizados pela atividade herdada estavam definidos nos pacotes e desta forma, eram considerados globais no metamodelo, podendo ser acessados por qualquer atividade.

Para finalizar esta etapa da avaliação do Cenário 4 o *framework* de validação foi executado e o resultado é mostrado na Figura 91.

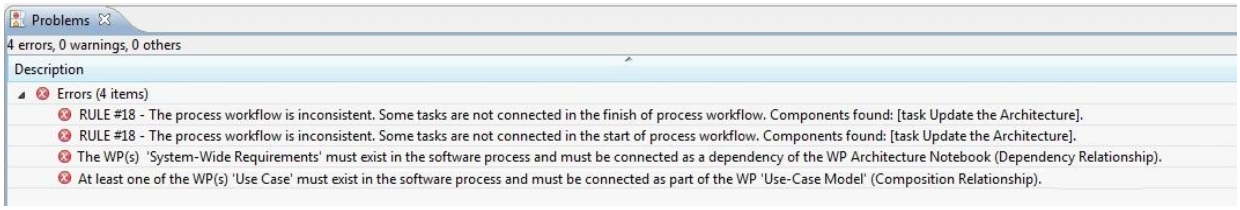

Figura 91 – Resultado do *Framework* de Validação para a Herança da Atividade *Agree on Technical Approach*

A Figura 91 apresenta 4 inconsistências para a atividade herdada. Duas delas são relacionadas com a regra de boa-formação *Rule #18* e informam que nenhum sequenciamento foi definido para o processo de software. Os outros dois erros encontrados já eram esperados e não estão relacionados especificamente com nenhuma regra de boa-formação. Esses erros aparecem devido as alterações realizadas nas semânticas dos tipos de herança *extension* e *localContribution* do mecanismo *usedActivity*. Inicialmente, o primerio erro indica que o produto de trabalho herdado *Architecture Notebook* possui dependências com o produto de trabalho *System-Wide Requirements* e que essa dependência não foi herdada. Já o segundo erro informa que o produto de trabalho herdado *Use-Case Model* era o todo de uma relação de composição no processo de onde ele foi herdado e, desse modo, é necessário que uma de suas partes seja também herdada e relacionada como sua parte no novo processo.

A partir dos resultados acima, nota-se que embora todo conteúdo da atividade *Agree on Technical Approach* tenha sido herdado, não foi possível herdar algumas de suas dependências que se encontravam definidas em outros pontos do processo de software. O que se pretendeu demonstrar a partir desse último exemplo, é que embora o conteúdo herdado apresente inconsistências, elas são sempre indicadas pelo *framework*

de validação, para que nenhuma das premissas de consistência definidas nesta pesquisa seja violada.

Concluído o exemplo acima, considera-se encerrada a avaliação do Cenário 4. A Tabela 21 relaciona a análise das premissas e regras de boa-formação atendidas e não atendidas nesse cenário.

Tabela 21 - Premissas e regras analisadas no Cenário 4

Premissas e Regras Atendidas		**Premissas e Regras Não Atendidas**	
Premissa	**Regra de Boa-Formação**	**Premissa**	**Regra de Boa-Formação**
P #1 **P #2** **P #3**	*Rule #1*	**P #1** **P #2** **P #3**	*Rule #2* e *#3*
P #4	*Rule #5, #7 e #49*	**P #6**	*Rule #8* e *#24*
P #5	*Rule #5*	**P #9**	*Rule #20*
P #7	*Rule #4, #9, #10 e #21*	**P #14**	*Rule #48*
P #8	*Rule #11 e #19*	**P #15** **P #16** **P #17** **P #18** **P #19** **P #20**	*Rule #23, #27, #28, #30 e #32*
P #9	*Rule #19*	**P #21** **P #22**	*Rule #33, #34, #35* e *#36*
P #10 **P #11** **P #12**	*Rule #6, #12, #13, #14, #15, #16, #17, #18, #41, #42, #43, #44, #45 e #50*	-	-
P #13	*Rule #22*	-	-
P #15 **P #16** **P #17** **P #18** **P #19** **P #20**	*Rule #25, #26, #29* e *#31*	-	-
P #21 **P #22**	*Rule #37, #38, #39 e #40*	-	-

7.2.7 Cenário 5

O Cenário 5 foi criado apenas para demonstrar inconsistências relacionadas com algumas regras de boa-formação que não foram citadas até este momento (*Rule #1, #4, #5, #6, #7, #9, #10, #14, #15, #21, #22, #25, #26,#37, #40, #41, #42, #44 e #45*).

O processo utilizado neste ponto da avaliação foi o processo final do Cenário 4, ou seja, o processo após todas operações de exclusão realizadas e que não possui erros (conforme Figura 86). Basicamente, nesse cenário, somente informações inconsistentes foram incluídas no processo (sem a utilização do *Method Content*) com objetivo de ocasionar alguns erros. Após tais inclusões, o *framework* de validação foi sempre acionado para demonstrar que os erros incluídos de forma proposital são encontrados.

Inicialmente, para avaliar as regras *Rule #6, #14, #15, #41, #42, #44 e #45* que são relacionadas com os nodos inicial e final do processo, os elementos listados na Tabela 22 foram incluídos.

Tabela 22 - Relação de elementos relacionados com nodos inicial e final incluídos no Cenário 5

Elemento Incluído
Activity END (atributo *specialNode = 'end'*)
WorkSequence 25 (atributo *linkKind = finishToStart*; atributo *predecessor = Task Use Plan Project* e atributo *sucessor = Task Use Start*)
WorkSequence 26 (atributo *linkKind = finishToStart*; atributo *predecessor = Task Use End* e atributo *sucessor = Task Use Plan Project*)
Process Performer 26 (atributo *linkedRoleUse = Analyst*; atributo *linkedTaskUse = Task Use End*)
ProcessParameter TEST START (incluído na *TaskUse Start*)

Além das inclusões acima, uma alteração foi realizada na *WorkSequence* 20 que liga a *Task Use Assess Results* com a *Task Use End*. A alteração feita foi a mudança do valor do atributo *linkKind* de *finishToStart* para *startToStart*.

Neste momento da avaliação o *framework* de validação foi executado. O resultado desta validação é apresentado na Figura 92.

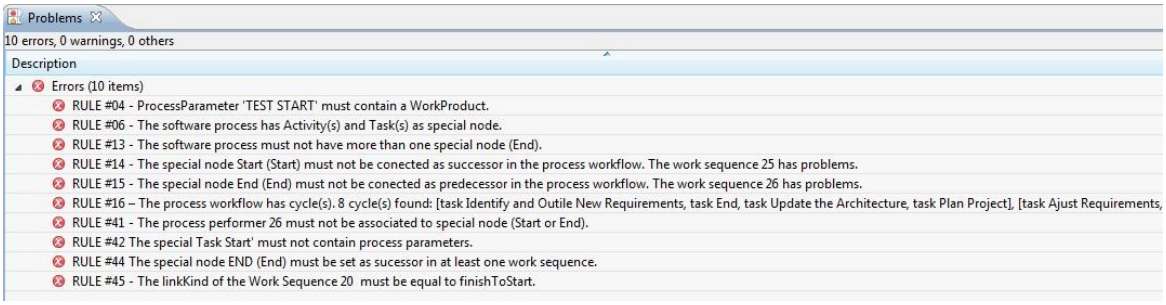

Figura 92 - Validação do Cenário 5 após inclusões de elementos no processo relacionados com nodos especiais

O resultado acima mostra que grande parte erros provém dos elementos incluídos. Como já esperado, a maioria desses erros é relacionado a problemas com nodos especiais do processo. Por exemplo, a regra *Rule #06* mostra que o processo possui *Activity* e *Task* como nodo especial. Isso quer dizer que atividades e tarefas estão definidas ao mesmo tempo como início e/ou fim no processo.

Prosseguindo a avaliação outros elementos foram incluídos no processo. A Tabela 23 lista estes elementos e a Figura 93 exibe o resultado do *framework* de validação do processo após a inclusão realizada.

Tabela 23 - Relação de elementos incluídos no Cenário 5

Elemento Incluído
*Activity EMPTY(*atributo *isOptional = 'false')*
*Activity TEST (*atributo *isOptional = 'true')*

Task Use EMPTY (incluída na *Activity TEST* com atributo *isOptional = 'false'*)
External Use TEST EXTERNAL (incluído na *Activity TEST*)
Process Parameter TEST PP (incluído na *Activity TEST* com atributo *direction = 'out'*; atributo *parameterType = External Use TEST EXTERNAL*)
Process Parameter DUPLICATED (incluído na *Activity Initiate Project* com atributo *direction = 'inout'*; atributo *parameterType = External Use Vision*)
Work Product Use Relationship 7 (atributo *relationType* = '*composition'*; atributo *source = External Use Test Case*; atributo *target = External Use Use Case*)
Work Product Use Relationship 8 (atributo *relationType* = '*composition'*; atributo *source = External Use Use Case*; atributo *target = External Use Use-Case Model*)
InternalUse TEST 1 (incluído na *Activity TEST* com atributo *isOptional* = '*false'*)
InternalUse TEST 2 (incluído na *Activity TEST* com atributo *isOptional* = '*false'*)
InternalUse TEST 3 (incluído na *Activity TEST* com atributo *isOptional* = '*false'*)
Work Product Use Relationship 9 (atributo *relationType* = '*dependency*; atributo *source = InternalUse TEST 1*; atributo *target = InternalUse TEST 2*) (incluído 2 vezes)
Work Product Use Relationship 10 (atributo *relationType* = '*agregation*; atributo *source = InternalUse TEST 3*; atributo *target = InternalUse TEST 1 e InternalUse TEST 2*)

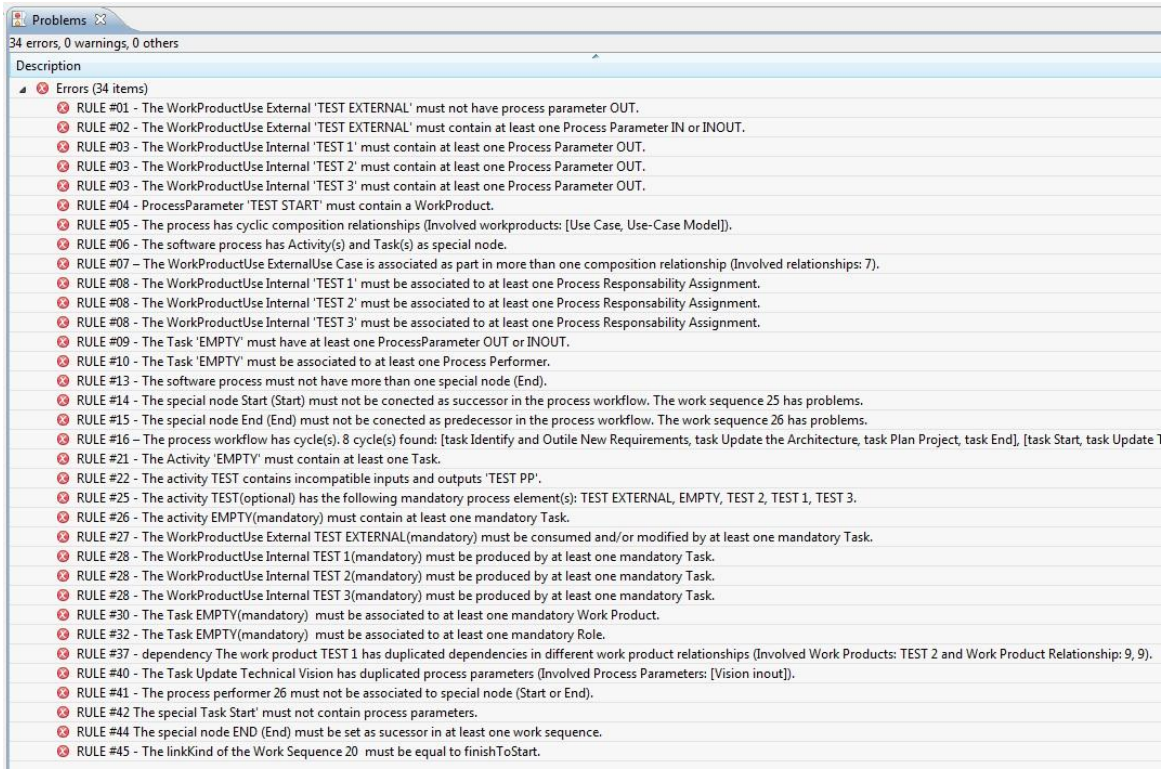

Figura 93 - Validação do Cenário 5 após as últimas inclusões de elementos neste Cenário

Os erros exibidos na Figura 93 mostram que todos os elementos incluídos apresentam algum problema. Esse é o caso, por exemplo, da *Activity TEST* que exibe o erro da regra *Rule #22,* pois possui parâmetros incompatíveis com os parâmetros das suas tarefas. Isso ocorre porque foi definido um *Process Parameter* dentro da *Activity TEST* do tipo *out* para o novo produto de trabalho *TEST EXTERNAL* e o mesmo *Process Parameter* não existe em nenhuma de suas tarefas internas. Essa situação indica que o produto de trabalho *TEST EXTERNAL* é produzido em alguma das tarefas da *Activity TEST.* Contudo, isso não é verdade, pois nenhuma tarefa desta atividade produz o novo produto de trabalho. Outro problema, ainda relacionado com essa situação, é que um

Process Parameter do tipo *out* foi associado para um produto de trabalho do tipo externo, sendo que tal fato viola a regra de boa-formação *Rule #1*.

Considerando que as últimas regras de boa-formação foram testadas a execução do Cenário 5 foi encerrada. A Tabela 24 relaciona a análise das premissas e regras de boa-formação analisadas neste Cenário.

Tabela 24 - Premissas e regras analisadas no Cenário 5

Premissas e Regras Atendidas		**Premissas e Regras Não Atendidas**	
Premissa	**Regra de Boa-Formação**	**Premissa**	**Regra de Boa-Formação**
P #1 **P #2** **P #3**	*Rule #3*	**P #1** **P #2** **P #3**	*Rule #1* e *#2*
P #4	*Rule #49*	**P #4**	*Rule #5 e #7*
P #5	*Rule #5*	**P #7**	*Rule #4, #9, #10 e #21*
P #6	*Rule #8* e *#24*	**P #10** **P #11** **P #12**	*Rule #6, #13, #14, #15, #16, #41, #42, #44 e #45*
P #8	*Rule #11 e #19*	**P #13**	*Rule #22*
P #9	*Rule #19* e *#20*	**P #15** **P #16** **P #17** **P #18** **P #19** **P #20**	*Rule #25, #26, #27, 30 e #32*
P #10 **P #11** **P #12**	*Rule #12, #17, #18, #43 e #50*	**P #21** **P #22**	*Rule #37 e #40*
P #14	*Rule #48*	-	-
P #15 **P #16** **P #17** **P #18** **P #19** **P #20**	*Rule #23, #28, #29* e *#31*	-	-
P #21 **P #22**	*Rule #33, #34, #35, #36,#38 e #39*	-	-

7.3 Considerações Finais

Este capítulo apresentou cinco cenários para definição e adaptação de processos de software que demosntraram a aplicação do metamodelo sSPEM 2.0 e do conjunto de regras de boa-formação para consistência. Toda avaliação foi desenvolvida, utilizando-se do processo de software *OpenUp* e conduzida no protótipo de ferramenta *sSPEM Tool*.

Como pôde ser visto ao longo do capítulo, foram utilizadas na avaliação todas as informações (metaclasses e relacionamentos) incluídas no metamodelo original SPEM 2.0 pela extensão proposta nesta pesquisa (metamodelo sSPEM 2.0). Esse é o caso, por exemplo, dos relacionamentos de dependência entre os produtos de trabalho, que

embora não estivessem modelados no processo *OpenUP,* puderam ser facilmente constatados na descrição textual deste processo. Outro tipo de informação que se aplicou muito bem nesta avaliação foi o uso dos produtos de trabalho do tipo *External* que foram utilizados no Cenário 3 para a confecção de um processo específico para um projeto de manutenção. De acordo com OMG [Omg07a], para esses tipos de projeto, constata-se a necessidade de diferenciar os produtos de trabalho que são produzidos ou somente modificados, uma vez que muitos desses produtos de trabalho já se encontram documentados nestes projetos de software.

Durante a condução da avaliação, 48 regras de boa-formação (conforme Capítulo 4) para consistência relacionadas com a atividade de definição de processos foram avaliadas e demonstradas através da inclusão de erros no processo *OpenUP*. As únicas regras de boa-formação que não foram avaliadas através da inclusão de erros, foram as 2 regras específicas dos mecanismos *usedActivity* e *Variability* (*Regra #46* e *Regra #47*). Isso ocorreu porque em *sSPEM Tool* estas regras foram implementadas através de filtros aplicados sobre os atributos *usedActivity* e *variabilityBasedOnElement*, os quais não permitem que elementos do processo ou do repositório de conteúdo definam um auto-relacionamento para utilização dos mecanimos *usedActivity* e *Variability.* Essa implementação faz com que nenhum elemento em *sSPEM Tool* possa herdar (no caso do mecanismo *usedActivity*) ou variar (no caso do mecanismo *Variability*) seu próprio conteúdo, o que respeita a definição das regras *Regra #46* e *Regra #47.*

Também foram demonstrados na avaliação alguns exemplos dos mecanismos de adaptação *usedActivity* e *supressedBreakdownElement.* Durante a condução desses exemplos, foi possível demonstrar o funcionamento da análise de impacto proposta para a exclusão de elementos em um processo de software.

Por fim, vale destacar que durante a criação dos cenários para esta avaliação foi utilizado o repositório de conteúdo (*Method Content*) em *sSPEM Tool*. Embora não tenha sido explorado na descrição do texto os resultados de cada apontamento realizado para o repositório, deixou-se claro que para criação dos Cenários 1, 2 e 3 todas as informações foram provenientes de reuso de elementos e relacionamentos.

8 CONCLUSÕES E TRABALHOS FUTUROS

Apesar da importância da definição e adaptação de processos de software consistentes, constatou-se, a partir da literatura analisada, que o aspecto da consistência de processos é abordado de forma parcial nos estudos. Muitos autores propõem soluções para o tratamento da consistência apenas nas atividades de definição dos processos de software, ou, abordam este aspecto, somente durante as atividades de adaptação desses processos.

Ainda, considerando as pesquisas que apresentam regras para definição de processos de software consistentes, pôde-se verificar que, em todos os estudos analisados, apenas alguns aspectos de consistência e/ou elementos de um processo de software são tratados, como por exemplo, o estudo apresentado em [Bao08], que aborda apenas o aspecto de consistência relacionado com a definição de um sequenciamento de tarefas.

Nesta tese, como forma de viabilizar a definição e adaptação de processos de software consistentes, uma infraestrutura para consistência foi definida utilizando o metamodelo SPEM 2.0, o qual, é tido nos dias de hoje pela OMG, como principal referência para modelagem de processos de software.

Para construção da infraestrutura, inicialmente, foi definido um conjunto de premissas a serem observadas durante as atividades de definição e adaptação dos processos de software. Tais premissas foram identificadas com base na literatura analisada, nos processos de software RUP, OPEN e *OpenUP* e também a partir do metamodelo SPEM 2.0. Para atendimento dessas premissas, foi definida uma extensão ao metamodelo SPEM 2.0 (sSPEM 2.0) com a inclusão e alteração de um conjunto de elementos e relacionamentos, bem como foi desenvolvido um conjunto de regras de boa-formação para consistência que estabelecem as condições que devem ser verdadeiras para todos os processos de software (premissas) em qualquer ponto do seu ciclo de vida.

As regras de boa-formação devem ser utilizadas durante a definição de um processo de software e, necessitam ser respeitadas quando um processo de software é adaptado. Nesse sentido, para auxiliar as atividades de adaptação de processos, foi também desenvolvido, nesta pesquisa, a análise de impacto sobre os elementos e relacionamentos que são afetados durante a condução de alguns mecanismos de adaptação do metamodelo sSPEM 2.0. A principal contribuição da análise de impacto é

garantir que as dependências de um processo de software serão respeitadas durante uma operação de adaptação, evitando assim, que inconsistências sejam geradas no processo resultante.

Um fato importante sobre as regras de boa-formação, é que, elas podem ser utilizadas de maneira independente. Pesquisados e/ou implementadores de ferramentas podem utilizar as regras de boa-formação de acordo com os pacotes do metamodelo sSPEM 2.0. Por exemplo, no caso de um pesquisador ter interesse nas regras definidas para o pacote *Process Structure*, seria possível a utilização de somente um subconjunto das regras de boa-formação para consistência. Isso seria possível, uma vez que como pôde ser observado no Capítulo 4, todas as regras de boa-formação são incluídas de maneira incremental utilizando-se do mecanismo de *merge* da UML. Salienta-se apenas que as regras de boa-formação que envolvem os elementos e relacionamentos incluídos no metamodelo original SPEM 2.0 por esta pesquisa, necessitariam de maior atenção, já que várias metaclasses e relações do metamodelo sSPEM 2.0 seriam necessárias.

Além do metamodelo sSPEM 2.0 e das regras de boa-formação, é parte também da infraestrutura para consistência, o protótipo de ferramenta *sSPEM Tool,* que foi desenvolvido para viabiliar a utilização do metamodelo sSPEM 2.0 e das regras de boa-formação de forma automática. As principais funcionalidades implementadas no *sSPEM Tool* são as funcionalidades para definição e adaptação de processos de software, bem como a funcionalidade para validação desses processos, que consistiu-se na inclusão de todas as regras de boa-formação no *framework* de validação existente no *plugin* EMF.

Por fim, foi definido um guia para auxiliar a definição e adaptação de processos a partir do metamodelo sSPEM 2.0 e das regras de boa-formação. A contribuição desse guia é facilitar o uso da infraestrutura para consistência, uma vez que devido a grande quantidade de elementos e relacionamentos do metamodelo sSPEM 2.0, bem como a elevada quantidade de regras de boa-formação, seu uso se torna uma tarefa não trivial.

Embora todos os produtos acima possam ser destacadas como contribuições para a solução apresentada nesta tese, apresentam-se a seguir algumas sugestões para continuidade do trabalho.

8.1 Trabalhos Futuros

Durante a elaboração desta tese, algumas ideias adicionais foram identificadas e são aqui indicadas como sugestão para trabalhos futuros:

- ampliar o conjunto de regras de boa-formação para consistência das atividades de definição e adaptação de processos para contemplar todos os elementos e relações do metamodelo SPEM 2.0. Isto envolveria a criação de regras para as metaclasses *Guidance*, *Section*, *Composite Role*, *Team Profile* e *Qualification*.

- avaliar se solução proposta é aplicável em processos de software que não são compatíveis ao metamodelo SPEM 2.0. Isso seria necessário, pois, embora esse metamodelo seja atualmente considerado como principal referência para definição e adaptação de processos, não é possível afirmar que a infraestrutura para consistência, incluindo todas as regras de boa-formação, sejam aplicáveis a qualquer processo de software.

REFERÊNCIAS BIBLIOGRÁFICAS

[Atk03] Atkinson, C.; Kuhne, T. “Model-driven Development: A Metamodeling Foundation Software”. Software, IEEE, Vol. 20-5, Set-Out. 2003, pp. 36–41.

[Atk07] Atkinson, D. C.; Weeks, D. C.; Noll, J. “Tool Support for Iterative Software Process Modeling”. Information and Software Technology, Vol. 49-5, Maio 2007, pp. 493-514.

[Bao08] Bao, E. “A Study of Rationality Test Rules for Software Process Model”. In: International Conference on Information Management, Innovation Management and Industrial Engineering, 2008, pp. 28-32.

[Baj07] Bajec, M.; Vavpotic, D.; Krisper, M. “Practice-Driven Approach for Creating Project-Specific Software Development Methods”. Information and Software Technology, Vol. 49-4, Abril 2007, pp. 345-365.

[Ben07] Bendraou, R.; Combemale, B.; Crégut, X.; Gervais, M. “Definition of an eXecutable SPEM 2.0”. In: APSEC, IEEE Computer Society, 2007, pp. 390-397.

[Bor01] Borges, L. M. S.; Falbo R. A. “Gerência de Conhecimento sobre Processos de Software”. In: Anais do VIII Workshop de Qualidade de Software, 2001, 12p.

[Boe06] Boehm, L. H. B.; Lu, H. J. G.; Qian C. “Applying the Value/Petri Process to ERP Software Development in China”. In: International Conference on Software Engineering – ICSE’06, 2006, pp. 502-511.

[Bri96] Brinkkemper, S. “Method Engineering: Engineering of Information Systems Development Methods and Tools”. Information and Software Technology, Vol. 38-4, Abril 1996, pp. 275-280.

[Bur08] Burns, T.; Klashner, R.; Deek F. “An Empirical Evaluation of a Methodology – Tailoring Information System Development Model”. Journal of Software Process Improvement and Practice, Vol. 13-5, Setembro 2008, pp. 387-395.

[Cal08] Callegari, D.; Bastos, R. M. “A Systematic Review of Dynamic Reconfiguration of Software Projects”. In: Simpósio Brasileiro de Engenharia de Software – SBES’08, 2008, pp. 299-313.

[Chr03] Chrissis, M. B.; Korad, M.; Shrum, S. “CMMI Guidelines for Process Integration and Product Improvement”. Addisson-Wesley, 2003, 663p.

[Cod70] Codd, E. F. “A Relational Model of Data for Large Shared Data Banks”. Comunications of the ACM, Volume 13- 6, Outubro 1970, pp. 377-387.

[Col06] Collaris, R. A.; Dekker, E.; Warmer, J. “Tailoring RUP made easy: Introducing the Responsibility Matrix and the Artifact Flow”. Capturado em: http://www.ibm.com/developerworks/rational/library/sep06/collaris_dekker_warmer/index.html, Março 2011.

[Cug98] Cugola, G.; Ghezzi, C. “Software Processes: A Retrospective and a Path to the Future”. In: International Conference on Software Process - ICSP, 1998, pp.101-123.

[Dai07] Dai, F.; Li, T. “Tailoring Software Evolution Process”. In: International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing – ACIS’07, 2007, pp. 782 –787.

[Ecl08] Eclipse. “Eclipse modeling framework project (emf)”. Capturado em: http://www.eclipse.org/modeling/emf, Janeiro 2009.

[Egy07a] Egyed, A. “Fixing Inconsistencies in UML Design Models”. In: 29th International Conference on Software Engineering - ICSE’07, 2007, pp. 292–301.

[Egy07b] Egyed, A. “UML/Analyzer: A Tool for the Instant Consistency Checking of UML Models”. In: 29th International Conference on Software Engineering – ICSE’0, 2007, pp. 292–301.

[Emf07] E.M.F. Project. “Eclipse Modeling Framework (EMF)”. Capturado em: http://www.eclipse.org/modeling/emf/, Março 2007.

[Eng01] Engels, G.; Küster, J. M.; Heckel, R.; Groenewegen, L. “A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models”. In: Proceedings of Eighth European Software Engineering Conference Held Jointly with Nineth ACM SIGSOFT International Symposium on Foundations of Software Engineering - ESEC/FSE2001, 2001, pp.186–195.

[Fei93] Feiler, P.; Humphey, W. “Software Process Development and Enactment: Concepts and Definitions”. In: International Conference on The Software Process- ICSP, 1993, pp. 28 – 40.

[Fer10] Fuentes-Fernandez, R.; Garcia-Magarino, I.; Gomes-Rodriguez, A. M. “A Technique for Defining Agent-Oriented Engineering Process”. Engineering Applications of Artificial Intelligente, Vol. 23-3, Abril 2010, pp. 432-444.

[Fir02] Firesmith, D.; Henderson-Sellers, B. “The OPEN Process Framework – An Introduction”. Addison-Wesley: Harlow, 2002, 272p.

[Fit03] Fitzgerald B.; Russo N. L.; O’kane T. “Software Development Method Tailoring at Motorola”. Communications of the ACM, Volume 46-4, Abril 2003, pp. 64-70.

[Fra99] Franch, X.; Ribó, J. M. “Using UML for modelling the static part of a software process”. In: Proceedings of the 2nd International Conference on the Unified Modeling Language, 1999, pp. 292-307.

[Fug00] Fuggetta, A. “Software Process: A Roadmap”. In: Proceedings of the Conference on The Future of Software Engineering - ICSE '00, 2000, pp. 25-34.

[Gam93] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. “Design Patterns: Abstraction and Reuse of Object-oriented Design”. In: '93 Proceedings of the 7th European Conference on Object-Oriented Programming, 1993, pp. 406-431.

[Gin95] Ginsberg, M. P.; Quinn, L. H. “Process Tailoring and the Software Capability Maturity Model”. Technical Report, 2005, 60p. Capturado em: http://www.sei.cmu.edu/library/abstracts/reports/94tr024.cfm, Maio 2010.

[Han05] Hanssen, G. K.; Westerheim, H.; Bjornson, F. O. “Tailoring RUP to a Defined Project Type: A Case Study”. In: Proceedings of Product Focused Software Process Improvement - PROFES, 2005, pp. 314-327.

[Har94] Harmsen, F. S.; Brinkkemper, H. “Situational Method Engineering for Information System Projects. Methods and Associated Tools for the Information Systems Life Cycle”. In: Proceedings of the IFIP WG8.1 Working Conference CRIS’94, 1994, pp. 169-194.

[Har97] Harmsen, A. F. “Situational Method Engineering”. Moret Ernst & Young, 1997, 223p.

[Hug09] Hug, C.; Front, A.; Rieu, D.; Henderson-Sellers, B. “A Method to Build Information Systems Engineering Process Metamodels”. The Journal of Systems and Software, Vol. 82-10, Outubro 2009, pp. 1730-1742.

[Hum89] Humphrey, W. S. “Managing the Software Process”. Addison-Wesley, 1989, 512p.

[Hsu08] Hsueh, N. L.; Shen, W. H.; Yang, Z. W.; Yang, D. L. “Applying UML and Software Simulation for Process Definition, Verification and Validation”. Information and Software Technology, Vol, 50-9, Agosto 2008, pp. 897-911.

[Hug09] Hug, C.; Front, A.; Rieu, D.; Henderson-Sellers, B. “A Method to Build Information Systems Engineering Process Metamodels”. The Journal of Systems and Software, Vol. 82-10, Outubro 2009, pp. 1730-1742.

[Ibm09] IBM. “IBM Rational Software Modeler”. Capturado em: http://www-01.ibm.com/software/awdtools/modeler/swmodeler/, Março 2011.

[Iee98] IEEE, Institute. “IEEE Std. 1219-1998. IEEE Standard for Software Maintanance”. Institute of Electrical and Electronics Engineers - IEEE. Capturado em: http://homes.ieu.edu.tr/~kkurtel/Documents/IEEE%20Std%201219-1998%20Software%20Maintenance.pdf, Maio 2011.

[Iso95] ISO/IEC 12207. “Information Technology – Software Life-Cycle Processes”. Technical Report. Capturado em: http://www.iso.org/iso/iso_catalogue/catalogue_tc/cataloguedetail.htm?csnumber=21208, Maio 2011.

[Iso98] ISO/IEC TR 15504. “Information Technology – Software Process Assessment”. Technical Report. Capturado em: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38932, Maio 2011.

[Jac01] Jacobson, I.; Booch G.; Rumbaugh J. “The Unified Software Development Process”. Upper Saddle River, Addison Wesley, 2001, 512p.

[Jal02] Jalote, P. “CMM in Practice. Processes for Executing Software Projects at Infosys”. The SEI Series in Software Engineering, 2002, 400p.

[Kan08] Kang, D.; Song I. G.; Park S.; Bae D. H.; Kim H. K. “A Case Retrieval Method for Knowledge-Based Software Process Tailoring Using Structural Similarity”. In: Proceedings of the 15th Asia-Pacific Software Engineering Conference, 2008, pp. 51-58.

[Kar01] Karlsson, F.; Agerfalk, P. J.; Hjalmarsson, A. “Method Configuration with Development Tracks and Generic Project Types”. In: International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design – EMMSAD, 2001, 12p.

[Kel96] Kellner, M. I. “Connecting reusable software process elements and components”. In: Proceedings of the International Software Process Workshop, 1996, pp. 8-11.

[Kit04] Kitchenham, B., “Procedures for performing systematic reviews”. Technical report Software Engineering Group, Department of Computer Science, Keele University. Capturado em: http://www.cin.ufpe.br/~in1037/leitura/systematicReviewSE-COPPE.pdf, Fevereiro 2011.

[Kor10] Kornyshova, E.; Deneckere, R.; Claudepierre B. “Contextualization of Method Components”. In: Research Challenges in Information Science – RCIS, 2010, pp. 235-246.

[Kru00] Kruchten, P. “The Rational Unified Process: An Introduction”. Upper Saddle River, NJ: Addison-Wesley, 2000, 298p.

[Lal08] Laleau, R.; Polack, F. “Using Formal Metamodels to Check Consistente of Functional Views in Information Systems Specification”. Information Software Technology, Vol. 50-7, Junho 2008, pp. 797–814.

[Lee02] Lee, S.; Shim, J.; Wu, C. “A metal model approach using uml for task assignment policy in software process”. In: Proceedings of the Ninth Asia-Pacific Software Engineering Conference – APSEC, 2002, 376p.

[Li03] Li, J. Q.; Fan, Y. S.; Zhou, M. C. “Timing Constraint Workflow Nets for Workflow Analysis”. IEEE Tran. on Systems, Man, and Cybernetics-Part A: Systems and Humans, Vol. 33-2, 2003, pp. 179-193.

[Li09] Li, J.; Mao, M. “A Case Study on Tailoring Software Process for Characteristics Based on RUP”. In: Computational Intelligence and Software Engineering – CiSE, 2009, pp. 1-5.

[Luc05] Lucas, F. J.; Toval, A. “A Precise Approach for the Analysis of the UML Models Consistency”. In: 1st International Workshop on Best Practices of UML, in 24th International Conference on Conceptual Modeling – ER, 2005, pp. 74-84.

[Luc09] Lucas, F. J.; Molina, F.; Toval, A. “A Systematic Review of UML Model Consistency Management”. Information and Software Technology, Vol. 51-12, Dezembro 2009, pp. 1631-1645.

[Mac00] Machado, L.F.C. “Modelo para Definição de Processos de Software na Estação TABA”, Dissertação de Mestrado, COPPE/UFRJ, 2000, 126p.

[Mar00] Marjanovic, O. “Dynamic Verification of Temporal Constraints in Production Workflows”. In: Proceedings of the 11th Australian Database Conference, 2000, pp. 74-81.

[Mir06] Mirbel, I.; Ralyté, J. “Situational Method Engineering: Combining Assembly-based and Roadmap-driven Approaches”. In: Requirements Engineering, 2006, pp. 58-78.

[Moo04] Moore, B.; Dean, D.; Gerber, A. Wagenknecht, G., & Vanderheyden, P., “Eclipse development Using the Graphical Editing Framework and the Eclipse Modeling Framework”. RedBooks, 2004, 250p.

[Mou01] Moura, A. V. “Especificações em Z: Uma Introdução”. Editora da UNICAMP, 2001, 306p.

[Mus05] Muskens, J.; Bril, R.; Chaudron, M. “Generalizing Consistency Checking between Software Views”. In: 5th Working IEEE/IFIP Conference on Software Architecture – WICSA’05, 2005, pp. 169–180.

[Nor09] Norris, N.; Letkeman, K. “Governing and managing enterprise models: Part 1. Introduction and concepts”. IBM Developer Works, Capturado em: http://www.ibm.com/developerworks/rational/library/09/0113_letkeman-norris/, Março 2011.

[Oat06] Oates, B. “Researching Information Systems and Computing”. Sage Publications Ltda, 2006, 341p.

[Oli07] Oliveira, C.; Falcão, M. “Prognóstico da hipoglicemia hiperinsulinêmica persistente da infância: uma revisão sistemática”. Revista. paul. pediatr., Vol. 25- 3, 2007, p.271-275.

[Oje09] Garcia-Ojeda, J. C.; DeLoach S. A.; Robby, B. “agentTool Process Editor: Supporting the Design of Tailored Agent-based Processes”. In: Symposium on Applied Computing – SAC’09, 2009, pp. 707-714.

[Omg02] OMG. “Software & Systems Process Engineering Metamodel Specification 1.1”. Capturado em: http://www.omg.org/cgi-bin/doc?formal/02-11-14, Março 2011.

[Omg03] OMG. “Unified modeling language specification v1.5”. Capturado em: http://www.omg.org/docs/formal/03-03-01.pdf, Agosto 2010.

[Omg06] OMG. “Meta Object Facility 2.0”. Capturado em: http://www.omg.org/spec/MOF/2.0/, Março 2011.

[Omg07a] OMG. “Software & Systems Process Engineering Metamodel Specification 2.0”. Capturado em: http://www.omg.org/spec/SPEM/2.0/, Março 2011.

[Omg07b] OMG. “Omg unified modeling language (omg uml), infrastructure, v2.1.2”. Capturado em: http://www.omg.org/docs/formal/07-11-04.pdf, Janeiro 2009.

[Omg08] OMG. “Mof 2.0 Facility And Object Lifecycle”. Capturado em: http://www.omg.org/spec/SPEM/2.0/, Março 2011.

[Ope10] Eclipse. “OpenUP”. Capturado em: http://epf.eclipse.org/wikis/openup//, Março 2011.

[Ost87] Osterweil, L. “Software processes are software too”. In: Proceedings of the 9th international conference on Software Engineering. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 2–13.

[Par11] Park, S.; Bae D. H.; “An Approach to Analysing the Software Process Change Impact Using Process Slicing and Simulation”. The Journal of Systems and Software, Vol. 84-4, Abril 2011, pp. 528-543.

[Ped07] Pedreira O.; Piatiini M.; Luaces M. R.; Brisaboa N. R. “A Systematic Review of Software Process Tailoring”. *ACM SIGSOFT Software Engineering Notes*, Vol. 32-3, Maio 2007, pp. 1-6.

[Per05a] Gonzalez-Perez, C.; Henderson-Sellers, B. “Templates and Resources in Software Development Methodologies”. Journal of Object Technology, Vol.4, 2005, pp. 173-190.

[Per05b] Gonzalez-Perez, C.; Henderson-Sellers, B. “A Comparison of Four Process Metamodels and the Creation of a New Generic Standard”. Information and Software Technology, Vol. 47-1, Janeiro 2005, pp. 49-65.

[Per07a] Gonzalez-Perez, C.; Henderson-Sellers, B. “Modelling Software Development Methodologies: A Conceptual Foundation”. The Journal of Systems and Software, Vol. 80-11, Novembro 2007, pp. 1778-1796.

[Per07b] Pereira, E. B.; Bastos, R. M.; Oliveira T. C. “A Systematic Approach to ProcessbTailoring”. In: International Conference on Systems Engineering and Modeling - ICSEM, 2007, pp. 71-78.

[Pli98] Plihon, V.; Ralyté, J.; Benjamen, A.; Maiden, N. A. M.; Sutcliffe, A.; Dubois, E.; Heymans, P. “A Reuse-Oriented Approach for the Construction of Scenario based Methods”. In: 5th International Conference on Software Process - ICSP, 1998. 14p.

[Puv09] Puviani, M.; Serugendo, G. D. M.; Frei, R; Cabri G. “Methodologies for Self-organising Systems: a SPEM Approach”. In: International Conference on Web Intelligence and Intelligent Agent Technology, 2009, pp. 66-99.

[Ral99] Ralyté, J. “Reusing Scenario based Approaches in Requirements Engineering Methods: CREWS Method Base”. In: Proceedings of 10th International Workshop on Database and Expert Systems Applications – DEXA, 1999, 5p.

[Ral01a] Ralyté, J.; Rolland, C. “An Assembly Process Model for Method Engineering”. In: Advanced Information Systems Engineering - CAISE, 2001, pp. 267-283.

[Ral01b] Ralyté, J.; Rolland, C. “An Approach for Method Engineering”. In: Proceedings of 20th International on Conceptual Modelling – ER, 2001, pp. 471-484.

[Ral03] Ralyté, J. Deneckere, R., Roll, C., “Towards a Generic Model for Situational Method Engineering”. In: Advanced Information Systems Engineering – CAISE, 2003, pp. 95-110.

[Ral04] Ralyté, J. “Towards Situational Methods for Information Systems Development: Engineering Reusable Method Chunks”. In: Proceedings of 13th International Conference on Information Systems Development, 2004, pp.271-282.

[Rib02] Ribó, J. M.; Franch, X. “A Precedence-based Approach for Proactive Control in Software Process Modelling”. In: International Conference on Software Engineering and Knowledge Engineering – SEKE, 2002, pp. 457-464.

[Roc01] Rocha, A. R. C.; Maldonado, J. C.; Weber, K. C. “Qualidade de Software: Teoria de Prática”. Prentice Hall, 2001, 303p.

[Rod10] Rodriguez, D.; Garcia, E.; Sanchez, S.; Nuzzi, C. R. S. “Defining Software Process Model Constraints with Rules Using OWL and SWRL”. International Journal of Software Engineering and Knowledge Engineering, Vol. 20-4, Abril 2010, pp. 533-548.

[Rol96] Rolland, C.; Prakash, N. “A Proposal for Context-Specific Method Engineering” Method Engineering. Principles of Method Construction and Tool Support”. In: Working Conference on Method Engineering, 1996, pp.191-208.

[Rol98] Rolland, C.; Plihon, V.; Ralyté, J. “Specifying the Reuse Context of Scenario Method Chunks”. In: Advanced Information Systems Engineering 10th International Conference – CAISE, 1998, pp. 91-218.

[Rui09] Rui, H.; Hao, W.; Zhiqing, L. “A Software Process Tailoring Approach Using a Unified Lifecycle Template”. In: Computational Intelligence and Software Engineering – CiSE’09, 2009, pp. 1-7.

[Sel03] Henderson-Sellers, B. “Method Engineering of OO Systems Development”. Communications of the ACM - Service-oriented computing, Vol. 46-10, Outubro 2003, pp. 73-78.

[Sel08] Henderson-Sellers, B.; Gonzalez-Perez, C.; Ralyté, J. “Comparison of Method Chunks and Method Fragments for Situational Method Engineering”. In: Australian Conference on Software Engineering – ASWEC’08, 2008, pp. 479-488.

[Ser04] Serour, M. K.; Henderson-Sellers, B. “Introducing Agility: A Case Study of Situational Method Engineering Using the OPEN Process Framework”. In: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC’04), 2004, pp. 50-57.

[Smu95] Smullyan, R., M. “First-Order Logic”. Dover Publications, Inc. New York, 1995, 157p.

[Xu05] Xu, P.; Ramesh, B. “Knowledge Support in Software Process Tailoring”. In: Proceedings of the 38th Hawaii International Conference on System Sciences (HICSS), 2005, pp. 87-95.

[Yoo01] Yoon, I.; Min, S.; Bae, D. “Tailoring and Verifying Software Process”. In: Institute of Electrical and Electronic Engineers - IEEE, 2001, pp.202-209.

[Wag03] Wagner, R.; Giese, H.; Nickel, U. “A plug-in for Fexible and Incremental Consistency Management”. In: Proceedings of the International Conference on the Unified Modeling Language (Workshop 7: Consistency Problems in UML-based Software Development), 2003, 8p.

[War03] Warmer, J.; kleppe, A. “The Object Constraint Language Second Edition – Getting Your Models Ready for MDA”, 2 edição, 2003, 203p.

[Was06] Washizaki, H. “Deriving Project-Specific Processes from Process Line Architecture with Commonality and Variability”, In: IEEE International Conference on Industrial Informatics, 2006, pp. 1301-1306.

[Wel95] Welzel, D.; Hausen, H. L.; Schmidt W. “Tailoring and Conformance Testing of Software Processes”. In: Institute of Electrical and Electronic Engineers - IEEE, 1995, pp. 41-49.

[Wes96] West, D. B. “Introduction to Graph Theory”. Prentice Hall,1996, 166p.

[Wes05] Westerheim, H.; Hanssen, G. K. “The Introduction and Use of a Tailoired Unified Process – A Case Study”. In: Conference on Software Engineering and Advanced Applications – EUROMICRO-SEAA, 2005, pp. 196-213.

[Zha98] Zahran, S. “Software Process Improvement”. Addison Wesley Longman Inc., 1998, 480p.

[Zho05] Zhong, J.; Song, B. “Verification of Resource Constraints for Concurrent Workflows”. In: Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing – SYNASC, 2005, 8p.

APÊNDICE A – PROTÓCOLO DA REVISÃO SISTEMÁTICA

1. Questão-foco

O objetivo geral desta revisão é encontrar pesquisas interessadas em propor soluções relacionadas com a consistência de um processo de desenvolvimento de software durante suas atividades de definição e adaptação.

1.1 Qualidade e amplitude da questão

Esta seção objetiva descrever a sintaxe da questão de pesquisa – através dos itens problema e questão - e a sua semântica – através dos itens intervenção, controle, efeito, medição dos resultados, população, aplicação e projeto experimental.

- Problema

Processos de desenvolvimento de software são procedimentos que geralmente convertem especificações informais, tipicamente coletadas em cenários reais, em partes de código formais que irão compor um produto de software. Neste sentido, definir um processo de desenvolvimento de software consistente que atenda as necessidades específicas de um projeto de software torna-se essencial. Como o objetivo desta pesquisa é propor uma solução que garanta a consistência de um processo nas suas atividades de definição e adaptação esta revisão sistemática foi executada para verificar o estado da arte deste campo de pesquisa, explicitando quais trabalhos estão relacionados com esta pesquisa e verificando lacunas ainda existentes.

- Questão de Pesquisa

Quais são as abordagens que apresentam soluções relacionadas com o aspecto de consistência para verificar um processo de desenvolvimento de software nas suas atividades de definição e/ou adaptação?

- Palavras-Chave e Sinônimo

Tabela A.1. Palavras-chaves e sinônimos.

Palavra-chave	Tradução em português	Sinônimos em inglês
Software Development Process	Processo de Desenvolvimento de	*Method* *Software Development Methodology*

	Software	
Process Definition	Definição de Processo	*Process Specification*
Process Modeling	Modelagem de Processo	-
Process Tailoring	Adaptação de Processos	*Process Configuration* *Situational Method Engineering*
Well-Formedness Rule	Regra de boa-formação	*Rule* *Constraint* *Restriction*
Process Verification	Verificação de Processo	*Process Checking* *Process Validation*
Consistency	Consistência	*Consistence*

- Intervenção: Identificação e avaliação das soluções expostas nas pesquisas.

- Controle: inexistente.

- Efeito: definir o estado da arte sobre aspectos relacionados com a consistência dos processos de desenvolvimento de software.

- Medição de resultados: número de trabalhos identificados considerando os critérios de inclusão e exclusão de trabalhos.

- População: artigos publicados em periódicos e anais de conferências relacionados ao tema de pesquisa, cuja data de publicação seja posterior a 2005.

- Aplicação: pesquisadores da área e engenheiros de processo.

- Projeto experimental: não se aplica.

2. Seleção das Fontes

Definição dos Critérios de Seleção das Fontes

Disponibilidade de consulta a artigos completos (*full papers*) através da Web pelo convênio PUCRS-CAPES; existência de mecanismos de busca com suporte a inserção de operadores booleanos (como *and* e *or*); e base de dados atualizada (com publicações dos últimos 5 anos).

Idiomas: inglês

Identificação das Fontes:

- **Método de pesquisa das fontes:** através dos mecanismos de busca próprios do IEEE Xplore, ACM Digital Library, SpringerLink, Scopus e Science@Direct.

- **String de busca:** *(“software process” **OR** method*) **AND** (“process definition” **OR** “process modeling” or “process modelling” **OR** “process tailoring” or “situational method engineering”) **AND** (consistenc*)*

- **Lista de fontes:** vide Tabela 6 do Capítulo 3.

Seleção das fontes após avaliação: todas as fontes foram aprovadas pela doutoranda e pelo seu orientador.

Verificação de referências: todas as fontes selecionadas foram aprovadas.

2 Seleção dos estudos

Definição dos estudos:

- **Critérios de inclusão e exclusão de estudos:** foram desconsiderados os estudos:

  C1. Que não estavam no formato de artigo completo (*full paper*).

  C2. Cujo conteúdo não estava relacionado a área de definição e adaptação de processos de desenvolvimento de software.

  C3. Trabalhos que citem importância da definição e adaptação de processos de software consistentes mas que não possuam nenhum tipo de solução para isto.

- **Definição do tipo de estudo:** foram removidos estudos do tipo *roadmaps* (trabalhos que indicam os desafios e as direções a serem tomadas em uma área de pesquisa).

- **Procedimentos para a seleção de estudos:** execução da *string* de busca nos mecanismos de busca oferecidos em cada uma das fontes selecionadas. A análise da seleção dos estudos foi realizada em 3 passos: (1) leitura de títulos e resumos (*abstract*) dos artigos retornados; (2) leitura dos artigos completos; e (3) seleção dos artigos que traziam soluções relacionadas com a consistência dos processos de software. Nessa última etapa alguns artigos que ficaram de fora desse universo foram utilizados como referencial teórico por apresentarem importantes conceituações.

Execução da seleção:

- **Seleção inicial de estudos:** a busca em todos os mecanismos resultou em 212 artigos.

- **Avaliação da qualidade dos estudos:** 10 estudos (ou 4,7% da amostra inicial) foram selecionados para a extração de informações.

- **Revisão da seleção:** a seleção de estudos foi aprovada.

3 Extração de informações

Critérios de inclusão e exclusão de informações: conforme descrito na Seção 3.3 do Capítulo 3.

Formulários para extração dos dados: os 8 diferentes aspectos identificados estão descritos na Seção 3.3 do Capítulo 3.

Execução da extração: vide Tabelas 7 e 8 do Capítulo 3.

Resolução de divergências entre os pesquisadores: não houve divergências.

4 Sumarização dos resultados

Cálculos estatísticos sobre os resultados: não se aplica.

Apresentação dos resultados em tabelas: vide Tabelas 7 e 8 do Capítulo 3.

Análise de sensibilidade: não se aplica.

Plotagem: não se aplica.

Comentários finais:

- **Número de estudos:** foram retornados 212 artigos dos quais 10 foram selecionados. Além disso, outros 3 artigos foram incluídos para extração de informações.
- **Vieses identificados:** o número de fontes de busca utilizadas (cinco); a qualidade dos motores de busca das fontes selecionadas; e a influência da autora na seleção dos artigos e na extração das informações.
- **Variação entre revisores**: não se aplica.
- **Aplicação dos resultados:** os resultados servirão de base para conclusão da tese da autora.
- **Recomendações:** nenhuma.

APÊNDICE B – REGRAS DE BOA-FORMAÇÃO REDEFINIDAS PARA O PACOTE *PROCESS WITH METHODS*

Regra #13 – Um processo de software deve possuir exatamente um nodo inicial.

Semântica: Apenas uma instância da metaclasse *Activity* ou da metaclasse *TaskUse* com o valor do atributo *specialNode* igual a "*start*" deverá ser instanciada em um processo de software.

Regra #14 – O nodo inicial não pode ser sucessor para nenhuma atividade ou tarefa no sequenciamento definido em um processo de software.

Semântica: Uma instância da metaclasse *Activity* ou da metaclasse *TaskUse* que possui o valor do atributo *specialNode* igual a "*start*" não pode estar associada no atributo *successor* de uma instância da metaclasse *WorkSequence.*

Regra # 15 – O nodo final não pode ser predecessor para nenhuma atividade ou tarefa no sequenciamento definido em um processo de software.

Semântica: Uma instância da metaclasse *Activity* ou da metaclasse *TaskUse* que possui o valor do atributo *specialNode* igual a "*end*" não pode estar associada no atributo *predecessor* de uma instância da metaclasse *WorkSequence.*

Regra #16 – O sequenciamento de atividades ou tarefas não deve possuir situações que representem um deadlock na execução de um processo de software.

Semântica: Após a divisão das atividades em *atividade_início* e *atividade_fim* ou das tarefas em *tarefa_início* e *tarefa_fim* e a simplificação das informações sobre as transições não devem existir em nenhum ponto do sequenciamento resultante ciclos e laços.

Regra #17 – Não pode existir transições duplicadas entre duas atividades ou tarefas em um sequenciamento.

Semântica: Não pode existir duas ou mais instâncias da metaclasse *WorkSequence* que possuam o mesmo valor para o atributo *linkKind* e que possuam a mesma instância da metaclasse *Activity* ou da metaclasse *TaskUse* selecionada para o atributo *predecessor,* bem como a mesma instância da metaclasse *Activity* ou da metaclasse *TaskUse* selecionada para o atributo *successor*.

Regra #18 – Todas as atividades ou tarefas que são sequenciadas em um processo de software deverão ter ligação com o nodo inicial e final do processo para garantir que toda atividade ou tarefa é iniciada após o nodo inicial e possui seu término antes do nodo final .

Semântica: Após a divisão das atividades em *atividade_início* e *atividade_fim* ou das tarefas em *tarefa_início* e *tarefa_fim* e a simplificação das informações sobre as transições toda *atividade_início* ou *tarefa_início* deve possuir pelo menos uma aresta chegando nessa atividade e toda *atividade_fim* ou *tarefa_fim* deve possuir pelo menos uma aresta partindo para outra atividade.

Regra # 41 – As atividades ou tarefas do processo que representam os nodos inicial e final não devem ser associados com instâncias da metaclasse ProcessPerformer.

Semântica: Toda instância da metaclasse *Activity* ou da metaclasse *TaskUse* que possui o valor do atributo *specialNode* igual a "*start*" ou "*end*" não pode estar associada no atributo *linkedTaskUse* de nenhuma instância da metaclasse *ProcessPerformer*.

Regra #42 – As atividades ou tarefas do processo que representam os nodos inicial e final não devem possuir elementos.

Semântica: Nenhuma instância da metaclasse *Activity* ou da metaclasse *TaskUse* que possui o valor do atributo *specialNode* igual a "*start*" ou "*end*" não pode possuir outras instâncias através de um relacionamento de composição com metaclasses que mantém relacionamento de herança com a metaclasse *BreakdownElement*.

Regra # 43 – O nodo inicial deve ser definido como predecessor para pelo menos uma atividade ou tarefa no sequenciamento definido em um processo de software.

Semântica: Toda instância da metaclasse *Activity* ou da metaclasse *TaskUse* que possui o valor do atributo *specialNode* igual a "*start*" deve estar associada no atributo *predecessor* de pelo menos uma instância da metaclasse *WorkSequence.*

Regra # 44 – O nodo final deve ser definido como sucessor para pelo menos uma atividade ou tarefa no sequenciamento definido em um processo de software.

Semântica: Toda instância da metaclasse *Activity* ou da metaclasse *TaskUse* que possui o valor do atributo *specialNode* igual a "*end*" deve estar associada no atributo *successor* de pelo menos uma instância da metaclasse *WorkSequence.*

Regra # 45 – O sequenciamento estabelecido entre as atividades ou tarefas e o nodo inicial ou final em um processo de software deve ser realizado através do tipo de sequenciamento finishToStart.

Semântica: Toda instância da metaclasse *Activity* ou da metaclasse *TaskUse* que possui o valor do atributo *specialNode* igual a “*start*” ou “*end*” deve ser associado somente com instâncias da metaclasse *WorkSequence* que possuem valor do atributo *linkKind* igual a “*finishToStart*”.

APÊNDICE C – ANÁLISE DE IMPACTO PARA OS MECANIMOS *USEDACTIVITY*, *SUPRESSEDBREAKDOWNELEMENT* E *VARIABILTIY*

Tabela C.1 – Análise de impacto para a exclusão de uma instância do elemento *ExternalUse*

Elemento do Processo	**Impacto**
Metaclasse *Activity*	- *A instância de ExternalUse* excluída deve ter suas associações com as instâncias de *Activity* eliminadas.
Metaclasse *TaskUse*	**-**
Metaclasse *RoleUse*	**-**
Metaclasse *InternalUse*	Se a instância de *ExternalUse* excluída está associada ao atributo *target* de uma ou mais instâncias de *WorkProductUseRelationship* com valor de atributo *relationType* igual a *dependency,* então a instância de *InternalUse* associada no atributo *source* da(s) instância(s) *WorkProductUseRelationship* deve(m) ser excluída(s). Se a instância de *ExternalUse* excluída for o único elemento associado no atributo *target* de uma instância de *WorkProductUseRelationship* com valor de atributo *relationType* igual a *composition*, então a instância de *InternalUse* associada no atributo *source* da(s) instância(s) *WorkProductUseRelationship* deve(m) ser excluída(s). Se a instância de *ExternalUse* excluída está associada ao atributo *source* de uma ou mais instâncias de *WorkProductUseRelationship* com valor de atributo *relationType* igual a *composition,então todas as instâncias de InternalUse associadas no atributo target da(s) instância(s) WorkProductUseRelationship deve(m) ser excluída(s).*
Metaclasse *ExternalUse*	Se a instância de *ExternalUse* excluída está associadoa ao atributo *target* de uma ou mais instâncias de *WorkProductUseRelationship* com valor de atributo *relationType* igual a *dependency,* então a instância de *ExternalUse* associada no atributo *source* da(s) instância(s) *WorkProductUseRelationship* deve(m) ser excluída(s). Se a instância de *ExternalUse* excluída for o único elemento associado no atributo *target* de uma de *WorkProductUseRelationship* com valor de atributo *relationType* igual a *composition*, então a instância de *ExternalUse* associada no atributo *source* da(s) instância(s) *WorkProductUseRelationship* deve(m) ser excluída(s). Se a instância de *ExternalUse* excluída está associada ao atributo *source* de uma ou mais instâncias de *WorkProductUseRelationship* com valor de atributo *relationType* igual a composition,*então todas as instâncias de ExternalUse associadas no atributo target da(s) instância(s) WorkProductUseRelationship deve(m) ser excluída(s).*

Metaclasse *ProcessParameter*	- Todas as instâncias de *ProcessParameter* que estão associadas com a instância de *ExternalUse* excluída devem também ser excluídas.
Metaclasse *ProcessPerformer*	**-**
Metaclasse *WorkSequence*	**-**
Metaclasse *WorkProductRelationshipUse*	- A instância de *ExternalUse* excluída deve ter suas associações com as instâncias de *WorkProductRelationshipUse* eliminadas. Se alguma das instâncias de *WorkProductRelationshipUse* afetadas não possuir mais instâncias de *InternalUse* e/ou *ExternalUse* associadas aos seus atributos *source* e *target,* então estas instâncias deverão também ser excluídas.
Metaclasse *ProcessResponsabilityAssingment*	- Todas as instâncias de *ProcessResponsabilityAssingment* que estão associadas com a instância de *ExternalUse* excluída devem também ser excluídas.

Tabela C.2 – Análise de impacto para a exclusão de uma instância do elemento *Activity*

Elemento do Processo	**Impacto**
Metaclasse *Activity*	- Todas as instâncias de *Activity* que pertencem a instância da *Activity* excluída devem ser também excluídas.
Metaclasse *TaskUse*	- Todas as instâncias de *TaskUse* que pertencem a instância da *Activity* excluída devem ser também excluídas.
Metaclasse *RoleUse*	- Todas as instâncias de *RoleUse* que pertencem a instância da *Activity* excluída devem ser também excluídas.
Metaclasse *InternalUse*	- Todas as instâncias de *InternalUse* que pertencem a instância da *Activity* excluída devem ser também excluídas.
Metaclasse *ExternalUse*	- Todas as instâncias de *ExternalUse* que pertencem a instância da *Activity* excluída devem ser também excluídas.
Metaclasse *ProcessParameter*	- Todas as instâncias de *ProcessParameter* que pertencem a instância da *Activity* excluída devem ser também excluídas.
Meetaclasse *ProcessPerformer*	- Todas as instâncias de *ProcessPerformer* que pertencem a instância da *Activity* excluída devem ser também excluídas.
Metaclasse *WorkSequence*	- Todas as instâncias de *WorkSequence* que estão associadas com a instância da *Activity* excluída devem também ser excluídas. - Se necessário, o sequenciamento de atividades deverá ser reorganizado com a criação de instâncias de *WorkSequence*. **
Metaclasse *WorkProductRelationshipUse*	- Todas as instâncias de *WorkProductRelationshipUse* que pertencem a instância da *Activity* excluída devem ser também excluídas.

Metaclasse *ProcessResponsabilityAssingment*	- Todas as instâncias de *ProcessResponsabilityAssingment* que pertencem a instância da Activity excluída devem ser também excluídas.

Tabela C.3 – Análise de impacto para a exclusão de uma instância do elemento *TaskUse*

Elemento do Processo	**Impacto**
Metaclasse *Activity*	- *A instância de TaskUse* excluída deve ter suas associações com as instâncias de *Activity* eliminadas. Se alguma das instâncias de *Activity* afetadas não possuir mais instâncias de *TaskUse* elas também devem ser excluídas.
Metaclasse *TaskUse*	**-**
Metaclasse *RoleUse*	**-**
Metaclasse *InternalUse*	**-**
Metaclasse *ExternalUse*	**-**
Metaclasse *ProcessParameter*	- Todas as instâncias de *ProcessParameter* que pertencem a instância da *TaskUse* excluída devem ser também excluídas. Se alguma das instâncias de *ProcessParameter* excluídos estiverem definidos nas atividades superiores da instância da *TaskUse* excluída eles também devem ser excluídos.
Meetaclasse *ProcessPerformer*	- Todas as instâncias de *ProcessPerformer* que estão associadas com a instância da *TaskUse* excluída devem também ser excluídas.
Metaclasse *WorkSequence*	- Todas as instâncias de *WorkSequence* que estão associadas com a instância da *TaskUse* excluída devem também ser excluídas. - Se necessário, o sequenciamento de tarefas deverá ser reorganizado com a criação de instâncias de *WorkSequence*. **
Metaclasse *WorkProductRelationshipUse*	**-**
Metaclasse *ProcessResponsabilityAssingment*	**-**

Tabela C.4 – Análise de impacto para a exclusão de uma instância do elemento *RoleUse*

Elemento do Processo	**Impacto**
Metaclasse *Activity*	- *A instância de RoleUse* excluída deve ter suas associações com as instâncias de *Activity* eliminadas.

Metaclasse *TaskUse*	**-**
Metaclasse *RoleUse*	**-**
Metaclasse *InternalUse*	**-**
Metaclasse *ExternalUse*	**-**
Metaclasse *ProcessParameter*	**-**
Metaclasse *ProcessPerformer*	- *A instância de RoleUse* excluída deve ter suas associações com as instâncias de *ProcessPerformer* eliminadas. Se alguma das instâncias de *ProcessPerformer* afetadas não possuir mais instâncias de *RoleUse* associadas aos seus atributos *linkedRoleUse,* então estas instâncias deverão também ser excluídas.
Metaclasse *WorkSequence*	**-**
Metaclasse *WorkProductRelationshipUse*	**-**
Metaclasse *ProcessResponsabilityAssingment*	- A instância de *RoleUse* excluída deve ter suas associações com as instâncias de *ProcessResponsabilityAssingment* eliminadas. Se alguma das instâncias de *ProcessResponsabilityAssingment* afetadas não possuir mais instâncias de *RoleUse* associadas aos seus atributos *linkedRoleUse*, então estas instâncias deverão também ser excluídas.

Tabela C.5 – Análise de impacto para a exclusão de uma instância do elemento *ProcessPerformer*

Elemento do Processo	**Impacto**
Metaclasse *Activity*	- A instância de *ProcessPerformer* excluída deve ter suas associações com as instâncias de *Activity* eliminadas.
Metaclasse *TaskUse*	- A instância de *ProcessPerformer* excluída deve ter suas associações com a metaclasse *TaskUse* eliminadas. Se alguma das instâncias de *TaskUse* afetadas não possuir mais associação com nenhuma outra instância de *ProcessPerformer*, então estas instâncias deverão também ser excluídas.
Metaclasse *RoleUse*	- A instância de *ProcessPerformer* excluída deve ter suas associações com as instâncias de *RoleUse* eliminadas. Se alguma das instâncias de *RoleUse* afetadas não possuir mais associação com nenhuma outra instância de *ProcessPerformer*, então estas instâncias deverão também ser excluídas.
Metaclasse *InternalUse*	**-**
Metaclasse *ExternalUse*	**-**

Metaclasse *ProcessParameter*	**-**
Metaclasse *ProcessPerformer*	**-**
Metaclasse *WorkSequence*	**-**
Metaclasse *WorkProductRelationshipUse*	**-**
Metaclasse *ProcessResponsabilityAssingment*	**-**

Tabela C.6 – Análise de impacto para a exclusão de uma instância do elemento *ProcessParameter*

Elemento do Processo	**Impacto**
Metaclasse *Activity*	- A instância de *ProcessParameter* excluída deve ter suas associações com as instâncias de *Activity* eliminadas.
Metaclasse *TaskUse*	- Se a instância de *ProcessParameter* excluída possuir o valor do atributo *optionality* igual a *false* então a instância da *TaskUse* a qual o *ProcessParameter* pertence também deve ser excluída. - Se a instância de *ProcessParameter* excluída possuir o valor do atributo *optionality* igual a *true* então sua associação com a instância de *TaskUse* deverá ser eliminada. Se a instância de *TaskUse* afetada não possuir nenhuma outra instância de *ProcessParameter* com o valor do atributo *direction* igual a *out* ou *inout,* então esta instância também deverá ser excluída.
Metaclasse *RoleUse*	**-**
Metaclasse *InternalUse*	- A instância de *ProcessParameter* excluída deve ter suas associações com as instâncias de *InternalUse* eliminadas. Se uma das instâncias de *InternalUse* afetadas não possuir associação com outra instância de *ProcessParameter* onde o valor do atributo *direction* é igual a *out*, então estas instâncias devem ser excluídas. - Se a instância de *ProcessParameter* excluída possui o valor do atributo *direction* igual a *out* e a instância de *InternalUse* associada no seu atributo *ParameterType* permanece no processo (por possuir associação com outra(s) instância(s) de *ProcessParameter* onde o valor do atributo *direction* é igual a *out)* então deverá ser considerado o sequenciamento deste processo para que seja feita uma nova análise de impacto. Esta análise de impacto deverá ser realizada nas tarefas que precedem todas as instâncias de *ProcessParameter* onde o valor do atributo *direction* é igual a *out* e a instância de *InternalUse* associada no atributo *parameterType* destes *ProcessParameters* são iguais à instância do *InternalUse* associada ao *ProcessParameter* sendo excluído.

Metaclasse *ExternalUse*	- A instância de *ProcessParameter* excluída deve ter suas associações com as instâncias de *ExternalUse* eliminadas. Se uma das instâncias de *ExternalUse* afetadas não possuir associação com outra instância de *ProcessParameter* onde o valor do atributo *direction* é igual a *in* ou *inout*, então estas instâncias devem ser excluídas.
Metaclasse *ProcessParameter*	**-**
Metaclasse *ProcessPerformer*	**-**
Metaclasse *WorkSequence*	**-**
Metaclasse *WorkProductRelationshipUse*	**-**
Metaclasse *ProcessResponsabilityAssingment*	**-**

Tabela C.7 – Análise de impacto para a exclusão de uma instância do elemento *WorkProductUseRelationship*

Elemento do Processo	**Impacto**
Metaclasse *Activity*	- A instância de *WorkProductUseRelationship* excluída deve ter suas associações com as instâncias de *Activity* eliminadas.
Metaclasse *TaskUse*	**-**
Metaclasse *RoleUse*	**-**
Metaclasse *InternalUse*	**-**
Metaclasse *ExternalUse*	**-**
Metaclasse *ProcessParameter*	**-**
Metaclasse *ProcessPerformer*	**-**
Metaclasse *WorkSequence*	**-**
Metaclasse *WorkProductRelationshipUse*	**-**
Metaclasse *ProcessResponsabilityAssingment*	**-**

Tabela C.8 – Análise de impacto para a exclusão de uma instância do elemento *ProcessResponsabilityAssignment*

Elemento do Processo	**Impacto**

Metaclasse *Activity*	- A instância de *ProcessResponsabilityAssingment* excluída deve ter suas associações com as instâncias de *Activity* eliminadas.
Metaclasse *TaskUse*	-
Metaclasse *RoleUse*	- A instância de *ProcessResponsabilityAssingment* excluída deve ter suas associações com as instâncias de *RoleUse* eliminadas.
Metaclasse *InternalUse*	- A instância de *ProcessResponsabilityAssingment* excluída deve ter suas associações com as instâncias de *InternalUse* eliminadas. Se alguma das instâncias de *InternalUse* afetadas não possuir mais associação com nenhuma outra instância de *ProcessResponsabilityAssingment*, então estas instâncias deverão também ser excluídas.
Metaclasse *ExternalUse*	- A instância de *ProcessResponsabilityAssingment* excluída deve ter suas associações com as instâncias de *ExternalUse* eliminadas.
Metaclasse *ProcessParameter*	-
Metaclasse *ProcessPerformer*	-
Metaclasse *WorkSequence*	-
Metaclasse *WorkProductRelationshipUse*	-
Metaclasse *ProcessResponsabilityAssingment*	-

** Reorganizar o sequenciamento de atividades ou tarefas envolve a criação de novas instâncias de *WorkSequence* que conectem as atividades ou tarefas precedentes com as atividades ou tarefas sucessoras daquelas que foram excluídas. Para facilitar o entendimento de como isto é feito nesta pesquisa, um exemplo é mostrado na Figura C.1. O exemplo mostra um conjunto de atividades sequenciadas, os quais se utilizam do sequenciamento proposto nesta pesquisa e a proposição de exclusão de alguns destes elementos destacados com o ícone (✖).

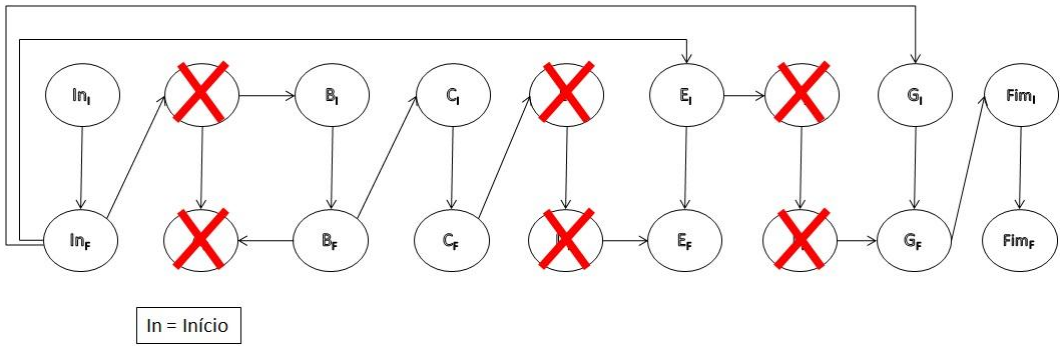

Figura C.1 – Exemplo de sequenciamento de atividades com proposta de exclusões

Realizando a análise de impacto no exemplo acima, contudo considerando apenas as informações de sequenciamento de atividades e tarefas tem-se o resultado que é exibido na Figura C.2.

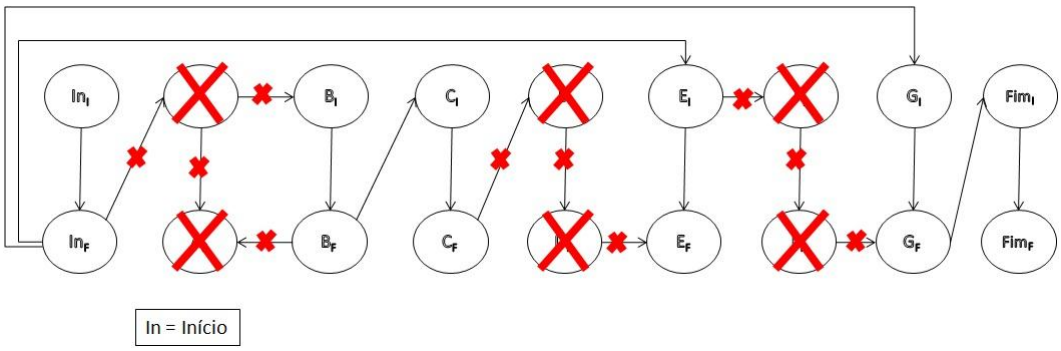

Figura C.2 – Análise de impacto para informações de sequenciamento

Nota-se na Figura C.2 que a exclusão dos elementos do exemplo em questão apontam para a exclusão de vários sequenciamentos, os quais em um processo de software são representados por instâncias de *WorkSequence*. Tais exclusões causarão inconsistências no sequenciamento do exemplo proposto, uma vez que a *Atividade B* não possuirá mais conexão com o nodo inicial e as *Atividades C* e *E* não possuirão mais conexão com o nodo final. Para resolver este tipo de problema, nesta pesquisa, é proposto utilizar as informações de transitividade das relações. Especificamente para resolver a situação da *Atividade B*, por exemplo, verifica-se na Figura C.1 que por transitividade a *Atividade B* está conectada com o nodo inicial. Desta forma, após a exclusão da *Atividade A* deve-se criar uma instância de *WorkSequence* que conecte o nodo inicial como predecessor da *Atividade B*. Esta *WorkSequence* deverá possuir o valor do atributo *linkKind* igual a *finishToStart*. Utilizando-se também de informações de transitividade, para resolver as situações das *Atividades C* e *E,* observa-se na Figura C.1 que devem ser criadas duas novas instâncias de *WorkSequence*. A primeira delas deverá conectar a *Atividade C* como predecessora da *Atividade E* com o valor do atributo *linkKind*

igual a *finishToFinish.* Já a segunda instância de *WorkSequence* deverá conectar a *Atividade E* como predecessora da *Atividade G* com o valor do atributo linkKind *startToFinish*. O conjunto de atividades e o novo sequenciamento definido após as exclusões acima, é mostrado na Figura C.3.

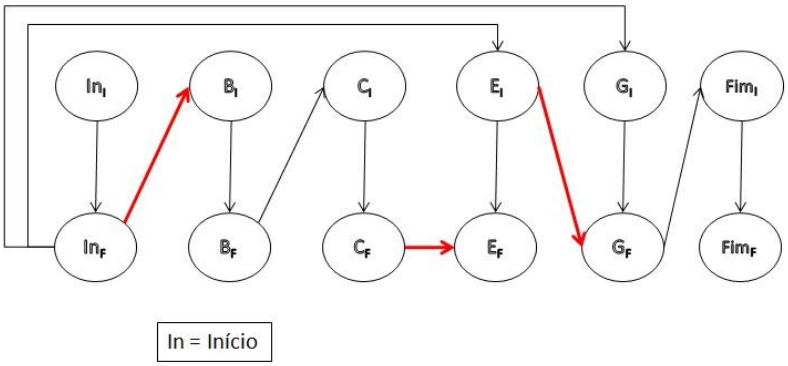

Figura C.3 – Reorganização do sequenciamento após as exclusões de atividades

APÊNDICE D – FORMALIZAÇÃO DAS REGRAS DE BOA-FORMAÇÃO REDEFINIDAS PARA O PACOTE *PROCESS WITH METHODS*

Este Apêndice redefine algumas regras do pacote *Process with Methods* e define vários novos precicados. Todos estes novos predicados são relacionados com aspectos de sequenciamento em um processo de software e necessitaram ser criados, uma vez que o elemento tarefa (que é definido no pacote *Process with Methods*) também pode assumir relações de precedência em um processo de software.

Inicialmente para capturar o conceito de tarefa predecessora e sucessora criou-se os predicados *pre-tarefa* (t_1, t_2) e *pos-tarefa* (t_2, t_1) indicando, respectivamente, que t_1 é uma tarefa predecessora de t_2 ou, de forma inversa, que t_2 é uma tarefa sucessora de t_1. Em ambos os predicados, t_1 e t_2 são instâncias de *TaskUse*. Considerando que as relações de precedência pré e pós-tarefas são transitivas e assimétricas, os seguintes axiomas foram definidos:

$\forall(t_1, t_2)$ (pre-tarefa(t_1, t_2) $\leftrightarrow$ pos-tarefa(a_2, a_1)) **(A15)**

$\forall(t_1, t_2, t_3)$ (pre-tarefa(t_1, t_2) $\wedge$ pre-tarefa(t_2, t_3) $\rightarrow$ pre-tarefa(t_1, t_3)) **(A16)**

$\forall(t_1, t_2)$ (pre-tarefa(t_1, t_2) $\rightarrow \neg$ pre-tarefa(t_2, t_1)) **(A17)**

$\forall t_1 \neg$ pre-tarefa(t_1, t_1) **(A18)**

Para formalizar os conceitos de tarefa_inicio e tarefa_fim, os seguintes predicados foram definidos: *tarefa-inicio*(t, t_i) e *tarefa-fim*(t, t_f) onde t representa uma instância de *TaskUse* e t_i e t_f identificam, respectivamente, o início e o fim desta *TaskUse*. Para registrar que toda tarefa realmente possui um início e um fim e que o início precede o fim considere os seguintes axiomas:

$\forall x$(taskUse(x) $\rightarrow \exists!(x1,x2)$ (taskUse(x1) $\wedge$ tarefa-inicio(x,x1) $\wedge$ taskUse (x2) $\wedge$ tarefa-fim(x,x2)) **(A19)**

$\forall x \exists!(x1,x2)$ (taskUse (x) $\rightarrow$ tarefa-inicio(x, x1) $\wedge$ tarefa-fim(x, x2) $\wedge$ pre-tarefa (x1, x2)) **(A20)**

Utilizando os novos conceitos de tarefa-início e tarefa-fim, traduziu-se os valores do atributo *linkKind* através dos seguintes predicados:

$\forall x, x1, x2$ (workSequence(x) $\wedge$ taskUse(x1) $\wedge$ taskUse(x2) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2) $\wedge$ linkKind(x, FS) $\rightarrow \exists!(a_1,a_2)$ (tarefa-fim(x1, a_1) $\wedge$ tarefa-inicio(x2, a_2) $\wedge$ pre- tarefa(a_1,a_2)))

$\forall x, x1, x2$ (workSequence(x) $\wedge$ taskUse(x1) $\wedge$ taskUse(x2) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2) $\wedge$ linkKind(x, FF) $\rightarrow \exists!(a_1,a_2)$ (tarefa-fim(x1, a_1) $\wedge$ tarefa-fim(x2, a_2) $\wedge$ pre-tarefa (a_1,a_2)))

$\forall x, x1, x2$ (workSequence(x) $\wedge$ taskUse(x1) $\wedge$ taskUse(x2) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2) $\wedge$ linkKind(x, SS) $\rightarrow \exists!(a_1,a_2)$ (tarefa-inicio(x1, a_1) $\wedge$ tarefa-inicio(x2, a_2) $\wedge$ pre- tarefa(a_1,a_2)))

$\forall x, x1, x2$ (workSequence(x) $\wedge$ taskUse(x1) $\wedge$ taskUse(x2) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2) $\wedge$ linkkind(x, SF) $\rightarrow \exists!(a_1,a_2)$ (tarefa-inicio(x1, a_1) $\wedge$ tarefa-fim(x2, a_2) $\wedge$ pre- tarefa(a_1,a_2)))

Os predicados acima estabelecem precedência considerando os inícios e fins de duas tarefas. Para registrar que uma tarefa precede outra tarefa sem precisar considerar seus inícios e fins, definiu-se os seguintes predicados:

$\forall x, y, x1, x2$ (taskUse(x) $\wedge$ taskUse(y) $\wedge$ tarefa-inicio(x, x1) $\wedge$ tarefa-inicio(y, x2) $\wedge$ pre_ tarefa(x_1, x_2) $\rightarrow$ pre-tarefa(x, y))

$\forall x, y, x1, x2$ (taskUse(x) $\wedge$ taskUse(y) $\wedge$ tarefa-inicio(x, x1) $\wedge$ tarefa-fim(y, x2) $\wedge$ pre- tarefa(x_1, x_2) $\rightarrow$ pre-tarefa(x, y))

$\forall x, y, x1, x2$ (taskUse(x) $\wedge$ taskUse(y) $\wedge$ tarefa-fim(x, x1) $\wedge$ tarefa-inicio(y, x2) $\wedge$ pre- tarefa(x_1, x_2) $\rightarrow$ pre-tarefa(x, y))

$\forall x, y, x1, x2$ (taskUse(x) $\wedge$ taskUse(y) $\wedge$ tarefa-fim(x, x1) $\wedge$ tarefa-fim(y, x2) $\wedge$ pre- tarefa(x_1, x_2) $\rightarrow$ pre- tarefa(x, y))

Baseado nos novos predicados, as seguintes regras foram redefinidas para o pacote *Process with Methods*:

Regra #6 – Em um sequenciamento definido para um processo de software os nodos inicial e final devem ser do mesmo tipo.

$\neg \exists$ (((taskUse(x) $\wedge$ specialNodel(x,'end')) $\vee$ (taskUse(x) $\wedge$ specialNodel(x,'start'))) $\wedge$ ((activity(x) $\wedge$ specialNodel(x,'end')) $\vee$ (activity(x) $\wedge$ specialNodel(x,'start'))))

Regra #12 – Um processo de software deve possuir exatamente um nodo final.

$\exists!x$ ((activity(x) $\wedge$ specialNodel(x,'end')) $\vee$ (taskUse(x) $\wedge$ specialNodel(x,'end')))

Regra #13 – Um processo de software deve possuir exatamente um nodo inicial.

$\exists!x$ ((activity(x) $\wedge$ specialNodel(x,'start')) $\vee$ (taskUse(x) $\wedge$ specialNodel(x,'start')))

Regra # 14 – O nodo inicial não deve ser sucessor para nenhuma atividade ou tarefa no sequenciamento definido em um processo de software.

$\forall x$ ((activity(x) $\wedge$ specialNodel(x,'start')) $\vee$ (taskUse(x) $\wedge$ specialNodel(x,'start')) $\rightarrow \neg \exists y$ (worksequence(y) $\wedge$ sucessor(y,x)))

Regra # 15 – O nodo final não deve ser predecessor para nenhuma atividade ou tarefa no sequenciamento definido em um processo de software.

$\forall x$ ((activity(x) $\wedge$ specialNodel(x,'end')) $\vee$ (taskUse(x) $\wedge$ specialNodel(x,'end')) $\rightarrow \neg \exists y$ (worksequence(y) $\wedge$ sucessor(y,x)))

Regra #16 – O sequenciamento de atividades ou tarefas não deve possuir situações que representem um deadlock na execução de um processo de software.

$\forall$(x1, x2) (activity(x1) $\wedge$ activity(x2) $\wedge$ pre-atividade(x1, x2) $\rightarrow \neg$ pre-atividade(x2, x1))

$\forall$(x1, x2) (taskUse(x1) $\wedge$ taskUse(x2) $\wedge$ pre-tarefa(x1, x2) $\rightarrow \neg$ pre-tarefa(x2, x1))

Regra #17 – Não devem existir transições duplicadas entre duas atividades ou tarefas em um sequenciamento.

$\forall$x, x1, x2(workSequence(x) $\wedge$ ((activitvy(x1) $\wedge$ activity(x2)) $\vee$ (taskUse(x1) $\wedge$ taskUse(x2))) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2) $\wedge$ linkKind(x, 'SF') $\rightarrow \neg \exists$y (workSequence(y) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2)$\wedge$ linkKind(x, 'SF')))

$\forall$x, x1, x2 (workSequence(x) $\wedge$ ((activitvy(x1) $\wedge$ activity(x2)) $\vee$ (taskUse(x1) $\wedge$ taskUse(x2))) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2) $\wedge$ linkKind(x, 'FF') $\rightarrow \neg \exists$y (workSequence(y) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2)$\wedge$linkKind(x, 'FF')))

$\forall$x, x1, x2 (workSequence(x) $\wedge$ ((activitvy(x1) $\wedge$ activity(x2)) $\vee$ (taskUse(x1) $\wedge$ taskUse(x2))) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2) $\wedge$ linkKind(x, 'SS') $\rightarrow \neg \exists$y (workSequence(y) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2) $\wedge$ linkKind(x, 'SS')))

$\forall$x, x1, x2 (workSequence(x) $\wedge$ ((activitvy(x1) $\wedge$ activity(x2)) $\vee$ (taskUse(x1) $\wedge$ taskUse(x2))) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2) $\wedge$ linkKind(x, 'FS') $\rightarrow \neg \exists$y (workSequence(y) $\wedge$ predecessor(x, x1) $\wedge$ sucessor(x, x2) $\wedge$ linkKind(x, 'FS')))

Regra #18 – Todas as atividades ou tarefas que são sequenciadas em um processo de software deverão ter ligação com o nodo inicial e final do processo para garantir que toda atividade ou tarefa é iniciada após o nodo inicial e possui seu término antes do nodo final .

$\forall$x, y, z, w (activity(x) $\wedge$ (workSequence(y) $\wedge$ predecessor(y, x) $\vee$ sucessor(y,x)) $\wedge$ (activity(z) $\wedge$ specialNode(z, 'start')) $\wedge$ (activity(w) $\wedge$ specialNode(w, 'end')) $\rightarrow$ (pre-atividade(z,x) $\wedge$ pos-atividade (x,w)))

∀x, y, z, w (taskUse(x) ∧ (workSequence(y) ∧ predecessor(y, x) ∨ sucessor(y,x)) ∧ (taskUse(z) ∧ specialNode(z, 'start')) ∧ (taskUse(w) ∧ specialNode(w, 'end')) → (pre-tarefa(z,x) ∧ pos-tarefa (x,w)))

Regra # 41 – As atividades ou tarefas do processo que representam os nodos inicial e final não devem ser associados com instâncias da metaclasse ProcessPerformer.

∀x (((activity(x) ∧ specialNode(x,'end')) ∨ (activity(x) ∧ specialNode(x,'start')) ∨ (taskUse(x) ∧ specialNode(x,'start')) ∨ (taskUse(x) ∧ specialNode(x,'end'))) → ¬∃y (processPerformer(y) ∧ (linkedActivity(y, x) ∨ linkedTaskUse(y,x))))

Regra #42 – As atividades ou tarefas do processo que representam os nodos inicial e final não devem possuir elementos.

∀x (((activity(x) ∧ specialNode(x,'end')) ∨ (activity(x) ∧ specialNode(x,'start')) ∨ (taskUse(x) ∧ specialNode(x,'start')) ∨ (taskUse(x) ∧ specialNode(x,'end'))) → ¬∃a, b, c, d, e ,f, g, h, i ((roleUse(a) ∧ parte-de(a, x)) ∨ (activity(b) ∧ parte-de(b, x)) ∨ (externalUse(c) ∧ parte-de(c, x)) ∨ (internalUse(d) ∧ parte-de(d, x)) ∨ (milestone(e) ∧ parte-de(e, x)) ∨ (processParameter(f) ∧ parte-de(f, x)) ∨ (processPerformer(f) ∧ parte-de(f, x)) ∨ (workProductUseRelationship(g) ∧ parte-de(g, x)) ∨ (processResponsabilityAssignment(h) ∧ parte-de(h, x)) ∨ (workSequence(i) ∧ parte-de(i, x))))

Regra # 43 – O nodo inicial deve ser definido como predecessor para pelo menos uma atividade ou tarefa no sequenciamento definido em um processo de software.

∀x (((activity(x) ∧ specialNode(x,'start')) ∨ (taskUse(x) ∧ specialNode(x,'start'))) → ∃y (worksequence(y) ∧ predecessor(y,x)))

Regra # 44 – O nodo final deve ser definido como sucessor para pelo menos uma atividade ou tarefa no sequenciamento definido em um processo de software.

∀x (((activity(x) ∧ specialNode(x,'end')) ∨ (taskUse(x) ∧ specialNode(x,'end'))) → ∃y (worksequence(y) ∧ sucessor(y,x)))

Regra # 45 – O sequenciamento estabelecido entre as atividades ou tarefas e o nodo inicial ou final em um processo de software deve ser realizado através do tipo de sequenciamento finishToStart.

∀x,y, z ((((activity(x) ∧ specialNode(x,'start')) ∧ activity(y)) ∨ ((taskUse(x) ∧ specialNode(x,'start')) ∧ taskUse(y))) ∧ (workSequence(z) ∧ predecessor(z, x) ∧ sucessor(z,y)) → linkKind (z, 'FS'))

∀x,y, z ((((activity(x) ∧ specialNode(x,'end')) ∧ activity(y)) ∨ ((taskUse(x) ∧ specialNode(x,'end')) ∧ taskUse(y))) ∧ (workSequence(z) ∧ predecessor(z, y) ∧ sucessor(z,x)) → linkKind (z, 'FS'))

Regra #50 – Em um sequenciamento definido para um processo de software uma atividade não deve ter como predecessora e/ou sucessora uma tarefa, e vice-versa.

$\forall x,y,z\ (taskUse(x) \wedge (workSequence(y) \wedge (predecessor(y, x) \wedge sucessor(y, z)) \vee (predecessor(y, z) \wedge sucessor(y, x))) \rightarrow taskUse(z))$

$\forall x,y,z\ (activity(x) \wedge (workSequence(y) \wedge (predecessor(y, x) \wedge sucessor(y, z)) \vee (predecessor(y, z) \wedge sucessor(y, x))) \rightarrow activity(z))$