CLAYTON VIEIRA FRAGA FILHO

SIMULAÇÃO DE MODELOS DE PROCESSO DE SOFTWARE UTILIZANDO MÁQUINAS DE ESTADO ABSTRATAS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA

MINAS GERAIS - BRASIL

2009

CLAYTON VIEIRA FRAGA FILHO

SIMULAÇÃO DE MODELOS DE PROCESSO DE SOFTWARE UTILIZANDO MÁQUINAS DE ESTADO ABSTRATAS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 06 de Março de 2009.

Prof. Alcione de Paiva Oliveira
(Co-Orientador)

Prof. Vladimir Oliveira Di Iorio
(Co-Orientador)

Prof. Marco Túlio de Oliveira Valente

Prof. Ricardo dos Santos Ferreira

Prof. José Luis Braga

(Orientador)

Dedico essa dissertação à Fernanda, minha esposa, a João Vitor, meu filho amado, e a todos os meus familiares pelo incentivo constante e pela educação que me proporcionaram.

AGRADECIMENTOS

Agradeço primeiramente a Deus pela força, saúde, determinação e por iluminar meu caminho durante toda minha vida.

Aos meus pais pela vida, pelo amor, carinho e por acreditarem em mim.

A minha irmã pelo amor incondicional.

As minhas tias Marilda e Maria Helena que sempre estiveram presentes em todos os momentos e me ensinaram o valor do trabalho e a persistir nos momentos de dificuldade.

Ao Tio Heron pelo exemplo de vida, por mostrar que mesmo na dificuldade é possível vencer os obstáculos impostos pela vida.

Agradeço à Fernanda, minha esposa amada, guerreira, parceira, pela compreensão nos momentos de ausência, pelo companheirismo diante das dificuldades, pelo amor e por tudo o que representa para mim.

Ao meu amado filho João Vitor de 7 meses, que nasceu durante o desenvolvimento desta pesquisa, companheiro pelas madrugadas e que iluminou minha vida de maneira ímpar – indescritível.

Aos meus sogros, Vicente e Flávia por toda atenção e apoio dado desde o início do curso.

A toda minha família pelo suporte e dedicação.

Agradeço também ao amigo e orientador José Luis Braga, pela amizade, pelos ensinamentos, pelo apoio nos momentos de dificuldade e por acreditar nesta pesquisa.

Aos co-orientadores Vladimir e Alcione pela disponibilidade.

A UFV, e aos demais professores do Programa de Pós-Graduação em Ciência da Computação pelos momentos de aprendizado proporcionados.

Aos amigos do mestrado, especialmente ao Evaldo, Odilon e Marcelo Daibert pelas trocas e por sua amizade.

A todos, muito obrigado!

BIOGRAFIA

CLAYTON VIEIRA FRAGA FILHO, filho de Claiton Vieira Fraga e Magna Maria Schots Fraga, brasileiro nascido em 23 de agosto de 1978 na cidade de Manhumirim, no estado de Minas Gerais. Casado com Fernanda Vieira Mollica Fraga, e pai de João Vitor Mollica Fraga.

Concluiu o ensino médio na Escola Estadual de Manhumirim em 1996 e ingressou no curso de Sistemas de Informação na Faculdade de Viçosa em 2001, concluindo o curso de graduação em 2005.

Em 2006, foi aprovado na seleção do programa de pós-graduação do Departamento de Informática – DPI, onde, em maio de 2006, iniciou o mestrado em Ciência da Computação na Universidade Federal de Viçosa – UFV, defendendo sua dissertação em fevereiro de 2009.

Trabalhou como gestor de Tecnologia da informação, programador, analista de sistemas e gerente de projetos em empresas desenvolvedoras de software de 1996 a 2006, quando ingressou como docente no ensino superior.

SUMÁRIO

LISTA DE FIGURAS .... **vii**

LISTA DE TABELAS .... **ix**

LISTA DE QUADROS .... **x**

RESUMO .... **xi**

1 INTRODUÇÃO **1**

- **1.2 O problema da pesquisa** **3**
- **1.3 Objetivos** **4**
- **1.4 Organização da dissertação** **4**

2 REVISÃO BIBLIOGRÁFICA **6**

- **2.1 Processos de Desenvolvimento de Software** **6**
- **2.2 Meta-modelo de Engenharia de Processo de Software (SPEM)** **8**
  - 2.2.1 Elementos do SPEM 10
  - 2.2.2 A notação SPEM 13
- **2.3 Engenharia de Software dirigida por Modelo** **15**
  - 2.3.1 Arquitetura Dirigida por Modelos (MDA) 15
- **2.4 Máquinas de Estado Abstratas** **26**
  - 2.4.1 Vocabulários 28
  - 2.4.2 Estados e estados predefinidos 29
  - 2.4.3 Transições e programas ASM 29
  - 2.4.4 Agentes ASM 30
  - 2.4.5 Exemplo: Fatorial 31
- **2.5 Trabalhos relacionados** **32**

3 METODOLOGIA **33**

4 *SPEMASM* – UMA EXTENSÃO DO SPEM PARA EXECUÇÃO DE PROCESSOS DE SOFTWARE EM MÁQUINAS DE ESTADO ABSTRATAS **36**

- **4.1 O pacote Estrutural (*ProcessStructure*)** **36**
  - *4.1.1 Activity* 38
  - *4.1.2 ProcessRole* 43
  - *4.1.3 WorkProduct* 44
- **4.2 O pacote Ciclo de Vida (*ProcessLifeCycle*)** **45**
- **4.3 Considerações Finais** **47**

5 MAPEAMENTO DE PROCESSOS DE SOFTWARE EM MÁQUINAS DE ESTADO ABSTRATAS **48**

- **5.1 Mapeamento dos elementos estruturais** **49**
- **5.2 Mapeamento dos elementos de comportamento** **53**
  - 5.2.1 O Ciclo de Vida do Modelo ASM 54
- **5.3 Mapeamento de Restrições sobre o modelo** **65**
- **5.4 Considerações Finais** **66**

6 TRANSFORMAÇÃO DE MODELOS DE PROCESSOS PARA MÁQUINAS DE ESTADO ABSTRATAS 67
6.1 O modelo fonte 68
6.2 Marcas 76
6.3 O modelo fonte marcado 80
6.4 Transformação do modelo fonte marcado 82
6.5 Considerações Finais 87
7 SIMULAÇÃO DE PROCESSOS DE SOFTWARE COM O MODELO ALVO 89
7.1 Informações para simulação: o projeto 90
7.2 Simulação do projeto: Estudo de Caso 92
7.3 Considerações Finais 97
8 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS 99
8.1 Sugestões e Trabalhos Futuros 100
APÊNDICE A – Mapeamento estático 101
A1. Representação Estática dos elementos do SPEMasm para ASM 101
APÊNDICE B 114
B1. Modelo Específico de Plataforma na linguagem *AsmetaL* utilizada no estudo de Caso. 114
APÊNDICE C 124
C1. Registros dos estados do estudo de caso. 124
REFERÊNCIAS BIBLIOGRÁFICAS 129

LISTA DE FIGURAS

Figura 1– Ciclo de Vida de Processos de software. 7
Figura 2– Níveis de Modelagem definidos pela OMG 8
Figura 3 – Relação entre os conceitos de processo 9
Figura 4 – Pacotes do SPEM 10
Figura 5 – Pacote Estrutural de Processo 11
Figura 6 – Pacote Ciclo de Vida do Processo 12
Figura 7 – Relação entre fase, iteração e atividade. 13
Figura 8 – Fases e Iterações 13
Figura 9 – Atividades e produtos de trabalho genéricos do processo de desenvolvimento de software. 17
Figura 10 – Atividades e produtos de trabalho genéricos do processo de desenvolvimento de software na abordagem MDA. 18
Figura 11 – Relação entre os modelos de negócio (CIM) e de software. 19
Figura 12 – Representação geral da estrutura MDA 20
Figura 13 – PIM sendo transformado para mais de um PSM, que mantém a interoperabilidade utilizando pontes. 21
Figura 14 – Representação das Transformações entre modelos. 22
Figura 15 - Relação entre a transformação, o mapeamento e os modelos fonte e alvo. 23
Figura 16 - Mapeamentos um para um, um para muitos e muitos para um. 24
Figura 17 - Mapeamento de Instâncias 25
Figura 18 – Marcando um modelo. 25
Figura 19 - Mapeamento de instâncias com marcas providas pelo mapeamento. 26
Figura 20 - Representação gráfica dos estados a partir de uma computação. 28
Figura 21 - Interpretação do nome da função f. 31
Figura 22 – Esquema geral da metodologia para o desenvolvimento da pesquisa. 33
Figura 23 – Pacote Estrutural de Processo 37
Figura 24 – Pacote Estrutural de Processo do *SPEMasm* 37
Figura 25 – Associações complementares no Pacote Estrutural do SPEMasm 38
Figura 26 – Elemento Activity adaptado neste trabalho 39
Figura 27 – Precedência de Recurso de Trabalho 40
Figura 28 – Precedência de Produto de Trabalho 41
Figura 29 – Estados possíveis para uma atividade. 42
Figura 30 – Elemento *ProcessRole* adaptado neste trabalho 43
Figura 31 – Estados possíveis para um *ProcessRole*. 44
Figura 32 – Elemento WorkProduct 44
Figura 33 – Estados possíveis para um WorkProduct. 45
Figura 34 – Pacote Ciclo de Vida do SPEMasm 46
Figura 35 – Pacote Ciclo de Vida 46
Figura 36 – Pacote Estrutural de Processo do SPEMasm 50
Figura 37 – Ciclo de Vida do ASM proposto. 53
Figura 38 – Ciclo de Vida da *ProcessRole* mapeado no ASM. 55
Figura 39 – Ciclo de Vida da Activity no ASM. 57
Figura 40 – Ciclo de Vida de um WorkProduct do SPEMasm 59
Figura 41 – Ciclo de Vida de uma Iteração do SPEMasm 63
Figura 42 – Ciclo de Vida de uma Fase do SPEMasm 64
Figura 43 - Mapeamento de instâncias. 67
Figura 44 – Ciclo de Vida do *OpenUp/Basic* 68

Figura 45 – Composição da fase *Inception* 69
Figura 46 – Composição da iteração *Initiate Project* 69
Figura 47 – Ciclo de Vida do processo utilizado no estudo de caso 70
Figura 48 – Composição da fase Inception. 71
Figura 49 – Iteração Initiate Project 71
Figura 50 – Plan and Manage Iteration 72
Figura 51 – Composição da fase Elaboration 73
Figura 52 – Iteração Requirements and Architecture 73
Figura 53 – Estrutura baseada no MOF 76
Figura 54 – Domínios 77
Figura 55 – Funções 78
Figura 56 – Declaração do domínio abstrato *Activity.* 82
Figura 57 – Declaração do domínio abstrato *WorkDefinitionState.* 83
Figura 58 – Declaração do domínio abstrato *ProcessPerformerState.* 84
Figura 59 – Declaração do domínio abstrato *WorkProductState.* 85
Figura 60 – Estrutura de uma ASM 89
Figura 61 – Funcionamento da máquina de estado abstrata. 90
Figura 62 – Informações do projeto Estudo de Caso. 91
Figura 63 – Gráfico de Gantt do projeto 96
Figura 64 – Gráfico de Gantt do projeto atualizado com dados da simulação. 97

LISTA DE TABELAS

Tabela 1 – Estereótipos do SPEM..14
Tabela 2 – Mapeamento SPEM para ASM (Modelo de Mapeamento)..........................48
Tabela 3 – Exemplo de um modelo fonte marcado..74
Tabela 4 – Valores etiquetados na iteração *Initiate Project* 75
Tabela 5 – Modelo de Mapeamento e Marcas 78
Tabela 6 – Modelo de Mapeamento e Marcas detalhado..79
Tabela 7 – Modelo fonte marcado para a *Initiate Project* 80
Tabela 8 – Transformação do elemento estrutural *Activity*...83
Tabela 9 – Transformação do elemento estrutural *ProcessRole*.....................................84
Tabela 10 – Transformação do elemento estrutural *WorkProduct* 85
Tabela 11 – Transformação da instância do modelo fonte marcado para a iteração *Initiate Project* 85
Tabela 12 – Resumo dos produtos de trabalho produzidos, atualizados e utilizados no projeto Estudo de Caso. 91
Tabela 13 – Registro dos estados para o tempo decorrido das atividades. 94
Tabela 14 – Registro dos estados da precedência. 94
Tabela 15 – Registro dos estados da ordem de execução. 95
Tabela 16 – Registro dos estados da atividade *Define Vision* da ordem de execução....96

LISTA DE QUADROS

Quadro 1 – Regras de transição especificadas no ASM. 54
Quadro 2 – Regra *DefineProcessRoleParameters* 55
Quadro 3 – Regra Work para agentes *ProcessRole* 56
Quadro 4 – Regra *ActiveActivities* para atividades 56
Quadro 5 – Regra ExecuteActivities para atividades... 57
Quadro 6 – Regra MonitoreWorkingActivities para atividades 57
Quadro 7 – Regra MonitoreFinishedActivities para atividades.. 57
Quadro 8 – Regra DefineWorkDefinitionParameters para WorkDefinition.................. 58
Quadro 9 – Regra *DefineWorkProductParameters* 59
Quadro 10 – Regra MonitoreInitiatedWorkProducts... 60
Quadro 11 – Regra MonitoreOngoingWorkProducts 60
Quadro 12 – Regra MonitoreDoneWorkProducts 60
Quadro 13 – Regra MonitoreUpdatingWorkProducts 60
Quadro 14 – Regra MonitoreActivitiesWithUseWP 61
Quadro 15 – Regra *DefineDefaultOrders* 61
Quadro 16 – Regra *MonitoreActivitiesWithUpdateWP* 62
Quadro 17 – Regra MonitoreActivitiesPerformed... 62
Quadro 18 – Regra MonitoreIterations 63
Quadro 19 – Regra MonitoreReadyIterations.. 63
Quadro 20 – Regra MonitoreActiveIterations.. 64
Quadro 21 – Regra MonitorePhase 64
Quadro 22 – Regra MonitoreReadyPhases 65
Quadro 23 – Regra *MonitoreActivePhases* 65
Quadro 24 – Restrições sobre o Modelo 66
Quadro 25 – Informações do estudo de caso. 93
Quadro 26 – Regra *MonitoreActivitiesWithUpdateWP* 95
Quadro 27 – Regra *ExecuteActivities* para atividades.. 97

RESUMO

FRAGA FILHO, Clayton Vieira, M.Sc., Universidade Federal de Viçosa. Março de 2009. **Simulação de modelos de processo de software utilizando Máquinas de estado Abstratas**. Orientador: José Luis Braga. Co-Orientadores: Alcione de Paiva Oliveira e Vladimir Oliveira Di Iorio.

O estabelecimento de um modelo de processo de software em uma organização desenvolvedora de software não é uma atividade trivial. Muitas são as decisões a serem tomadas e abordagens utilizadas para dar suporte à tomada de decisão sobre quais atividades devem ser realizadas, como deve ser estabelecido seu sequenciamento, quais produtos de trabalho devem ser desenvolvidos, dentre outros. A primeira etapa para estabelecer o processo é definir sua representação estática, no entanto é importante que o processo seja simulado, demonstrando de forma dinâmica seu funcionamento. Este trabalho teve como objetivo realizar a transformação de modelos de processo de software de um alto nível de abstração para um nível que permita sua execução. O modelo estático denominado *SPEMasm* foi adaptado do SPEM e o modelo dinâmico foi mapeado para máquinas de estado abstratas (ASM). O mapeamento de instâncias foi utilizado para especificar o relacionamento dos elementos entre os modelos estáticos e dinâmicos, possibilitando a aplicação de uma técnica de transformação manual, baseada na abordagem da arquitetura dirigida por modelos, produzindo um modelo de simulação de processo de software com máquinas de estado abstratas utilizado no estudo de caso.

ABSTRACT

FRAGA FILHO, Clayton Vieira, M.Sc., Universidade Federal de Viçosa. March, 2009. **Software Process Model simulation using Abstract State Machines**. Adviser: José Luis Braga. Co-Advisers: Alcione de Paiva Oliveira and Vladimir Oliveira Di Iorio.

The definition of software process models in software devoloper organizations is a hard work. There are many decisions to be made and approaches used to decision support: what activities should be undertaken, their sequencing, what work products must be developed, among others. The first step is to obtain the process representation, that should be transformed in more detailed models to allow process simulation, essential to analyse its dynamic behavior. This work aimed to achieve software process model transformation from a high abstraction level to a lower abstraction level, that would enable process enactment. The static model was described using a SPEM extension, named SPEMasm, and the dynamic model was mapped to the formal language of Abstract State Machines (ASM), that was then properly executed. The mapping of instances was used to specify the relationship between the static and dynamic elements models, and was done manually, based on the model driven architecture approach. As a case study, we produced a specialization of a well known software process, that was simulated using the approach.

1 INTRODUÇÃO

O software tem desempenhado um papel crítico e essencial na sociedade, qualquer produto ou serviço moderno tem como componente ou utiliza software em algum momento (FUGGETTA, 2000). Desde o final da década de 60 a comunidade de Engenharia de Software realiza pesquisas e desenvolve práticas para a especificação e desenvolvimento de software com foco na redução dos custos e de retrabalho necessários para o seu desenvolvimento e manutenção (BONA, 2002). Apesar dos esforços, a literatura descreve como "Crise do Software" o fenômeno que reflete a incapacidade da indústria de software em atender satisfatoriamente às necessidades de um mercado consumidor cada vez mais exigente. Somente nos Estados Unidos foram gastos U$250 bilhões com o desenvolvimento de software no ano de 2002, sendo que destes, U$38 bilhões foram perdidos em projetos mal sucedidos e U$17 bilhões foram gastos extra, acima do previsto nos projetos (STANDISH, 2003).

A qualidade de software passou a ser um tema obrigatório na agenda de pesquisa e desenvolvimento da área. Duas abordagens principais passaram a ser o foco da qualidade: - qualidade baseada em produtos, como era o padrão da indústria de manufatura; - qualidade baseada em processos, em que o foco é nas etapas seguidas na produção. Com o tempo, entendeu-se que as duas abordagens devem ser complementares e nunca excludentes, na obtenção de software com qualidade. Ou seja, devem-se utilizar processos para desenvolvimento, associado com técnicas de teste de produtos intermediários como uma parte da garantia da qualidade.

Segundo (FUGGETTA, 2000), um processo de desenvolvimento de software é definido como um conjunto de políticas, estruturas organizacionais, ferramentas, procedimentos e artefatos que são necessárias para especificar, projetar, desenvolver, distribuir e manter um produto de software. Cada organização tem necessidade de processos de desenvolvimento de software especificados de acordo com a sua realidade e as suas necessidades. Não há garantia de que o uso de processos seja um remédio para todos os males da indústria de software, mas é um caminho promissor que já se mostrou efetivo em outras áreas industriais, como por exemplo, na indústria automobilística.

A adaptação de um processo para uma determinada empresa é uma atividade difícil, cheia de riscos e incertezas. Para saber se o processo vai funcionar ou não, o caminho a seguir é colocá-lo em uso no dia a dia da empresa, e ir fazendo os ajustes na medida do necessário. O ideal é que o processo possa ser modelado e colocado em execução por simulação antes de ser implantado, para que pelo menos os problemas mais visíveis sejam corrigidos e acertados antes da implantação no mundo real, evitando custos desnecessários e os imensos riscos envolvidos. Contudo, a modelagem de processos de software não é tarefa simples, a simulação do modelo do processo deve prover apoio automatizado para a sua execução (*enactment*) e acomodar atividades desempenhadas por seres humanos, que não podem ser automatizadas, como reuniões e negociações (REIS, 2002).

Um dos objetivos da modelagem é facilitar a comunicação e compreensão do processo. Diversas são as iniciativas de representar o processo de desenvolvimento de software para permitir seu compartilhamento e discussão com a equipe de desenvolvimento. Algumas utilizam a linguagem UML-Unified Modeling Language, criada para especificação de projetos de software (DI NITTO *et al*, 2002) e (RUÍZ *et al*, 2003). Segundo (RUMBAUGH, JACOBSON e BOOCH, 2000) a UML pode ser caracterizada como "*uma linguagem para especificar, visualizar e construir os artefatos de sistemas de software...*". Com seu poderoso recurso de extensão pelo uso dos Perfis (*Profile*) é possível estender a UML (OMG, 2008c) para domínios ou propósitos particulares, como por exemplo, para modelagem de processos de desenvolvimento de software.

A modelagem de processos de software e as ferramentas que a apóiam passou a ser um objetivo perseguido pela indústria de desenvolvimento de software. A partir da modelagem, e com ferramentas adequadas de simulação da especificação, é possível “ver” o processo em execução antes de ele ser colocado em prática, criando um laboratório para design de processos de valor inestimável para as empresas. O número de padrões e de empresas que se dedicam ao tema é muito grande, demonstrando a importância do tema.

1.2 O problema da pesquisa

Para (LAHOZ, 2004), a importância da modelagem de processos está em permitir que a visão global do processo e sua descrição sejam definidas. Seus principais objetivos são facilitar a comunicação da equipe, facilitar a melhoria do processo, permitir a gerência do processo, sua reutilização e suporte à execução automatizada. Com o modelo de processo de desenvolvimento de software definido, cada membro da equipe tem o suporte necessário para desempenhar seu papel no processo, com informações adequadas e fluxos de trabalho estabelecidos.

A simulação do processo é outro aspecto importante e permite que o processo seja verificado e validado antes mesmo de entrar em operação na organização. Com a simulação, erros no processo podem ser corrigidos, fluxos de trabalho podem ser revistos e melhorados, novos papéis e produtos de trabalho podem ser propostos, reduzindo os riscos, custos e retrabalho com o processo estabelecido e em uso.

Sem um modelo de processo estabelecido na organização, o produto final pode não corresponder às expectativas do cliente, a produtividade da equipe de desenvolvimento diminui pela falta de visibilidade do processo e a comunicação é prejudicada por não existir um arcabouço de atividades e papéis coerente para cada fase do processo.

Para permitir a execução do processo (*process enactment*) em um ambiente de simulação adequado e derivado a partir dos modelos obtidos da área de *workflow*[1] é necessário que o modelo de processo esteja formalizado em uma linguagem que tenha o poder expressivo suficiente para exibir as propriedades estáticas do processo e para representar suas propriedades dinâmicas. A partir dessa descrição do processo, torna-se possível interpretar os comandos da linguagem e proporcionar a desejada simulação do processo, em um ambiente que permita a interação com os analistas e especialistas em definição de processos de software.

[1] WorkflowManagement Coalition. http://www.aiim.org/wfmc

1.3 Objetivos

O objetivo geral do trabalho é realizar a transformação de modelos de processo de software em um alto nível de abstração para um nível que permita a execução do processo.

Pretende-se especificamente:

1. Selecionar uma linguagem adequada para representar estaticamente um modelo de processo de desenvolvimento de software, em um primeiro nível de abstração do modelo;
2. Selecionar uma linguagem adequada para representar o modelo de processo com suas propriedades estáticas e dinâmicas associadas, em um nível de abstração que permita a execução da especificação;
3. Definir semi-formalmente as regras de mapeamento entre os níveis de abstração apontados pelos objetivos 1 e 2 acima;
4. Validar o modelo gerado pela transformação por meio de um estudo de caso.

1.4 Organização da dissertação

Capítulo 1. Apresenta o problema e sua importância e os objetivos deste trabalho.

Capítulo 2. Apresenta os conceitos de modelo de processo de software, e o ciclo de vida de um modelo de processo de software. Descreve também o SPEM, um meta-modelo para representação de processos de software. Em seguida, é feita uma descrição da Engenharia de Software dirigida por modelos, e da arquitetura dirigida por modelos.

São apresentados os conceitos de máquinas de estado abstratas (Abstract State Machines - ASM) e finalmente os trabalhos relacionados a esta dissertação (GUREVICH, 1993), (BÖRGER e STÄRK, 2003).

Capítulo 3. Apresenta a metodologia para o desenvolvimento deste trabalho.

Capítulo 4. Este capítulo descreve o *SPEMasm*, adaptado do meta-modelo SPEM

para execução de processos de software com máquinas de estado abstratas. Adaptações foram realizadas nos elementos estruturais para permitir a definição do meta-modelo *SPEMasm*.

Capítulo 5. Descreve o mapeamento realizado do *SPEMasm* para os conceitos das máquinas de estado abstratas para definir o comportamento de um processo de software durante sua execução. O comportamento descrito é baseado nos elementos estruturais do *SPEMasm*, e é descrito formalmente em um alto nível de abstração, permitindo sua migração para diversas plataformas.

Capítulo 6. Neste capítulo, é realizada a transformação dos elementos estruturais e comportamentais definidos nos capítulos 5 e 7 usando a transformação manual, via mapeamento de instâncias da MDA. O modelo fonte definido segundo o *SPEMasm,* portanto independente de plataforma, é transformado no modelo alvo descrito na linguagem *AsmetaL* (ASMETA, 2007).

Capítulo 7. Apresenta um estudo de caso de simulação do processo transformado no capítulo 6 e a discussão dos resultados.

Capítulo 8. Apresenta uma síntese das contribuições do trabalho e sugestões de trabalhos futuros.

Apêndice A. Mapeamento estático dos elementos do *SPEMasm* para Máquinas de Estado Abstratas

Apêndice B. Modelo específico de plataforma na linguagem *AsmetaL*

Apêndice C. Registro dos estados produzidos no estudo de caso.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta o referencial teórico desta dissertação, descrevendo inicialmente as definições e os métodos utilizados na representação e arquitetura de processos de software.

2.1 Processos de Desenvolvimento de Software

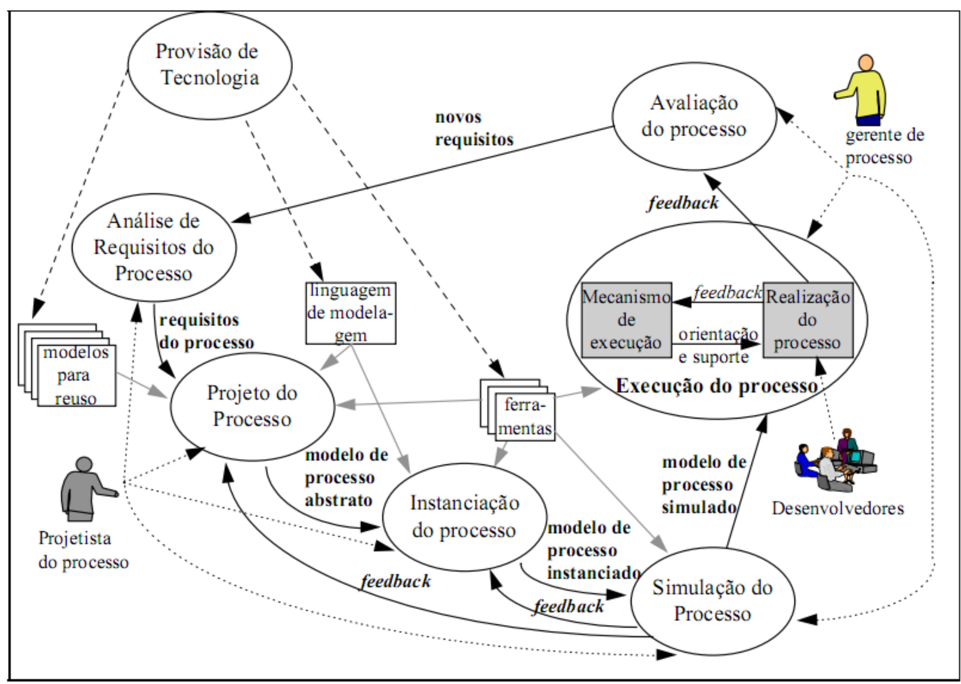
O processo de desenvolvimento de software é um processo de negócio com foco na produção de software. Envolve atividades que são desempenhadas por pessoas que atuam em papéis distintos em diferentes fases do processo, consumindo e construindo produtos de trabalho, que resultam no software (LONCHAMP, 1993).

Sommerville, 2003 afirma que há quatro atividades de processo fundamentais, comuns a todos os processos de software:

1. Especificação do software: Envolve a definição das funcionalidades e das restrições de um software.
2. Desenvolvimento do software: O produto de software deve ser produzido para atender a sua especificação.
3. Validação do software: O software depois de produzido precisa ser validado a fim de garantir que faça o que o cliente deseja.
4. Evolução de Software: Após a entrega, o software evolui para atender às mudanças das necessidades do cliente.

O modelo de processo de software representa uma descrição abstrata, e traz informações sobre quais atividades são desempenhadas, como são organizadas dentro de determinado período, quem são os atores que executam as atividades e são responsáveis pela produção dos artefatos intermediários do processo, tais como diagramas e documentos.

Segundo Reis (2002), o ciclo de vida de processos de desenvolvimento de software é definido pela Figura 1



Fonte: (REIS, 2002)

Figura 1– Ciclo de Vida de Processos de software.

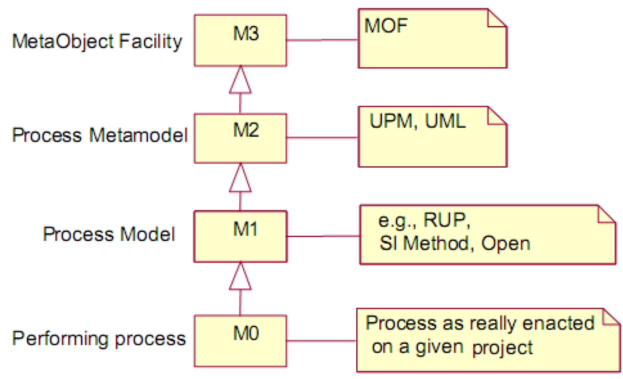
No ciclo apresentado, a Provisão de Tecnologia determina o fornecimento de tecnologia de suporte a produção do software e os produtos de trabalho associados, como diagramas, protótipos de interface com o usuário, ferramentas para testes, dentre outras. A Análise de Requisitos do Processo é realizada para identificar os requisitos do processo, para que o projeto seja realizado na fase de Projeto do Processo, que provê toda arquitetura do processo em detalhes, realizada em linguagens de modelagem de processo. A Instanciação do Processo inclui informações sobre prazos, pessoal que trabalhará no processo (atores ou agentes) e recursos utilizados, para permitir a Execução do modelo de Processo. Nesta etapa, informações sobre o andamento do processo (*feedback*) são coletadas e analisadas. Finalmente a Avaliação do Processo fornece informação quantitativa e qualitativa que descreve o desempenho de todo o processo em execução (REIS, 2002)

Neste trabalho, o Engenheiro de Processo representa um papel essencial para a manutenção dos modelos de processos existentes nas empresas de desenvolvimento. Ele deve desempenhar os papéis de Projetista e mantenedor (Gerente) do processo, e deve estar sempre atento à mudanças que sejam necessárias para o melhor desempenho destes processos, instanciados em projetos da empresa.

2.2 Meta-modelo de Engenharia de Processo de Software (SPEM)

Desenvolvido pelo *Object Management Group* (OMG), o *Software Process Engineering Metamodel* (SPEM) é um meta-modelo que pode ser usado para descrever um processo concreto ou uma família de processos de desenvolvimento de software relacionados. É classificada como uma linguagem de modelagem de processo não-executável, pois a execução (*enactment*) do processo não faz parte do seu escopo (OMG, 2005). No desenvolvimento deste trabalho, é utilizada a versão 1.1 do SPEM, contudo, durante seu desenvolvimento a OMG publicou a versão 2.0 do SPEM (OMG, 2008b).

Apresenta um conjunto de elementos de modelagem necessários para representar qualquer processo de desenvolvimento de software, utilizando uma abordagem orientada a objetos e a Unified Modeling Language (UML). Também é definido como um perfil da UML, que é um mecanismo de extensão da UML que permite a personalização de meta-modelos definidos pelo MOF para diferentes plataformas tecnológicas ou conceituais (OMG, 2006). A Figura 2 exibe os quatro níveis de modelos definidas pela OMG.

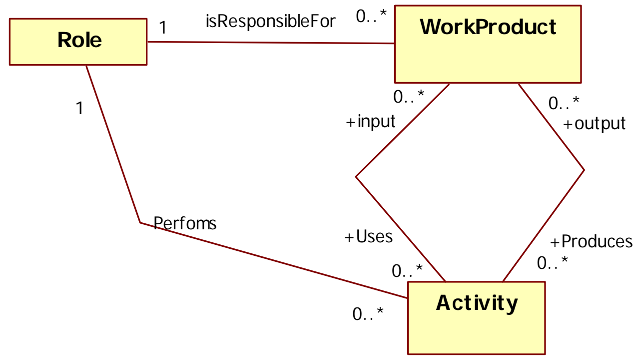


Fonte: (OMG, 2005)

Figura 2– Níveis de Modelagem definidos pela OMG

O SPEM é definido como meta-modelo, e encontra-se no nível M2, utilizando os blocos de construção definidos pelo MOF. Os elementos no nível M1 representam os modelos de processo, tais como RUP (KRUCHTEN, 2003) e *OpenUP* (KROLL, 2006), e outros modelos customizados e desenvolvidos por organizações desenvolvedoras de software. O nível M0 apresenta um modelo de processo instanciado, ou seja, que esteja sendo utilizado em um projeto de software em desenvolvimento para um cliente, em uma organização desenvolvedora.

O SPEM fornece uma notação para representar graficamente modelos de processos de desenvolvimento de software, e elementos que representam os conceitos utilizados nos modelos. A relação básica entre cada conceito é apresentada na Figura 3.



Fonte: (OMG, 2005)

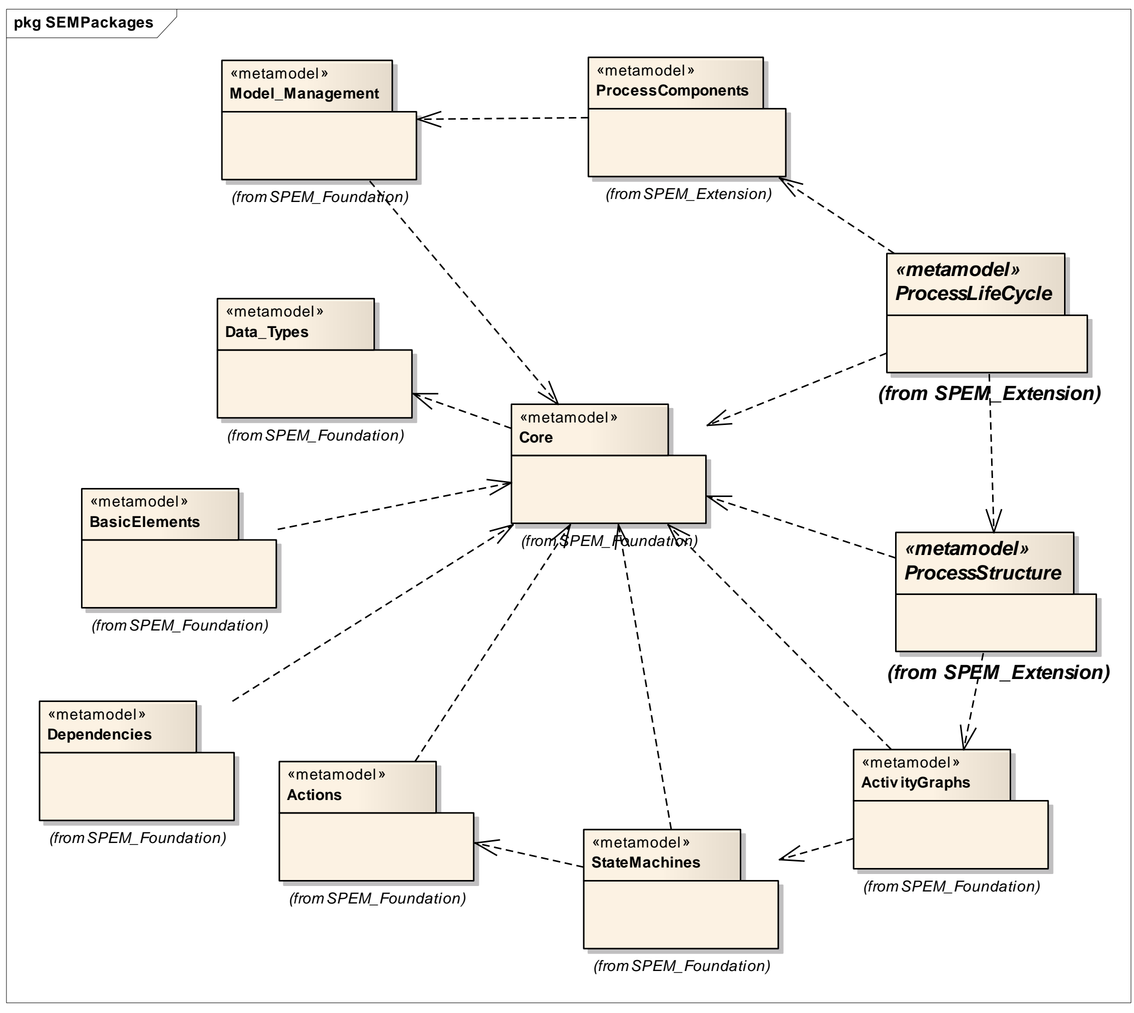
Figura 3 – Relação entre os conceitos de processo

Os atores são responsáveis pela realização das atividades (*Activity*) e assumem papéis (*Role*) no processo, tais quais Gerente de Projeto, Analista de Sistemas, Programador, dentre outros. A responsabilidade sobre os produtos de trabalho (*WorkProduct*) é dos atores que ao executar atividades, podem utilizar tais produtos de trabalho como entrada ou saída para a realização das atividades, indicando, por exemplo, que um documento ou diagrama é produzido em uma atividade. Kruchten (2003) enfatiza que o *Role* não é apenas um indivíduo ou um cargo, mas a responsabilidade e o comportamento que um indivíduo ou um grupo de indivíduos tem que assumir durante o processo. O comportamento é definido de acordo com as atividades que o ator desenvolve, já a responsabilidade é definida de acordo com os produtos de trabalho (*WorkProduct*) mantidos pelo ator, ao desempenhar determinado papel. O papel é desempenhado de acordo com a necessidade no momento em que o projeto acontece e de acordo com o que foi pré-estabelecido nas reuniões de planejamento. Pode ser necessário que um indivíduo atue hora como Analista de Sistemas e em outro momento como homologador de software, ou revisor de qualidade.

A realização de algumas atividades demanda informações produzidas em outra atividade do processo, e podem utilizar outros produtos de trabalho, produzidos em atividades anteriores do processo de desenvolvimento.

2.2.1 Elementos do SPEM

Os conceitos do SPEM são organizados em pacotes cuja divisão é feita em SPEM_*Foundations,* que trata blocos de construção baseados na UML 1.4 e o SPEM_*Extensions*, que descreve a semântica e os construtores para a representação de processos de software, conforme Figura 4.



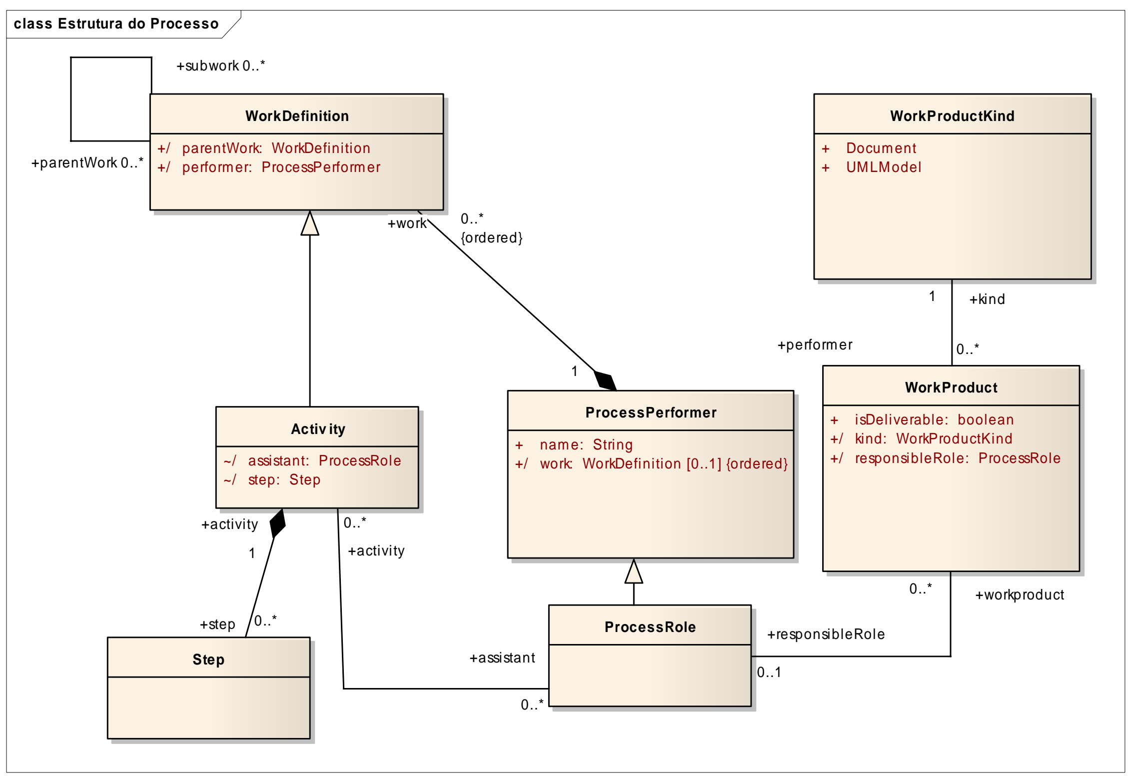
Fonte: (OMG, 2005)

Figura 4 – Pacotes do SPEM

Neste trabalho, somente os pacotes *ProcessStructure* (Estrutura do processo) e *ProcessLifeCycle* (Ciclo de Vida do Processo) serão considerados, pois representam os principais elementos para a representação do processo de software. A representação de componentes (*ProcessComponents*) é útil quando o foco é o encapsulamento de práticas do processo em vigor em pacotes e disciplinas, que ficarão disponíveis para uso em diversos modelos de processo utilizados por uma ou mais organizações.

O pacote *ProcessStructure* tem como objetivo apresentar um conjunto de elementos que são utilizados para a construção do processo de software. Toda

representação do processo é realizada utilizando os elementos apresentados neste pacote, exibido na Figura 5.



Fonte: (OMG, 2005)

Figura 5 – Pacote Estrutural de Processo

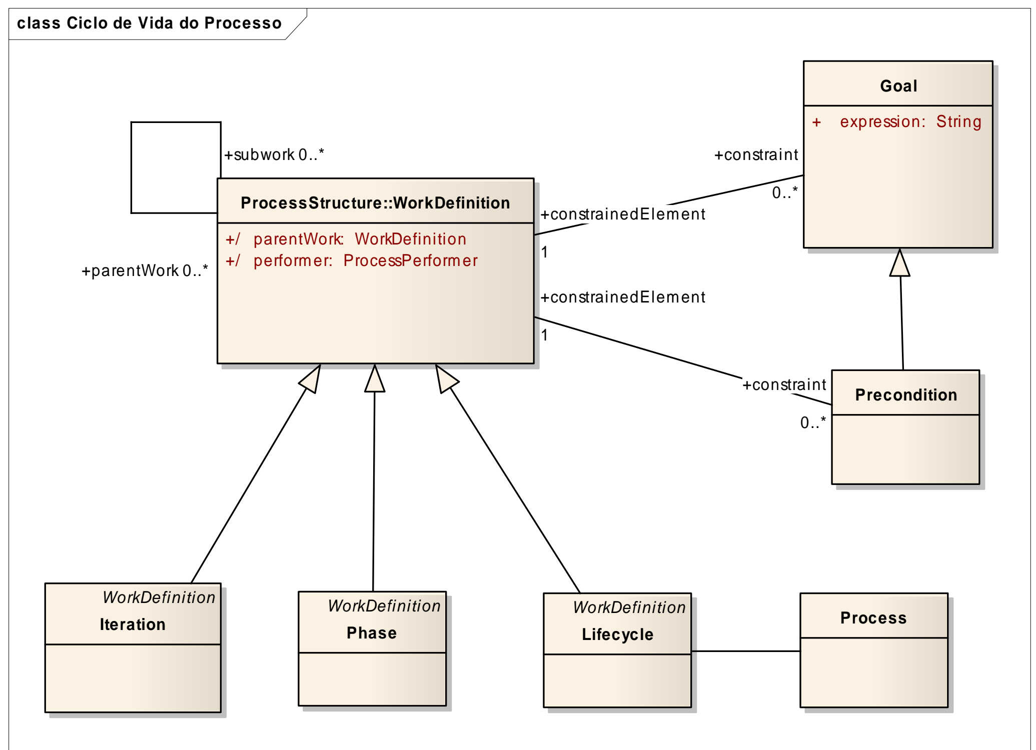
Um *WorkDefinition* tem como objetivo descrever um trabalho realizado no processo e sua principal especialização, *Activity* representa a atividade, ou o trabalho que é realizado em um processo de software. Uma atividade é composta por diversos passos (*Step*), que ao serem completados, dão como concluída a atividade.

O *ProcessPerformer* representa um responsável por um conjunto maior de trabalho realizado no processo, como iterações e fases que serão discutidas adiante. Sua especialização, o *ProcessRole* representa o responsável direto por uma ou mais atividades durante o processo de desenvolvimento de software e tem responsabilidade sobre um conjunto de produtos de trabalho produzidos (*WorkProduct*) ou utilizados nestas atividades.

Durante o desenvolvimento de um software, diversos são os tipos de produtos produzidos e utilizados. Sommervile (2003) relata que nas fases iniciais do desenvolvimento de software, são produzidos documentos de levantamento de requisitos. Na fase de projeto e *design*, diagramas UML (OMG, 2006) são produtos de trabalho comuns, cujo principal objetivo é representar a arquitetura do software e seu funcionamento. Nas fases posteriores, a codificação e os testes são realizados e código-

fonte é produzido. O SPEM utiliza a classe *WorkProductKind* para permitir que a diversidade de tipos de produtos de trabalho seja especificada.

O pacote ciclo de vida do processo (*ProcessLifeCycle*) fornece um conjunto de construtores para representar o ciclo de vida do modelo de processo, ou seja, como a composição e o sequenciamento de fases e iterações definem a ordem da execução do processo para que seu objetivo seja alcançado. A Figura 6 apresenta o pacote.



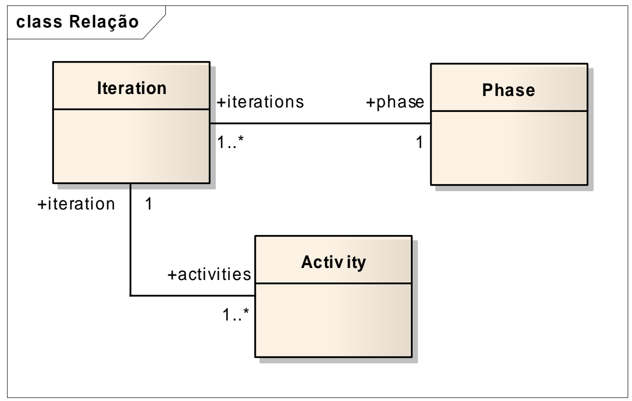
Fonte: (OMG, 2005)

Figura 6 – Pacote Ciclo de Vida do Processo

Assim como o elemento *Activity* apresentado no pacote Estrutural, os elementos *Iteration*, *Phase* e *Lifecycle* representam um conjunto ou parte do trabalho realizado (*WorkDefinition*) pelo processo de software (OMG, 2005).

Um *Lifecycle* descreve o sequenciamento de fases com foco em alcançar um objetivo no processo, definindo o comportamento completo do processo, quando instanciado para um projeto específico. O ciclo de vida pode ser definido de acordo com o modelo de processo utilizado, baseando-se nas pré-condições (*Precondition*) estabelecidas para início de cada iteração (*Iteration*) e fase (*Phase*). A *Phase* representa as fases do modelo de processo representado. As fases podem ser seqüenciais, baseadas no modelo cascata, ou iterativas, em que o início de uma fase não depende exatamente que a fase anterior seja finalizada, desde que seus objetivos (*Goal*) sejam alcançados. Em ambos os casos, elas são compostas de uma ou mais iterações (*Iteration*), que

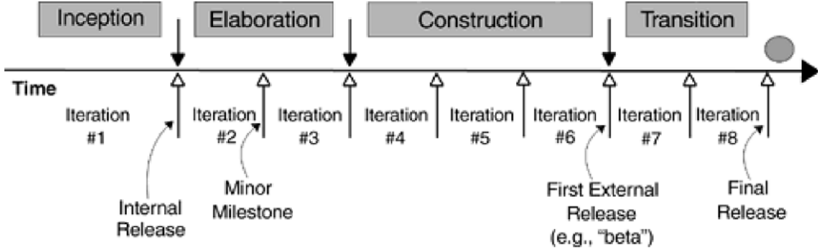
representam como as atividades são organizadas em um período determinado. A relação entre fase, iteração e atividade é apresentada na Figura 7.



Fonte: Elaborado pelo autor

Figura 7 – Relação entre fase, iteração e atividade.

Uma fase pode ter mais de uma iteração, que é definida como um período de tempo, com um marco (*Milestone*) definido como objetivo. A iteração (*Iteration*) é composta por um conjunto de atividades, que são executadas pelos atores que desempenham papéis no processo, conforme apresentado na Figura 8.



Fonte: (KRUCHTEN, 2003)

Figura 8 – Fases e Iterações

2.2.2 A notação SPEM

O SPEM utiliza diagramas da UML para representar diferentes perspectivas de um modelo de processo de software (OMG, 2005). Os diagramas utilizados são:

- Diagramas de Classes;
- Diagramas de Pacotes;
- Diagramas de Atividades;
- Diagramas de Caso de Uso;
- Diagramas de Seqüência; e
- Diagramas de Estados

Os diagramas de classes são utilizados no SPEM para representar aspectos estruturais de modelos de processo de software, como dependências, especialização de elementos e relações entre elementos do processo, tais como papel do processo (*ProcessRole*) e um produto de trabalho (*WorkProduct*).

Diagramas de Pacote são úteis para representação da composição de operações (*WorkDefinition*) e disciplinas já fundamentadas e que fazem parte do modelo de processo de uma organização. Um pacote apresenta de forma resumida o conteúdo do processo, que é definido principalmente por diagramas de classes, que apresentam os elementos estruturais do processo, ou seja, os elementos do meta-modelo, que são os blocos de construção usados para definição dos modelos.

O objetivo do diagrama de caso de uso no SPEM é demonstrar a relação entre os atores que executam os papéis no processo e as operações que são realizadas no processo (*WorkProduct*). Já o Diagrama de Seqüência, apresenta os padrões de interação entre as instâncias dos elementos do processo.

Diagramas de Estado representam a visão comportamental de cada elemento do processo, ou seja, qual o ciclo de vida de cada elemento, no decorrer da execução de um processo de software.

Os Diagramas de atividades permitem a representação seqüencial de fases, com detalhamento das iterações e cada atividade, com suas entradas, saídas e atores responsáveis (OMG, 2005).

Para que os diagramas incorporem a expressividade necessária para a descrição de modelos de processo de software, os elementos do SPEM são representados por estereótipos, que estendem o vocabulário da UML. Estereótipos têm representação gráfica (iconográficos) associadas ao domínio na qual o perfil está associado. A Tabela 1 apresenta os principais estereótipos do SPEM.

Tabela 1 – Estereótipos do SPEM

Estereótipo	Notação
<<WorkProduct>>	
<<WorkDefinition>>	
<<Guidance>>	
<<Activity>>	

<<ProcessPerformer>>	
<<ProcessRole>>	
<<ProcessPackage>>	
<<Phase>>	
<<Process>>	
<<Document>>	
<<UMLModel>>	

Fonte: (LAHOZ, 2004)

2.3 Engenharia de Software dirigida por Modelo

A Engenharia de Software dirigida por Modelos (*MDE - Model-Driven Engineering*) é uma área da Engenharia de Software em que modelos são artefatos primários no processo de desenvolvimento de software. A meta-modelagem é o conceito fundamental da MDE, cujo objetivo é separar de forma modular e em camadas uma notação abstrata de um formalismo, como uma implementação em uma linguagem de programação (FAVRE, 2004).

No contexto da MDE, a OMG definiu a arquitetura dirigida por modelos (*MDA – Model-Driven Architecture*), cuja idéia é separar a especificação de um modelo dos detalhes de sua especificação em uma plataforma específica visando aumentar sua portabilidade, interoperabilidade e reusabilidade através da separação arquitetural de interesses (CUNHA, 2004). A MDA é apresentada na próxima seção.

2.3.1 Arquitetura Dirigida por Modelos (MDA)

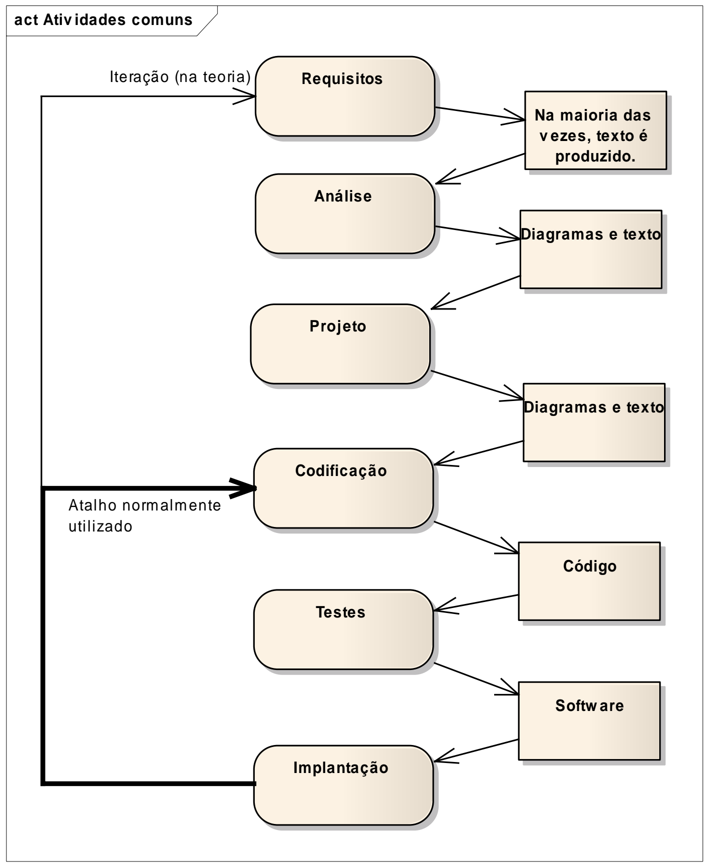
A MDA é uma abordagem de desenvolvimento de software orientada por modelos, cujo principal objetivo é separar a lógica do negócio da plataforma tecnológica em que tal lógica, ou seja, as regras de negócio serão implementadas e

transformadas em software. Seus principais artefatos de desenvolvimento são os modelos, que dirigem todo o processo, desde o início, quando as primeiras informações sobre o negócio são levantadas e especificadas junto aos clientes, projeto, construção (desenvolvimento), implantação e a evolução do software (OMG, 2003).

A principal diferença da MDA e outros métodos de desenvolvimento de software é a utilização de modelos em diferentes níveis de abstração, ou seja, a especificação do software é totalmente independente da sua operação. As definições a seguir são baseadas em (OMG, 2003).

- Modelo: É caracterizado na MDA como uma especificação de parte de uma função, do comportamento e da estrutura de um sistema. Desenvolvimento dirigido por modelos significa utilizar modelos para o entendimento da especificação, projeto, construção, implantação e mudança de um sistema, portanto, são artefatos principais do desenvolvimento.
- Arquitetura: É a especificação dos módulos, e interfaces de conexões de um sistema, e as regras para interação entre esses módulos utilizando tais interfaces. A MDA descreve tipos de modelos e como esses modelos se relacionam com outros modelos para que o sistema seja produzido.
- Plataforma: É um conjunto de subsistemas e tecnologias que provêm um conjunto de funcionalidades e especificações de padrões de uso, da forma que qualquer sistema desenvolvido sobre essa plataforma utilizará seus serviços e funções sem se preocupar como eles são implementados (CALIARI, 2007).

Métodos tradicionais de desenvolvimento de software estruturam as atividades de desenvolvimento em fases distintas, que podem ser executadas de forma iterativa e incremental, espiral ou cascata. Independente do modelo de ciclo de vida utilizado, as atividades apresentadas na Figura 9 são comuns para projetos de desenvolvimento de software.



Fonte: Traduzido de (KLEPPE, WARMER, BAST, 2003)

Figura 9 – Atividades e produtos de trabalho genéricos do processo de desenvolvimento de software.

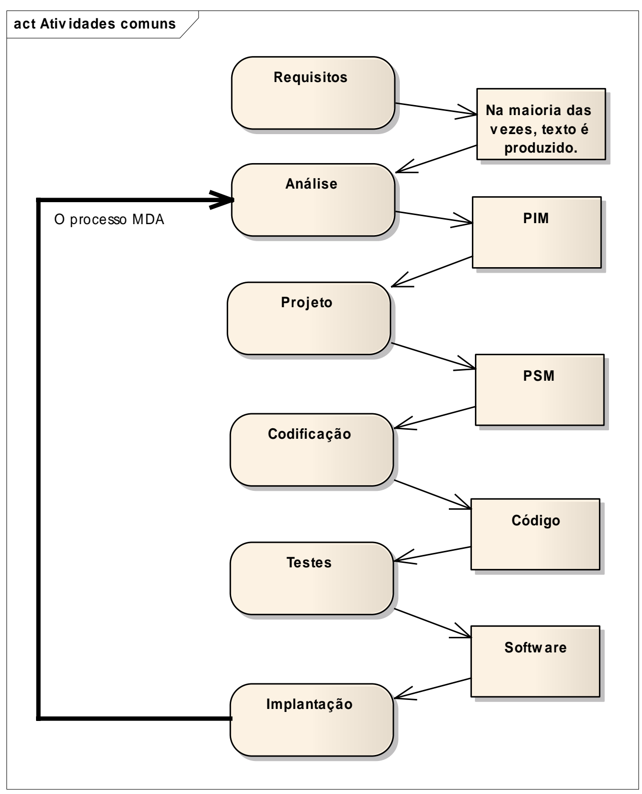
A Figura 9 apresenta a seqüência sugerida em que cada atividade ocorre desde o levantamento dos requisitos, até a implantação do software para o cliente. Com o passar do tempo, os requisitos do cliente mudam, e mudanças no software são necessárias para adequação à nova realidade do negócio.

Para que os novos requisitos sejam entendidos e o projeto seja feito em conformidade com os padrões, técnicas e tecnologias utilizados na empresa, a equipe de desenvolvimento deveria repetir o ciclo, ou seja, iniciar com o entendimento dos requisitos e seguir cada atividade até que a nova versão possa ser disponibilizada. Infelizmente não é o que ocorre nas empresas de desenvolvimento de software. Normalmente, os próprios desenvolvedores que fizeram a codificação, é que recebem as solicitações de mudança e iniciam o trabalho, porém na fase de codificação.

Tal prática é comum, contudo representa um risco para o cliente e para o negócio da empresa desenvolvedora de software, pois todo o conhecimento intrínseco à mudança solicitada não é incorporado à especificação de requisitos, assim como ao projeto lógico e de arquitetura do software alterado (KLEPPE, WARMER e BAST,

2003). Outro problema, é que na abordagem tradicional, o conhecimento do negócio é projetado e passa a fazer parte da arquitetura tecnológica do software. Qualquer mudança nos requisitos, demanda que parte da análise seja realizada, incluindo análise de impacto em outros componentes do software, em seguida, o projeto é realizado e a codificação. Uma mudança tecnológica também demanda a revisão do projeto e das abstrações do negócio realizadas considerando os recursos presentes na tecnologia anterior.

Na abordagem MDA, as atividades apresentadas na Figura 9 são realizadas de forma diferente, conforme apresentado na Figura 10.



Fonte: Traduzido de (KLEPPE, WARMER, BAST, 2003)

Figura 10 – Atividades e produtos de trabalho genéricos do processo de desenvolvimento de software na abordagem MDA.

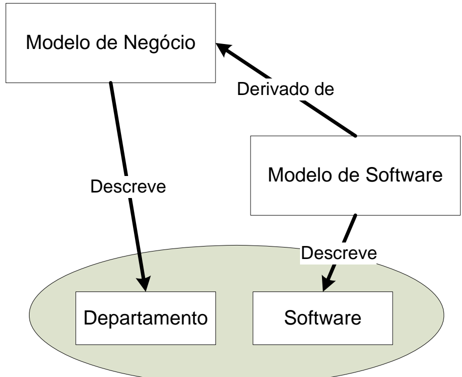
As atividades de desenvolvimento de software não são modificadas na abordagem MDA, contudo, a natureza dos produtos de trabalho sim. Os produtos de trabalho são modelos formais, que podem ser interpretados e “entendidos” pelo computador. Tais conceitos representam o núcleo da MDA e são apresentados a seguir.

2.2.2.1 Tipos de modelos da MDA

Nesta seção são apresentados os tipos de modelos utilizados na MDA para a especificação e desenvolvimento de sistemas.

2.2.2.1.1 Modelo Independente de Computação (CIM)

O Modelo Independente de Computação (*Computation Independent Model - CIM*) tem como principal foco os requisitos do negócio para o qual o sistema está sendo definido. Não apresenta qualquer informação sobre a arquitetura ou a plataforma utilizada pelo sistema, apenas utiliza o vocabulário utilizado no negócio pelos clientes para descrevê-lo, portanto pode ser um diagrama de blocos, ou mesmo uma descrição textual (OMG, 2003) e é também chamado de modelo de domínio ou de negócio. Sua relação com o modelos de software é apresentada na Figura 11.



Fonte: Traduzido de (KLEPPE, WARMER, BAST, 2003)

Figura 11 – Relação entre os modelos de negócio (CIM) e de software.

Os requisitos expressados no CIM devem ser mapeados em outros modelos da MDA, apresentados nas seções adiante (CALIARI, 2007).

2.2.2.1.2 Modelo Independente de Plataforma (PIM)

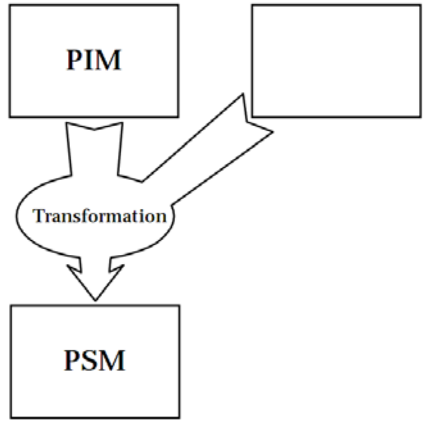
Após a definição do CIM, deve-se gerar o Modelo Independente de Plataforma (*Platform Independent Model* – PIM), normalmente de maneira manual (CALIARI, 2007). O PIM deve ser totalmente independente de tecnologia e de plataformas, de modo que com um PIM seja possível gerar softwares em diversas tecnologias e linguagens. Deve descrever detalhes de operação de um sistema, contudo, nenhum

detalhe específico de tecnologia ou plataforma deve estar presente, permitindo que o PIM seja reutilizado para diferentes plataformas.

O PIM deve ser especificado em uma linguagem de modelagem genérica, ou que seja aderente ao domínio do problema que está sendo tratado. Importante salientar que os detalhes de serviços e funções requeridos podem ser descritos, mas de modo que possam ser mapeados para diversas plataformas.

2.2.2.1.3 Modelo Específico de Plataforma (PSM)

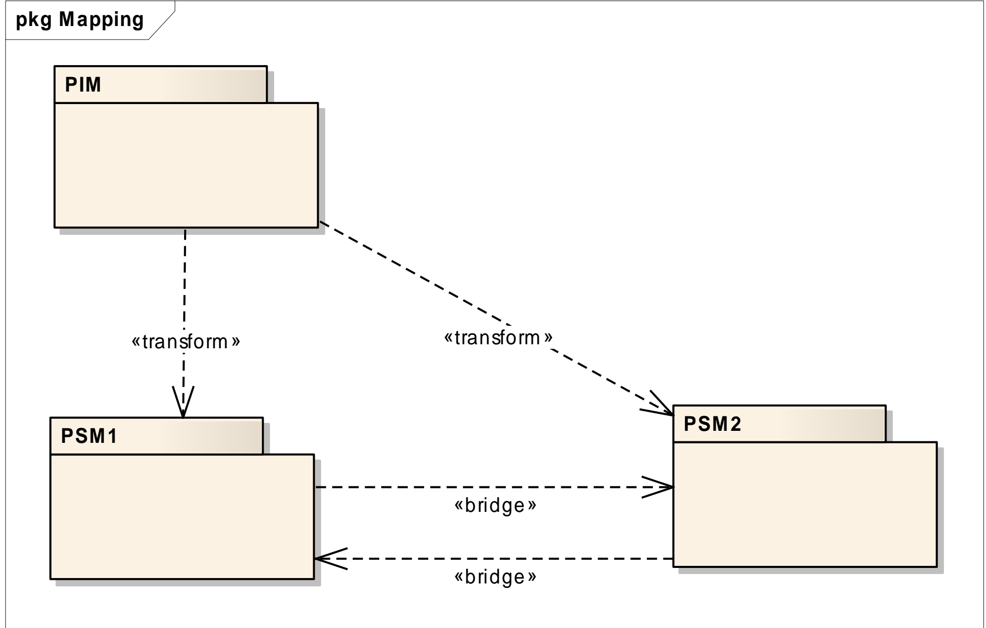
Com o PIM definido, regras de transformação devem ser definidas de modo que um ou mais Modelos Específicos de Plataforma (*Platform Specific Model* – PSM) possam ser produzidos. Na transformação de modelos regras definem como os conceitos do PIM são mapeados para o PSM, através dos modelos de mapeamento, que são descritos na seção 2.2.2.2. A Figura 12 apresenta o esquema geral utilizado pela MDA.



Fonte: (OMG, 2003)

Figura 12 – Representação geral da estrutura MDA

O quadro em branco representa as diferentes formas que a transformação pode ser realizada. No PSM a arquitetura e implementação das funções de uma plataforma são descritas em nível de detalhes, para permitir que a geração de produtos de trabalho possa ser seja realizada. Em um mesmo sistema, um PIM pode ser transformado em vários PSM de acordo com a necessidade dos serviços e funções desempenhados, e com a abstração que tenha sido realizada para as arquiteturas de destino, conforme apresentado na Figura 13.



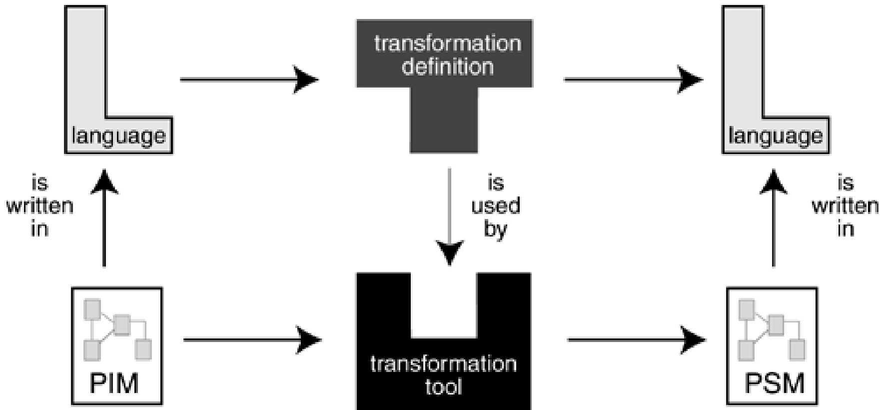
Fonte: Traduzido de (DUC, 2007)

Figura 13 – PIM sendo transformado para mais de um PSM, que mantém a interoperabilidade utilizando pontes.

Na Figura 13 um PIM é mapeado para dois PSM, que mantém uma ponte (*brigde*) para que conexões entre módulos e interfaces sejam realizadas. Caso o sistema especificado na Figura 13 utilizasse o padrão de projeto MVC, o PSM1 poderia representar a camada de Modelo (*Model*) e o PSM2, a camada de apresentação, ou interface gráfica (*View*). As pontes, ou conexões entre eles, representam a forma como tais camadas se comunicam. O PIM seria apenas um, e representaria o modelo de negócios.

2.2.2.2 Transformação de Modelos

Segundo Caliari (2007), a transformação é a atividade de gerar um modelo alvo a partir de um modelo fonte, baseando-se em um mapeamento, que define regras de mapeamento e juntas descrevem como um modelo em uma linguagem fonte pode ser transformado em um modelo na linguagem alvo, conforme Figura 14.



Fonte: (KLEPPE, WARMER, BAST, 2003)

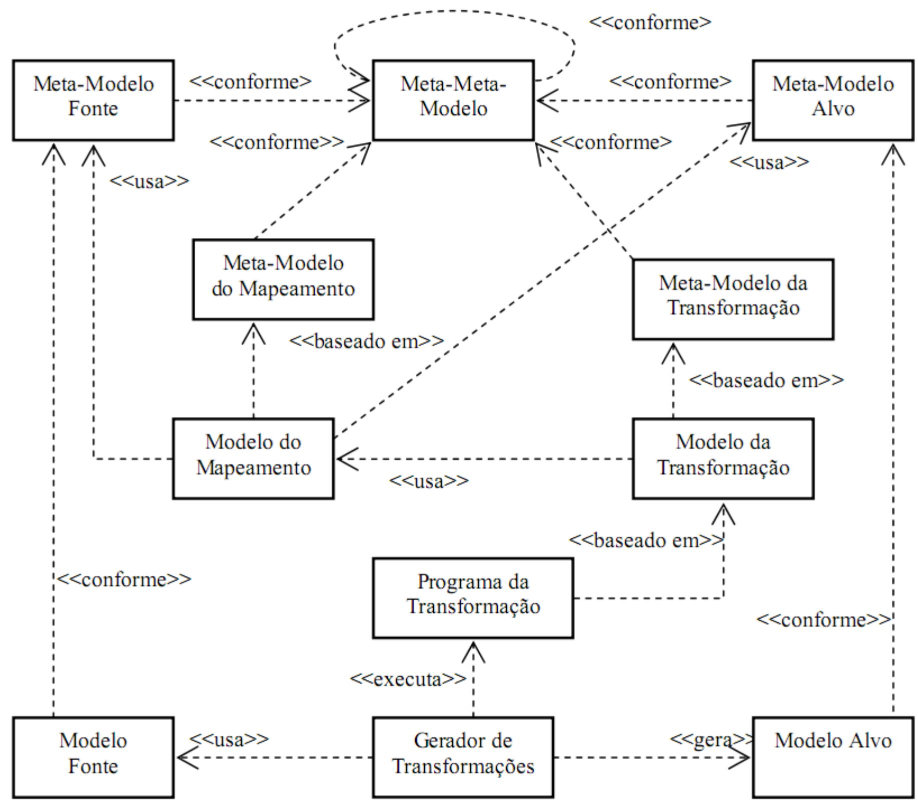
Figura 14 – Representação das Transformações entre modelos.

A definição da transformação é realizada através de modelos de mapeamentos, que definem como os elementos devem ser mapeados de um modelo para outro. CALIARI *apud* (LOPES 2005) define formalmente mapeamento e transformações:

> "Dado M*1*(s)/Ma e M*2*(s)/M*b*, em que M*1* é um modelo de um sistema *s* criado utilizando o meta-modelo M*a* e M*2* é um modelo do mesmo sistema *s* criado utilizando o meta- modelo M*b*, um mapeamento pode ser definido como $CMa \rightarrow Mb/Mc$, em que $CMa \rightarrow Mb$ é o mapeamento entre os meta- modelos M*a* e M*b* criado com a utilização do meta-modelo Mc. Uma transformação pode ser definida como a função $Transf(M1(s)/Ma, CMa \rightarrow Mb/Mc) \rightarrow M2(s)/Mb$".

Assim sendo, o modelo $M1(s)/Ma$ e o mapeamento $C_{Ma \rightarrow Mb/Mc}$ são utilizados como entrada na transformação $Transf(M1(s)/Ma, C_{Ma \rightarrow Mb/Mc}) \rightarrow M2(s)/Mb$ cuja saída é o modelo M2(*s*)/M*b.*

A Figura 15 apresenta o processo de transformação e todos os elementos envolvidos.



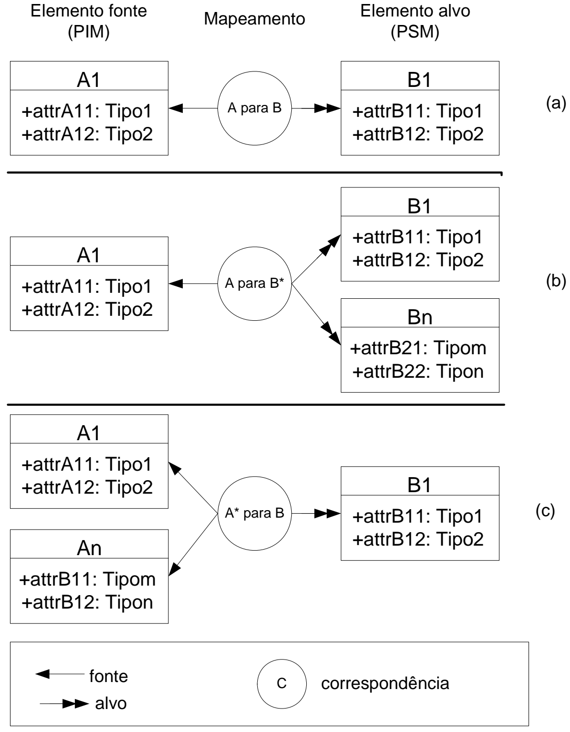
Fonte: (CALIARI, 2007)

Figura 15 - Relação entre a transformação, o mapeamento e os modelos fonte e alvo.

Os meta-modelos fonte, alvo e da transformação são baseados em um meta-meta-modelo, como por exemplo, o MOF. Em um primeiro passo é gerado um modelo de transformação baseado no meta-modelo de transformação. O modelo de transformação utiliza o modelo de mapeamento, que é responsável pela definição das regras que definem como os elementos do modelo fonte são mapeados para o modelo alvo. O modelo fonte deve ser definido com base no meta-modelo fonte. O modelo fonte serve como entrada para o gerador de transformação que executa o programa de transformação, que efetivamente transforma os elementos segundo definido no modelo de mapeamento, produzindo como saída um modelo alvo, conforme meta-modelo alvo. A transformação pode ser realizada de maneira manual, com assistência de um computador, ou totalmente automatizada (OMG, 2003).

2.2.2.3 Mapeamentos

Um mapeamento na MDA provê especificações para transformações de PIM em PSM para uma plataforma em particular. O modelo de plataforma determinará a natureza do mapeamento (OMG, 2003). O mapeamento pode ser realizado de um conceito do PIM para um conceito do PSM – denominado mapeamento um para um (a). Contudo é possível realizar o mapeamento de um elemento do PIM para vários elementos correspondentes no PSM (b), como também o mapeamento de vários elementos do PIM em um elemento do PSM (c), conforme exibido na Figura 16.



Fonte: (LOPES, 2005)

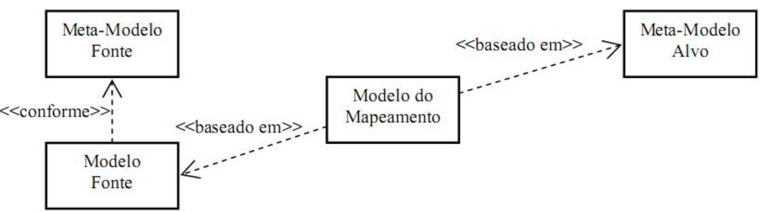
Figura 16 - Mapeamentos um para um, um para muitos e muitos para um.

O modelo de mapeamento deve descrever as regras de transformação para todos os casos apresentados na Figura 16 de maneira detalhada, ou seja, deve especificar como os elementos do modelo fonte, devem ser transformados em um ou mais elementos do modelo alvo.

A MDA define formas de realizar o mapeamento: mapeamentos de tipos e os mapeamentos de instâncias. O mapeamento de tipos mapeia tipos da linguagem do PIM em tipos da linguagem do PSM, mas não é descrito em detalhes neste trabalho, já que o mesmo utiliza o mapeamento de instâncias.

2.2.2.3.1 Mapeamento de instâncias

O mapeamento de instância especifica que os elementos do PIM devem ser transformados de forma específica em elementos do PSM, de acordo com a plataforma definida para o PSM, dada uma correspondência entre o elemento do PIM e um ou mais tipos do meta-modelo PSM (SENDALL, 2003). A Figura 17 apresenta o esquema geral.

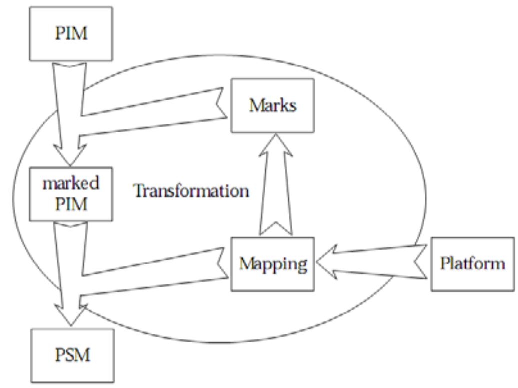


Fonte: (CALIARI, 2007)

Figura 17 - Mapeamento de Instâncias

A OMG apresenta o conceito de "marcas" como mecanismos que indicam de que forma elementos do PIM devem ser transformados para o PSM, mas não fazem parte do PIM, e sim do PSM. Sendo assim, no caso em que um PIM necessite ser mapeado para outros três PSM, com o mapeamento de instâncias é necessário definir três modelos de marca. A diferença entre os mapeamentos de instâncias e o mapeamentos de tipos e seus derivados, está na utilização de anotações feitas com marcas nos elementos dos modelos, ao invés da utilização de tipos.

A utilização de marcas na MDA gera um novo modelo, denominado PIM marcado, que contém o PIM original, com as marcas definidas, que é utilizado como entrada pelo gerador de transformações, conforme apresentado na Figura 18



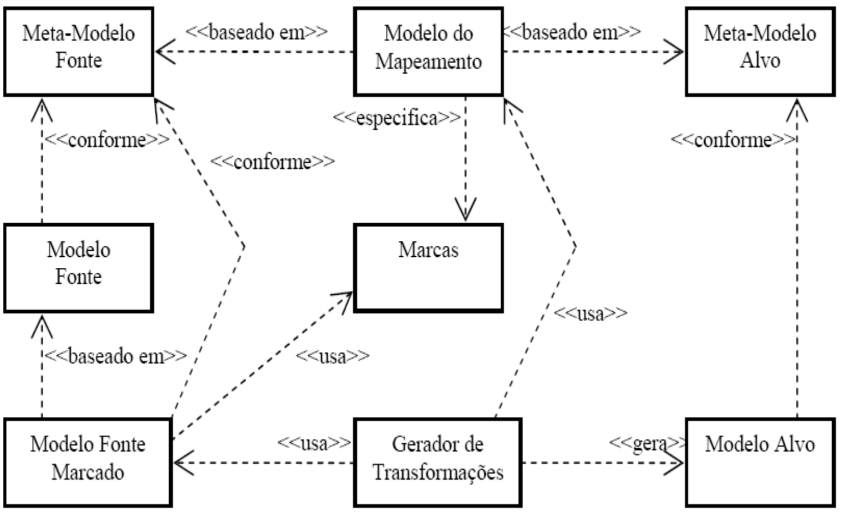
Fonte: (OMG, 2003)

Figura 18 – Marcando um modelo.

A OMG define que Marcas podem ser originadas de diferentes fontes, tais como (OMG, 2003):

- Tipos de um modelo, definidos por classes, associações e outros elementos;
- Padrões de um modelo, que descrevem papéis e atividades;
- Estereótipos de um perfil UML ;
- Elementos do meta-meta-modelo MOF;
- Elementos de um modelos especificado por um meta-modelo qualquer.

Caliari (2007) descreve em seu trabalho que marcas podem ser providas pelo mapeamento, por um perfil da UML ou por um modelo de marcas. Modelos marcados pelo mapeamento são específicos da plataforma do modelo alvo e por isso só podem ser utilizadas para a plataforma alvo. Neste trabalho as marcas providas pelo mapeamento são utilizadas para a realização das transformações. A Figura 19 apresenta a estrutura para que marcas sejam utilizadas.



Fonte: (CALIARI, 2003)

Figura 19 - Mapeamento de instâncias com marcas providas pelo mapeamento.

O modelo de mapeamento especifica as marcas que serão utilizadas pelo modelo fonte marcado, ou PIM marcado, que é utilizado pelo gerador de transformações para produzir o modelo alvo.

2.4 Máquinas de Estado Abstratas

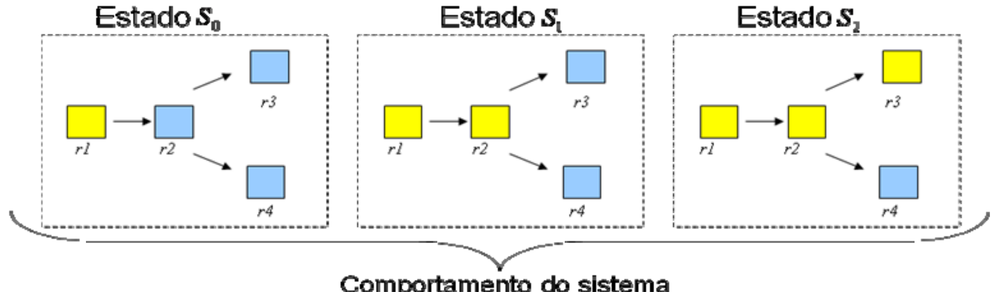
Máquinas de Estado Abstratas (*Abstract State Machines - ASM*) são um formalismo criado por Yuri Guerevich, originalmente conhecido por *Evolving Algebras*.

Constituem um conceito expressivo e elegante para modelagem matemática de sistemas dinâmicos discretos (GUREVICH, 1993), (BÖRGER e STÄRK, 2003). Valente (1999) define que constituem um modelo computacional capaz de especificar qualquer algoritmo seqüencial em seu nível natural de abstração. Segundo Ober (2001), ASM tem sido utilizada para especificar a semântica de diversas linguagens de programação, como C++, Prolog, como também para modelagem formal de algoritmos e descrição de arquitetura de hardware. Outras aplicações de ASM são a verificação de modelos UML (SHEN, 2005), especificação de sistemas de tempo real (OUIMET e LUNDQVIST, 2007), geração de casos de testes para casos de uso (GRIESKAMP *et al.*, 2003), definição, verificação e validação da máquina virtual da linguagem Java (STÄRK, SCHMID e BÖRGER, 2001).

São capazes de modelar um sistema e simular sua execução, modificando o estado global deste sistema (estado da máquina) de acordo com um conjunto de regras de transição (programa ASM) (OBER, 2001).

Em uma ASM, um estado S é uma álgebra definida por um vocabulário Υ de nome de funções, um conjunto X chamado de superuniverso e uma função de interpretação Val_S dos nomes do vocabulário em funções de $X^* \rightarrow X$. Regras de transição são utilizadas para modificar a interpretação dos nomes de função de um estado para outro, isto é, a função Val_S (VALENTE, 1999). Sendo assim a execução de um programa ASM consiste no disparo de regras de transição que fazem com que a máquina mude de estados sucessivamente, até endereçar o estado final. Uma regra de transição de ASM tem a forma de um programa de uma linguagem imperativa, contudo, a principal diferença é a ausência de iteração, já que o conceito está implícito na execução da máquina.

A Figura 20 apresenta um diagrama que exibe o comportamento geral de um sistema, dado pela mudança de estados realizada a cada computação do ASM.



Fonte: Elaborado pelo autor

Figura 20 - Representação gráfica dos estados a partir de uma computação.

No estado S_0 uma função, representada pelos nós, tem seu valor modificado pelas regras r_n se a condição para sua atualização (mudança de valor) for verdadeira, como a seguir:

if *Condition* ***then*** *Updates*

O disparo das regras é realizado de forma paralela, desde que as condições para a execução da regra sejam satisfeitas no estado corrente para as funções apresentadas no vocabulário, já que um programa ASM pode ser executado para cada agente definido. Os agentes executam de forma concorrente seus programas ASM e interagem globalmente compartilhando os endereços dos estados (OBER, 2001). Neste trabalho, agentes são representados pelos atores do processo (*ProcessRole*), que desempenham funções, tais como Analistas de Sistemas, Programadores, Gerente de Projetos, Arquitetos de Software, dentre outros (OMG 2005).

2.4.1 Vocabulários

Um vocabulário ASM é constituído de um conjunto finito de funções, predicados e domínios, geralmente definidos como *nomes.* Todos os nomes formam o vocabulário.

Todo sistema é formado de elementos básicos, agrupados no ASM dentro de *domínios* (também chamados de *universos* e *conjuntos*), para cada categoria de objetos. Estes domínios podem ser completamente abstratos, quando nenhuma restrição é imposta sobre eles. Por outro lado, os objetos que fazem parte de um domínio podem ter propriedades e até mesmo serem relacionados com outros objetos do mesmo domínio ou de outros domínios.

O modelo ASM pode conter funções matemáticas que se relacionam com determinado domínio. Considerando a mudança ou não de valores de uma função em vários estados ASM, as funções podem ser dinâmicas ou estáticas. Funções estáticas, representam apenas a estrutura básica de um sistema, enquanto funções dinâmicas refletem o comportamento dinâmico do sistema (OBER, 2001).

Os predicados são um tipo especial de função que retornam valores booleanos e podem ser utilizados para expressar restrições ou axiomas, e também podem ser estáticos ou dinâmicos.

Os nomes dinâmicos do ASM são classificados em quatro categorias (BÖRGER e STÄRK, 2003):

- controlados (*controlled)* – só pode ser modificado por um agente ASM como resultado da execução de algum programa ASM.
- monitorados (*monitored*) – só pode ser modificado pelo ambiente e o agente ASM apenas pode avaliá-lo.
- compartilhado (*shared*) – pode ser modificado tanto pelo ambiente quanto pelo agente ASM.
- saída (out) – podem ser atualizadas, mas não lidas por um agente ASM e são monitoradas pelo ambiente ou em geral, por outros agentes.

2.4.2 Estados e estados predefinidos

Um estado é obtido pela associação de uma interpretação com todos os nomes de um vocabulário *V* sobre um conjunto ASM, que pode ser infinito.

Uma especificação ASM contém a definição de um estado inicial S_0 e uma regra, *R*, que define as mudanças de estado. A execução de uma especificação é uma seqüência de estados $\langle S_n : n \geq 0 \rangle$, onde um estado S_i é obtido executando a regra R em S_{i-1} (DI IORIO, 2001).

O estado predefinido é um parâmetro inicial, necessário para a execução de uma especificação. Podem ser informados pelo usuário durante a execução da especificação como entrada de dados, ou, fazer parte da descrição da especificação e não requerer a ação do usuário para informar valores iniciais.

2.4.3 Transições e programas ASM

Transições (ou regras) ASM especificam uma atualização sobre estados ASM e representam a maneira básica para expressar comportamento em especificações ASM. Transições são geralmente expressadas como mudanças em alguns endereços de estados em um ASM, em que um endereço representa uma parte do estado global do ASM em execução.

As regras mais simples, conhecidas como regras básicas, são: *atualização*, *bloco* e *condicional* (DI IORIO, 2001).

Uma regra de atualização é da forma:

$$R \equiv f(\overline{x}) := y$$

Esta regra cria, a partir de um estado *S,* um novo estado *S'*, tal que a interpretação do nome de função $f : X^r \rightarrow X$, que, no ponto $\overline{x}$, o seu valor é *y*. Por exemplo, a regra *f(1):=2* determina um novo estado no qual o valor da função *f*, no ponto 1, é 2.

Uma regra condicional da forma:

$$R \equiv \textbf{if } g \textbf{ then } R_1 \textbf{ else } R_2 \textbf{ endif}$$

Tem a seguinte interpretação: se a expressão *g* for verdadeira, então o estado resultante é o resultado da regra R_1, caso contrário, R_2**.**

Uma regra bloco da forma:

$$R \equiv R_1, ..., R_n$$

tem a seguinte semântica: o estado formado por *R* é o resultado da execução de todas as regras R_i em paralelo. Por exemplo, a execução da regra bloco

$$f(1) := 2, f(2) := 4$$

produz um novo estado, no qual o valor da função *f*, no ponto 1 é 2, no ponto 2 é 4.

2.4.4 Agentes ASM

Um dos conceitos formais de ASM é chamada mono-agentes, cuja definição é feita sobre um vocabulário, por seus estados, e um estado especial chamado estado inicial, e seu programa que descreve as regras para transição entre estados.

Um ASM *multi-agente* terá um número finito de agentes ASM, que pode variar de acordo com os estados do ASM. Os agentes ASM geram um domínio ASM especial, que executa agentes ASM.

Cada agente ASM se comporta como especificado em um programa contendo uma coleção estática de programas ASM. O conjunto básico de programas ASM representa um domínio especial do vocabulário ASM, que executa um conjunto básico de programas.

Os agentes executam de forma concorrente seus programas ASM e interagem globalmente compartilhando os endereços dos estados (OBER, 2001).

2.4.5 Exemplo: Fatorial

Di Iorio (2001) apresenta um exemplo de uma especificação ASM para o cálculo do fatorial de um número, conforme apresentado a seguir:

Suponha que o estado inicial S_0 seja dado com o vocabulário Υ = {*f, i*}, onde *i* é um nome de função de zero argumento, interpretado como 0, e *f* é um nome de função unária, interpretado como a função f = $\lambda x(x=0 \rightarrow 1, undef)$. A regra da especificação é o bloco:

$$f(i+1) := (i+1) \times f(i), i := i+1$$

Ao executarmos a regra do estado S_0 no qual *i* é interpretado como 1 e *f* é interpretado como a função:

$$f = \lambda x.(x=0 \rightarrow 1, (x=1, undef))$$

Como é possível verificar na Figura 21, assim que a regra é executada no estado S_1, obtemos um novo estado S_2, no qual *i* é interpretado com 2 e a interpretação de *f*, no ponto 2, passa ser igual a 2. Da mesma forma, obtemos os estados S_3, S_4,...

Estado	*f*(0)	*f*(1)	*f*(2)	*f*(3)	*f*(4)	...	*f*(n)
S_0	1	*undef*	*undef*	*undef*	*undef*	...	*undef*
S_1	1	1	*undef*	*undef*	*undef*	...	*undef*
S_2	1	1	2	*undef*	*undef*	...	*undef*
S_3	1	1	2	6	*undef*	...	*undef*
S_4	1	1	2	6	24	...	*undef*
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
S_n	1	1	2	6	24	...	*n!*

Fonte: (DI IORIO , 2001)

Figura 21 - Interpretação do nome da função f.

Pode-se verificar que no estado $S_k, k \geq i$, a interpretação de *f* é uma função que, aplicada a *i*, retorna *i!*...

2.5 Trabalhos relacionados

Nesta seção, são listados trabalhos presentes na literatura relacionados à proposta deste trabalho.

No trabalho de Park *et al*, (2007), é realizado o mapeamento dos elementos SPEM para o formalismo *DEVS-Hybrid* e são definidas regras de transformação. As regras de transformação são responsáveis pela transformação automática de diagramas de atividades e caso de uso da UML para o formalismo *DEV-Hybrid* para realizar a simulação do modelo. Apesar de utilizar a transformação automática, o trabalho não utiliza a abordagem MDA, que prevê a existência de modelos independentes e específicos de plataforma, como também técnicas de mapeamento. O SPEM é utilizado como referência para a transformação, mas a notação dos diagramas não utiliza o conceito de estereótipos e valores etiquetados (UML, 2008c), sendo necessário suporte de outras ferramentas para a definição dos parâmetros dos modelos. A simulação do modelo é prevista pelo simulador DEVS[2], contudo o trabalho não descreve a estrutura do simulador.

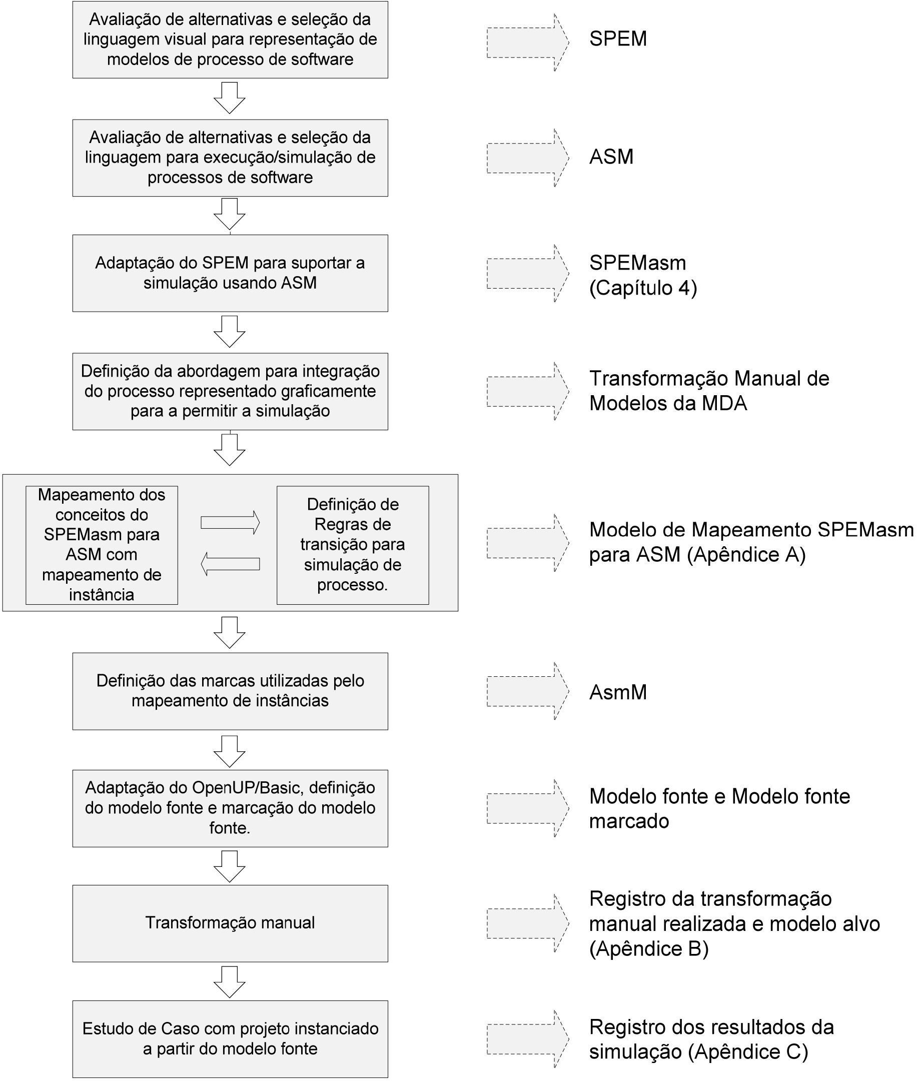
Em Bendraou *et al,* (2007) o SPEM 2.0 (OMG, 2008a) é estendido para o *xSPEM* para permitir a execução de modelos de processo. O mapeamento do xSPEM é feito para redes Petri, cujos conceitos são mapeados para o BPEL[3] *(Business Process Execution Language)* usando regras de transformação definidas na linguagem ATL[4]. A linguagem BPEL é baseada em XML e seu objetivo é permitir a execução de processos de negócio, contudo, o trabalho de Bendraou *et al,* (2007) não realizada a execução de processos, ou a simulação.

---

[2] http://www.acims.arizona.edu/SOFTWARE/software.shtml
[3] http://bpel.xml.org/specifications
[4] http://www.eclipse.org/m2m/atl/

3 METODOLOGIA

Este capítulo apresenta a metodologia utilizada no desenvolvimento deste trabalho, cujo esquema geral é apresentado na Figura 22, onde as setas com linhas pontilhas apontam os produtos de trabalho desta pesquisa.



Fonte: Elaborado pelo autor

Figura 22 – Esquema geral da metodologia para o desenvolvimento da pesquisa.

Foi realizado um levantamento bibliográfico para seleção de uma notação (linguagem visual) para representação de modelos de processo de software. Como um processo de software também é um processo de negócio, linguagens visuais como *Business Process Modeling Notation* (BPMN) (OMG, 2008a) e *Eriksson-Penker Business Extensions* (EPBE) (PENKER, 2000) inicialmente seriam uma opção.No

entanto, o *SPEM* (OMG, 2005) atendeu de forma plena aos requisitos, já que é baseada na UML e fornece diagramas para representação estrutural de processos de software, que possibilitam a apresentação dos elementos do processo, seus atributos e associações, como também a representação comportamental estática, principalmente por meio de diagramas de atividades e estados da UML (OMG, 2008c).

Após a seleção da linguagem visual, para representação do processo em um alto nível de abstração, foi realizada uma avaliação de linguagens que pudessem simular o processo de software modelado. Como primeira opção, a linguagem *XML Process Definition Language* (XPDL) foi avaliada, no entanto, seu objetivo é a representação textual de processo de negócio e a execução do modelo demanda um suporte tecnológico de ferramentas de simulação, que foge ao escopo deste trabalho.

A linguagem *Maude* (MAUDE, 2007) foi avaliada e inicialmente selecionada neste trabalho, contudo, com o andamento da pesquisa, verificou-se a indisponibilidade de referências e principalmente de ferramentas de apoio. As Máquinas de Estado Abstratas surgiram como alternativa, já que permitem a especificação de sistemas de qualquer natureza de maneira seqüencial utilizando formalismo matemático (DI IORIO, 2001), (VALENTE, 1999), (OBER, 2001). O fator determinante para a seleção das máquinas de estado abstratas foi a independência de tecnologia para execução das especificações escritas nesta plataforma.

O próximo passo foi a seleção da linguagem que desse suporte a implementação dos conceitos de máquinas de estado abstratas e para servir de plataforma específica na transformação de modelos. O processo de transformação manual foi guiado para a linguagem *CoreASM* (COREASM, 2007) e um protótipo para a simulação foi produzido. No entanto, as pesquisas bibliográficas revelaram o projeto *Asmeta* (ASMETA, 2007), um conjunto de ferramentas de validação e simulação de especificações ASM. Na pesquisa realizada, o meta-modelo *AsmM* surgiu como alternativa para apoiar a transformação manual, através das marcas utilizada no mapeamento de instâncias pela MDA (CALIARI, 2003).

Dentre as abordagens de transformação de modelos propostas por OMG (2003), a transformação manual foi selecionada, por representar, no entendimento neste trabalho, a primeira etapa antes da transformação automática. Desta forma, o registro das etapas da transformação manual realizado neste trabalho pode ser utilizado em trabalhos futuros como fonte para definição das regras de transformação em ferramentas de transformação automatizada de modelos.

Para que a transformação manual pudesse ser realizada, foi estabelecido o mapeamento entre os conceitos da notação SPEM para o ASM especificado para processos de software neste trabalho. Segundo OMG (2005), o SPEM não tem como foco a execução de modelos de processo, somente a representação estática. Um passo necessário antes do mapeamento foi a definição da máquina de estado abstrata que representa o comportamento do processo em execução.

Em seguida, a transformação manual foi realizada, e cada passo do processo de transformação registrado. A transformação utilizou o mapeamento de instâncias, em que um modelo fonte marcado (modelo independente de plataforma) é transformado para um modelo alvo (modelo específico de plataforma).

O ciclo de vida do *OpenUP/Basic* (KROLL, 2006) foi adaptado para servir de modelo fonte neste trabalho. Com o modelo fonte definido, a marcação foi feita tendo como referência o modelo de marcas que utiliza o meta-modelo *AsmM*.

A plataforma alvo, específica de plataforma (modelo alvo) utiliza a linguagem *AsmetaL* (*ASMETA Language*), contudo é um projeto em andamento, cuja versão disponibilizada (0.9.2 alfa) (ASMETA, 2007) e utilizada neste trabalho não fornece todos os construtores e funcionalidades necessárias para a definição de regras de transição totalmente aderentes ao mapeamento realizado, contudo, pequenas adaptações foram realizadas possibilitando a execução (*enactment*) do processo.

O estudo de caso foi realizado a partir do modelo fonte definido, foi gerado um modelo fonte marcado e criada uma instância para um projeto com prazos e metas definidas.

A execução da máquina de estado abstrata produzida pela transformação manual foi feita no Eclipse (ECLIPSE, 2006), com o plugin Asmee[5] (ASMETA, 2007) que produz um registro (*log)* em formato texto, que foi tratado e é apresentado no Apêndice C.

[5] http://asmeta.sourceforge.net/userdoc/asmee.html

4 *SPEMASM* – UMA EXTENSÃO DO SPEM PARA EXECUÇÃO DE PROCESSOS DE SOFTWARE EM MÁQUINAS DE ESTADO ABSTRATAS

Este capítulo apresenta uma proposta de extensão do meta-modelo SPEM (OMG, 2005) para execução de processos de software em Máquinas de estado abstratas, denominada *SPEMasm*. A extensão pode ser usada como perfil UML e apresenta um conjunto de informações que foram adicionadas ao pacote estrutural e ciclo de vida do SPEM. Além disso, a extensão visa especificamente:

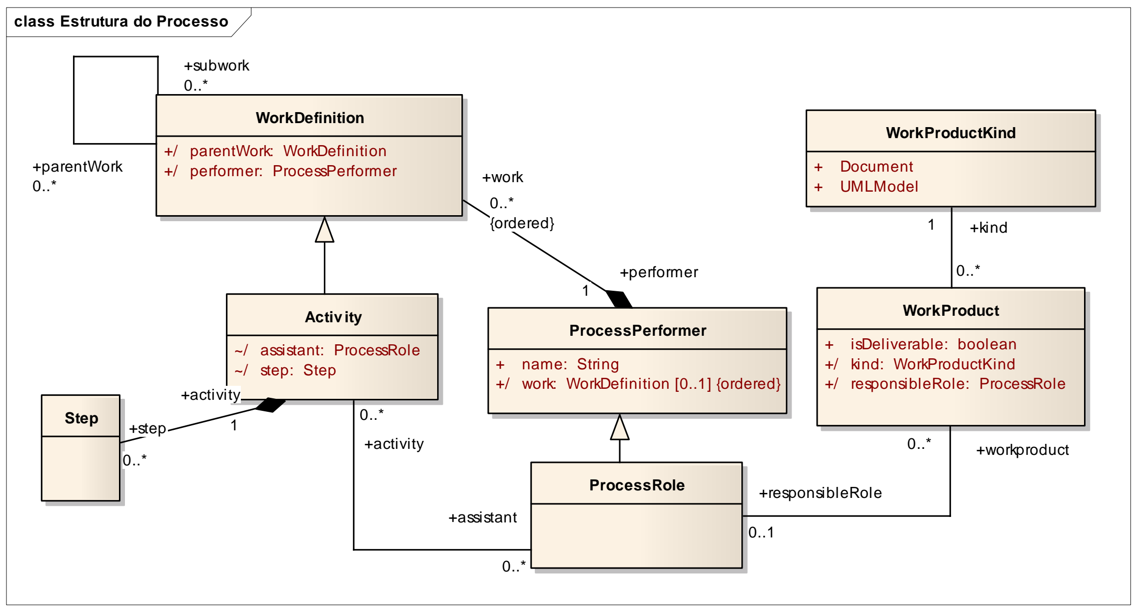
- Adicionar comportamento ao SPEM;
- Permitir que o SPEM seja instanciado para projetos de software, estendendo assim a proposta da OMG em (OMG, 2005);
- Possibilitar sua abstração para Máquinas de estado abstratas, de forma que não seja necessária uma infra-estrutura tecnológica para a execução (*enactment*) do processo.

Conforme apresentado na seção 2.2, o SPEM possui uma estrutura organizada em pacotes com diferentes pontos de vista para processos de software. Contudo, seu escopo não é a execução do processo, sendo assim não se espera que o modelo forneça um conjunto de elementos que possam prontamente ser utilizados em projetos reais, nas empresas desenvolvedoras de software.

Com base no que foi apresentado, o *SPEMasm* é dividido em pacotes, seguindo a abordagem de OMG (2005), que é apresentado a seguir.

4.1 O pacote Estrutural (*ProcessStructure*)

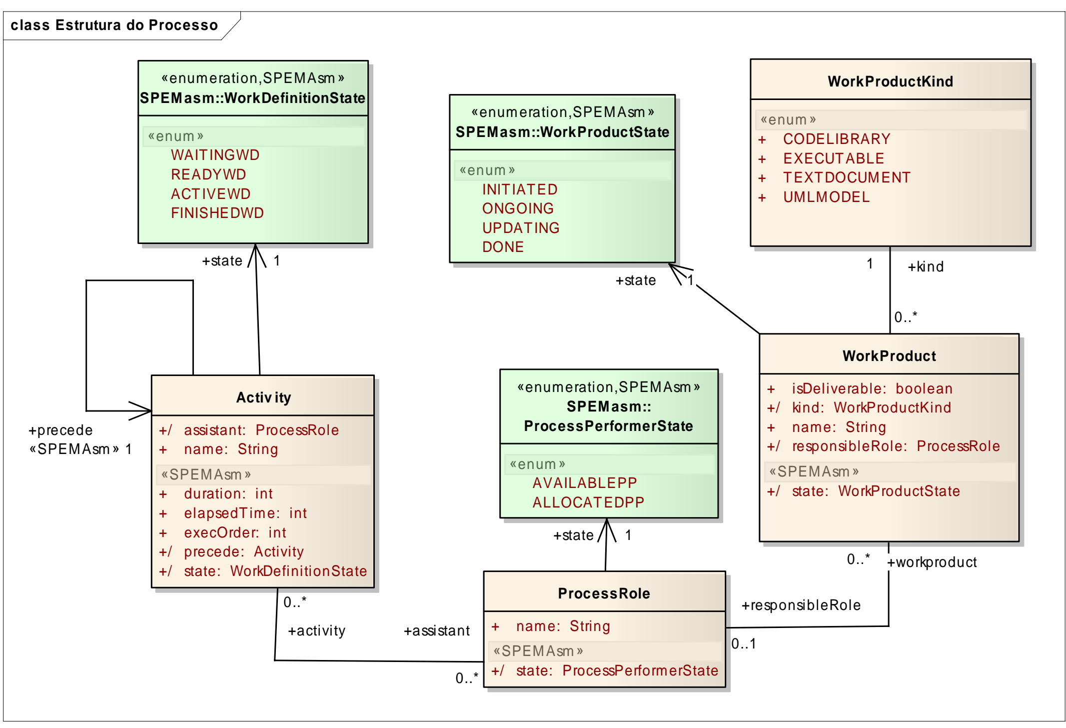
O pacote estrutural proposto no SPEM, versão 1.1, fornece um conjunto genérico de estruturas que permitem a representação de famílias processo de software (OMG, 2005). A composição das estruturas objetiva principalmente mostrar a relação entre elementos do modelo, como também fornece descrições básicas de atributos destes elementos. A Figura 23 exibe o modelo estrutural proposto em (OMG, 2005).



Fonte: (OMG, 2005)

Figura 23 – Pacote Estrutural de Processo

Conforme discutido, o elemento *Activity* apresenta apenas atributos para a descrição de como uma atividade é relacionada aos demais elementos do pacote estrutural. Como a execução, simulação ou descrição comportamental de modelos de processo de software não é foco do SPEM (OMG, 2005), é proposta neste trabalho uma adaptação apresentada na Figura 24.

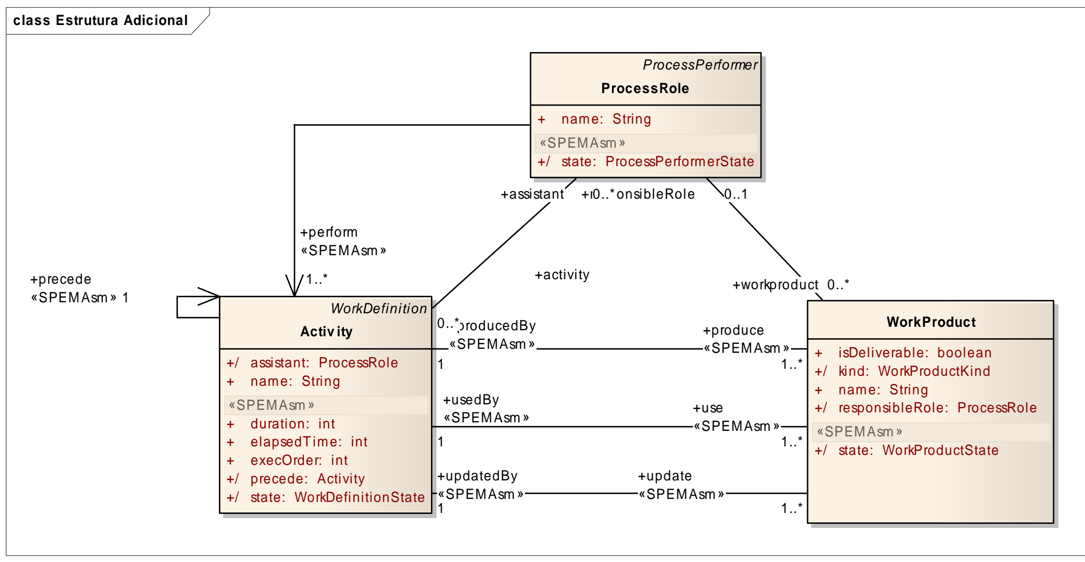


Fonte: Adaptado de (OMG, 2005)

Figura 24 – Pacote Estrutural de Processo do *SPEMasm*

No modelo apresentado, os elementos *ProcessPerformer* e *WorkDefinition* foram

eliminados, pois são destinados a representação de pacotes de trabalho de modelos de processo de software, e não tem relação direta com a execução ou simulação. Em outras palavras, são utilizados apenas pelo modelo estático ou descritivo do processo de software. Associações foram criadas no *ProcessStructure* para dar suporte a execução do ASM proposto neste trabalho e são apresentados na Figura 25.



Fonte: Adaptado de (OMG, 2005)

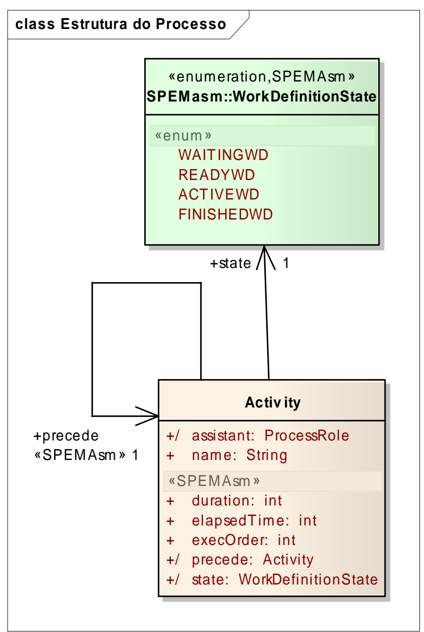
Figura 25 – Associações complementares no Pacote Estrutural do SPEMasm

A associação *perform* permite que o *ProcessRole* seja associado com pelo menos uma atividade que irá executar. As associações *produce, use, update* de *Activity* para *WorkProduct* associam à atividade respectivamente os produtos de trabalho que são produzidos (como saída da atividade), utilizados ou atualizados (entrada) para cada atividade.

O elemento *Step* foi eliminado do SPEM*asm*, pois consideramos para efeito da simulação, que não é necessário detalhar uma atividade (*Activity)* em passos (*steps)*.

4.1.1 Activity

Foram adicionados ao elemento *Activity* os atributos: *duration, elapsedTime, execOrder, precede* e *state*. Tais atributos estão estereotipados no modelo com *<<SPEMasm>>* para facilitar a identificação. A Figura 26 apresenta o elemento *Activity* adaptado neste trabalho.



Fonte: Adaptado de (OMG, 2005)

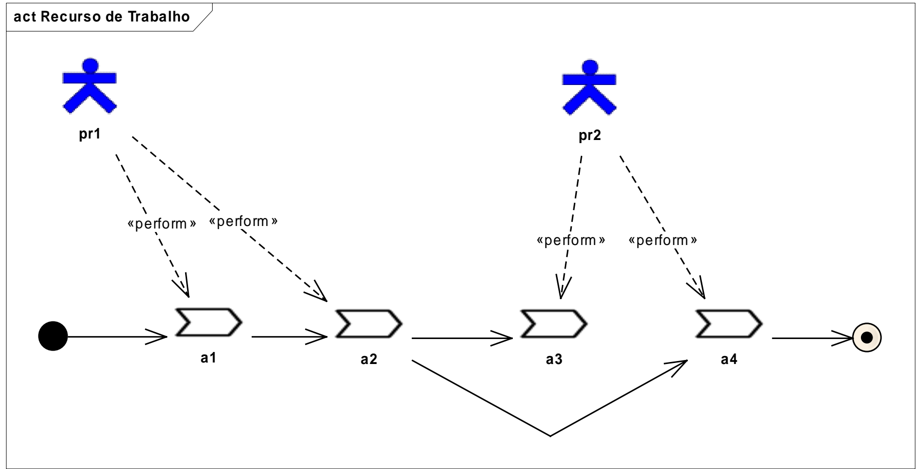
Figura 26 – Elemento Activity adaptado neste trabalho

O propósito do atributo *duration* é permitir que o gerente do processo informe a duração estimada da atividade. A duração deve ser informada em unidades inteiras de tempo, como dias, horas ou semanas. Como o foco não é a execução, valores não discretos podem ser abstraídos e arredondados.

O atributo *elapsedTime* tem como objetivo permitir que a máquina de estados abstrata proposta neste trabalho possa simular o tempo decorrido de uma atividade, assim que ela for iniciada pelo fluxo de trabalho, que é descrito pelos atributos *execOrder* e *precede*.

A ordem de execução das atividades deve ser definida pelo atributo *precede*, sendo assim, uma atividade $a1$, precede $a2$ se: $precede(a2) = a1$. O atributo *execOrder* é definido pela máquina de estado abstrata de acordo com a precedência padrão informada pelo atributo *precede,* que aqui é denominada *precedência por fluxo de atividade*, ou ainda por precedências de *recurso de trabalho* e de *produto de trabalho*.

A *precedência por recurso de trabalho* é avaliada quando, numa rede de atividades, a precedência de fluxo de atividade permite a execução da atividade *a4* em paralelo a atividade *a3*, porém o recurso de trabalho, ou seja, o ator (*ProcessRole*) é o mesmo, conforme apresentado na Figura 27.



Fonte: Elaborado pelo autor

Figura 27 – Precedência de Recurso de Trabalho

No exemplo da Figura 27 a precedência por fluxo de atividade, que é definida pelo modelo, é dada pela seqüência:

precede(a1):= ∅

precede(a2):= a1

precede(a3):= a2

precede(a4):= a2

Pela regra de precedência de fluxo de atividade, a3 e a4 podem ser executadas em paralelo, e a máquina de estado abstrata define valores iniciais para *execOrder:*

execOrder(a1):= 1

execOrder(a2):= 2

execOrder(a3):= 3

execOrder(a4):= 3

Antes que o trabalho das atividades seja iniciado, o ASM avalia pelas regras de atualização, cada recurso de trabalho e quais atividades esse recurso realiza (*perform*), e redefine por meio das regras a ordem de execução, como também a precedência, cujo *execOrder* resulta, no exemplo apresentado na Figura 27, pois a3 e a4 estão alocados para pr2:

execOrder(a1):= 1

execOrder(a2):= 2

execOrder (a3):= 3

execOrder(a4):= 4

E redefine a precedência padrão para:

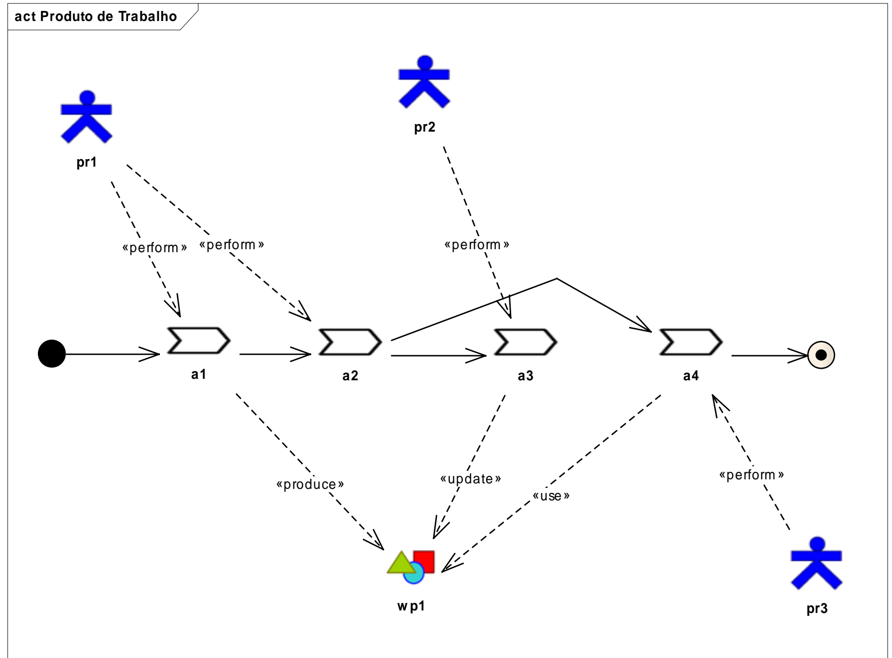
precede(a1):= ∅

precede(a2):= a1

precede(a3):= a2

precede(a4):= a3

A precedência por produto de trabalho também é verificada pela ASM, quando um produto de trabalho produzido por uma atividade *a1* (*produce*), é utilizado (*use*) na atividade *a4*, mas é atualizado (*update*) pela atividade *a3* como é apresentado na Figura 28.



Fonte: Elaborado pelo autor

Figura 28 – Precedência de Produto de Trabalho

No exemplo apresentado na Figura 28, a precedência inicial é a mesma apresentada no caso de recursos de trabalho.

Na precedência de produtos de trabalho, o ASM avalia o uso (*use*) e a atualização (*update*) dos produtos de trabalho. Neste caso, o *execOrder* é modificado para 4, já que a atividade *a3* atualiza um produto de trabalho que é utilizado pela atividade *a4,* conforme apresentado a seguir:

execOrder(a1):= 1

execOrder(a2):= 2

execOrder (a3):= 3

execOrder(a4):= 4

E redefine a precedência para:

precede(a1):= ∅

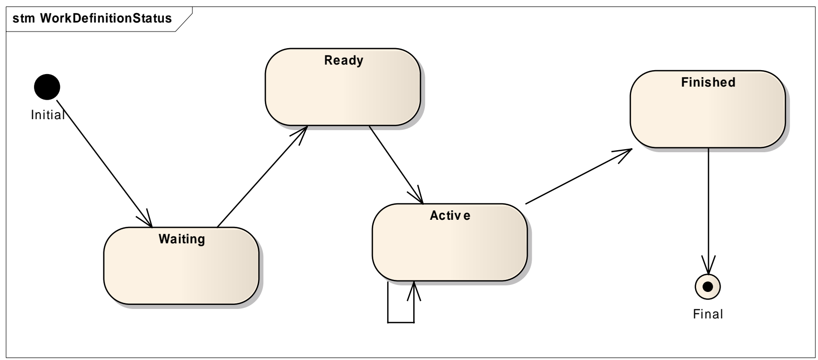
precede(a2):= a1

precede(a3):= a2

precede(a4):= a3

No ASM proposto, apresentado adiante no Capítulo 5, as regras que verificam as precedências de produtos de trabalho e recursos de trabalho são executadas em paralelo pela máquina, portanto não são mutuamente exclusivas.

Finalmente, o atributo *state* de *Activity* é utilizado para controlar o ciclo de vida de definições de trabalho (*WorkDefinition*), durante a simulação do processo pela ASM. Os estados possíveis para uma definição de trabalho são apresentados no elemento *WorkDefinitionState* e a transição de estados para a atividade é demonstrada na Figura 29.



Fonte: Adaptado de (REIS, 2003)

Figura 29 – Estados possíveis para uma atividade.

Conforme adaptação realizada na transição proposta por Reis (2003), a atividade pode assumir os seguintes estados, ou seja, o atributo *state* pode assumir os seguintes valores:

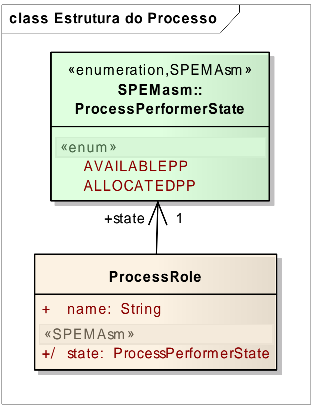
- ***Waiting***: aguardando que o agente (*ProcessRole*) inicie o trabalho;
- ***Ready***: pronta para iniciar o trabalho, pois foi colocada na fila de execução pelo agente, considerando a precedência.
- ***Active***: a atividade está em execução, e permanece, até que o tempo decorrido (*elapsedTime*) seja igual ao tempo estimado(*duration*) para a atividade.
- ***Finished***: o trabalho da atividade foi finalizado, o recurso de trabalho (*ProcessRole*) foi desalocado e os produtos de trabalho atualizados ou produzidos nesta atividade foram finalizados.

Neste trabalho, optou-se por adaptar-se a transição de estados proposta por Reis (2003) e eventuais Falhas, Cancelamentos e Pausas não são consideradas para as atividades. Contudo, a abordagem orientada a objetos utilizada pelo SPEM e portanto, pelo SPEMasm, permite que novos estados e transições sejam incorporados.

A transição de estados não foi descrita, pois é realizada no Capítulo 5, quando a máquina de estado abstrata (ASM) proposta neste trabalho é apresentada.

4.1.2 ProcessRole

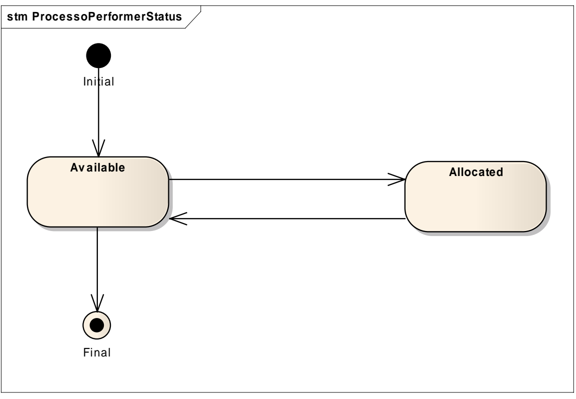
A adaptação realizada no elemento *ProcessRole* permite que seu estado ao longo da simulação de um processo seja atualizado pelo ASM. A Figura 30 apresenta o *ProcessRole* adaptado neste trabalho.



Fonte: Adaptado de (OMG, 2005)

Figura 30 – Elemento *ProcessRole* adaptado neste trabalho

Um ator que desempenha um papel no processo precisa ter seu estado definido, de acordo com sua alocação em cada atividade. Sendo assim, enquanto uma atividade cujo ator (*ProcessRole*) estiver em execução (*Active*), por convenção adotada neste trabalho, o ator não pode executar ou assistir outra atividade. Os estados para um recurso de trabalho foram adaptados de Reis (2003), e são apresentados na Figura 31.



Fonte: Adaptado de (REIS, 2003)

Figura 31 – Estados possíveis para um *ProcessRole*.

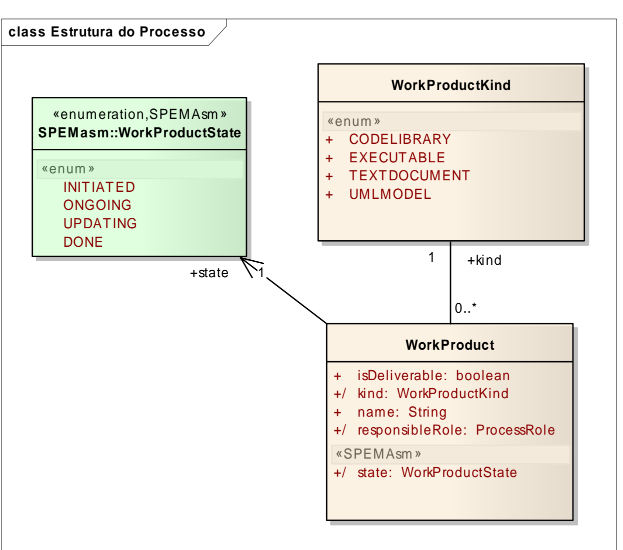
Um ator pode assumir os seguintes estados:

- ***Available***: o ator está disponível para realizar uma atividade;
- ***Allocated***: o ator está alocado em uma atividade e não pode realizar outra atividade.

No diagrama apresentado na Figura 30, o estado inicial do ator é disponível (Available) e logo que ele inicia o trabalho em uma atividade, é alocado (Allocated), voltando a ficar disponível assim que o trabalho é finalizado.

4.1.3 WorkProduct

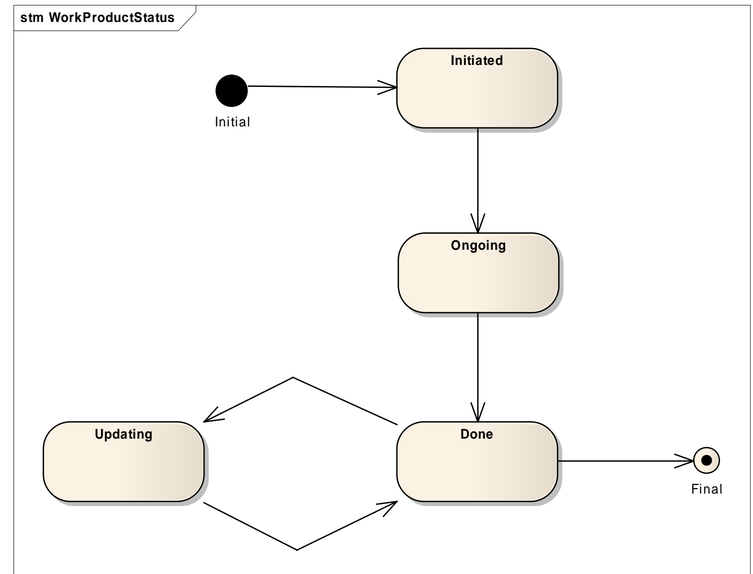
O elemento *WorkProduct* foi adaptado para que os estados pelos quais um produto de trabalho passa durante a instanciação de um processo de desenvolvimento de software sejam considerados. Figura 32 apresenta o *WorkProduct* adaptado neste trabalho.



Fonte: Adaptado de (OMG, 2005)

Figura 32 – Elemento WorkProduct

O atributo *state* foi adicionado e seu ciclo de vida é apresentado na Figura 33.



Fonte: Adaptado de (REIS, 2003)

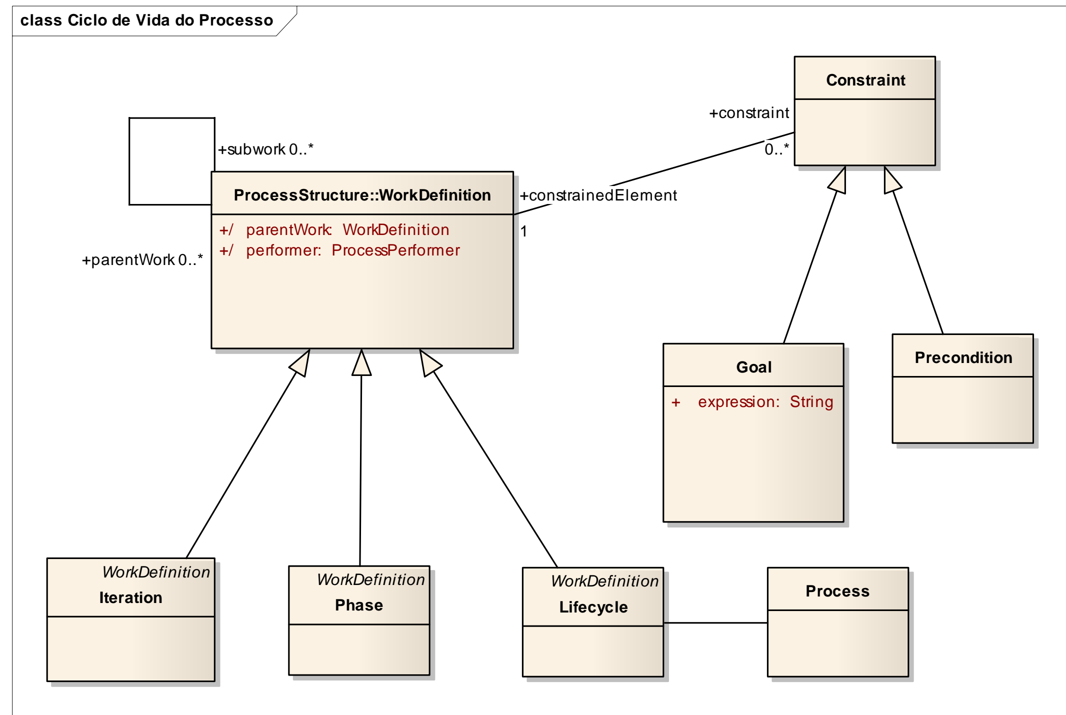
Figura 33 – Estados possíveis para um WorkProduct.

Um produto de trabalho pode assumir os seguintes estados:

- **Initiated**: o produto de trabalho é iniciado assim que a ASM entra em execução;
- **Ongoing**: assim que o produto de trabalho começa a ser produzido ou atualizado por uma atividade, ele está em andamento, ou em processo de produção;
- **Done**: o produto de trabalho é concluído assim que a atividade termina de produzí-lo ou atualizá-lo;
- **Updating**: Depois que um produto de trabalho é produzido ou atualizado por outras atividades, uma atividade pode atualizá-lo.

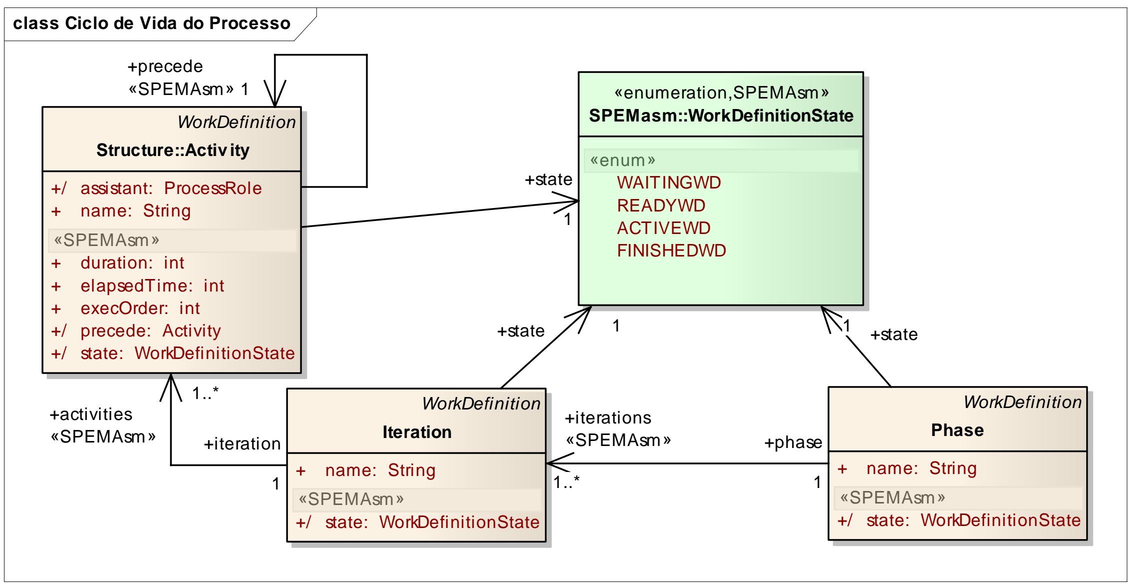
4.2 O pacote Ciclo de Vida (*ProcessLifeCycle*)

O pacote ciclo de vida, apresentado em OMG (2005) também é estendido neste trabalho, conforme apresentado na Figura 35. A proposta original, da versão 1.1 do SPEM é apresentada na Figura 34.



Fonte: (OMG, 2005)

Figura 34 – Pacote Ciclo de Vida do SPEMasm



Fonte: Adaptado de (OMG, 2005)

Figura 35 – Pacote Ciclo de Vida

Na adaptação realizada neste trabalho os elementos *LifeCycle* e *Process* foram ignorados, já que o ASM proposto define o ciclo de vida baseando-se nas precedências e tais elementos são meramente descritivos.

As associações entre os elementos *Iteration*, *Phase* e *Activity* foram redefinidas, considerando que uma iteração tem *n* atividades e uma fase, *n* iterações, de acordo com

o que é apresentado na seção 2.2.1 deste trabalho (KRUTHEN, 2003) e (KROLL, 2006).

Os elementos *Iteration* e *Phase* são especializações de *WorkDefinition* e portanto seguem o mesmo ciclo de vida, inclusive das atividades (*Activity*). Contudo, a semântica das transições de estados é diferente, e é discutida no Capítulo 5.

4.3 Considerações Finais

O meta-modelo *SPEMasm* apresentado neste capítulo foi definido buscando atender a seleção da linguagem para definição de modelos de processos em um nível mais alto de abstração. Foi necessário adaptar o meta-modelo SPEM (OMG, 2005) já que este não apresenta as estruturas necessárias para suportar a simulação (*process enactment).*

A adaptação inclui informações ao meta-modelo selecionado e sugere um ciclo de vida para cada elemento estrutural de modo a fornecer meios para sua simulação. O capítulo seguinte apresenta o mapeamento dos conceitos do meta-modelo definido neste capítulo para máquinas de estado abstratas.

5 MAPEAMENTO DE PROCESSOS DE SOFTWARE EM MÁQUINAS DE ESTADO ABSTRATAS

Neste capítulo, são apresentados os resultados do mapeamento realizado entre os conceitos de processos de software e máquinas de estado abstratas. Entende-se por mapeamento, a correspondência realizada entre elementos de um conjunto *A* para elementos do conjunto *B*, de modo que o elemento mapeado no conjunto *A* possa ser transformado no elemento do conjunto *B* em diferentes níveis de abstração.

A abstração realizada é que o processo de software é um vocabulário, cujo conjunto de domínios representam os elementos do processo de software, neste trabalho utilizando-se o *SPEMasm* como referência. Os atributos dos elementos do *SPEMasm* são mapeados para funções estáticas e dinâmicas e o conjunto destes elementos formam o vocabulário ASM para processos de software. A noção de processo surge a partir da definição dos domínios, que conforme apresentado anteriormente, representam os elementos do *SPEMasm,* e possuem funções associadas, cujos estados são modificados pelas regras que definem a semântica comportamental para um modelo processo de software, e permitem a instanciação de projetos de software no vocabulário descrito.

Cavarra *et al,* (2004) propõe o mapeamento de modelos UML para ASM, contudo neste trabalho a proposta estende a proposta de Cavarra *et al,* (2004), pois adiciona a semântica de processos de software, além dos elementos puramente estruturais da UML.

Ainda de acordo com a abordagem de Cavarra (2004) e utilizando-se como referência o modelo MOF apresentado na seção 2.2, os elementos do SPEM são representações de classes da UML, e suas propriedades, são atributos da UML. Park *et al,* (2007) propõem o mapeamento do SPEM para um formalismo denominado DEVS. A proposta de mapeamento deste trabalho é uma adaptação das propostas apresentadas e é descrita na Tabela 2.

Tabela 2 – Mapeamento SPEM para ASM (Modelo de Mapeamento)

Elemento	*SPEMasm*		ASM
Estrutura	*ProcessStructure*	*WorkDefitinitionState, WorkProductState, ProcessPerformState,*	Domínio estático enumerado pré-definido

		WorkProductKind	
		Activity, WorkProduct	Domínios abstratos
		ProcessRole	Agente
		Atributos de *Activity*, *WorkProduct* e *ProcessRole*	Funções estáticas e dinâmicas
		Associações	Funções estáticas
		Instâncias	Funções estáticas
	ProcessLifeCycle	*Iteration, Phase*	Domínios abstratos
		Precondition e Goal	Funções dinâmicas
		Associações	Funções estáticas
Comportamento	*Diagramas de Estados*	*WorkDefitinitionState* para *Activity, Phase e Iteration,*	Regras de Transição
		WorkProductState, ProcessPerformState,	Regras de Transição
		Restrições	Axiomas (ou Predicados)

Adaptado de (PARK *et al*, 2007)

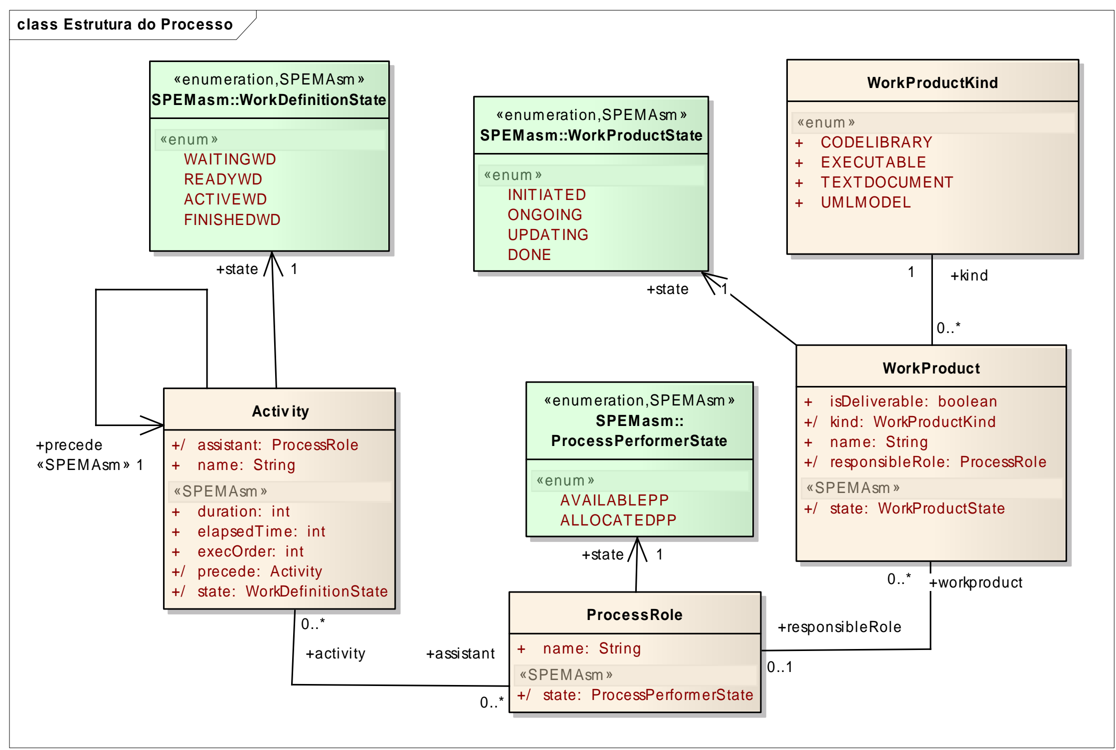
O modelo de mapeamento proposto neste capítulo é usado adiante para definir as marcas para o estudo de caso, realizado utilizando a abordagem MDA, apresentada na seção 2.3.1.

Os elementos dos pacotes apresentados no Capítulo 3 foram mapeados para ASM e serão apresentados a seguir.

5.1 Mapeamento dos elementos estruturais

Nesta seção, é apresentado o mapeamento realizado dos elementos estruturais do processo, de modo que seja possível demonstrar a representação dos elementos do *SPEMasm* para domínios e funções do ASM.

O mapeamento realizado para os elementos do *ProcessStrutucture* do *SPEMasm* é apresentado na Figura 36.



Fonte: (OMG, 2005)

Figura 36 – Pacote Estrutural de Processo do SPEMasm

O elemento *Activity* é mapeado como um domínio abstrato em ASM, pois não há restrições sobre ele, ou seja, sua estrutura e as atualizações sobre ele seguirão uma semântica definida para processos de desenvolvimento de software, apresentado como uma máquina de estado abstrata proposta neste trabalho. O elemento mapeado para domínio segue:

abstract domain *ACTIVITY*

Cada domínio mapeado do *SPEMasm*, é instanciado por meio de funções estáticas, ou simplesmente instâncias. Tais instâncias são utilizadas para endereçar as funções mapeadas dos atributos, que são processadas durante a computação da máquina. Por exemplo, a partir do domínio *ACTIVITY*, podem ser criadas *n* instâncias, cujas funções dinâmicas *State, ElapsedTime e ExecOrder* passam por mudanças de estados durante a simulação do ASM.

No *SPEMasm*, o elemento *Activity* possui atributos que os descrevem e associações com outros elementos para a composição de uma definição de trabalho. Os atributos apresentados no capítulo 3 foram mapeados em forma de funções ASM tanto dinâmicas quanto estáticas, e que são apresentadas e discutidas a seguir:

(a) **dynamic** *State: ACTIVITY* $\rightarrow$ *WORKDEFINITIONSTATE*

(b) **static** *Name: ACTIVITY* $\rightarrow$ *STRING*

(c) **static** *Duration: ACTIVITY* $\rightarrow$ *INTEGER*

(d) **dynamic** *ElapsedTime: ACTIVITY* $\rightarrow$ *INTEGER*

(e) **dynamic** *Precede: ACTIVITY* $\rightarrow$ *ACTIVITY*

(f) **dynamic** *ExecOrder: ACTIVITY* $\rightarrow$ *INTEGER*

As funções *b* e *c* são estáticas, pois não sofrem alteração durante a computação do ASM. As demais funções são alteradas de acordo com a semântica definida no Capítulo 3 e que serão discutidas adiante. A função *a* controla o estado de uma atividade, ou seja, a partir do *state* a atividade entra em computação no ASM e todo seu ciclo de vida é controlado com base nesta função, desde o momento em que a atividade está pronta para ser iniciada, até sua finalização. Seu mapeamento é realizado utilizando-se um domínio *enumerado*, que permite a representação de valores finitos para cada um dos estados possíveis para uma definição de trabalho (*WorkDefinition*), conforme apresentado a seguir:

enum domain *WORKDEFINITIONSTATE = { WAITINGWD | READYWD | ACTIVEWD | FINISHEDWD }*

Cada um dos valores apresentados no mapeamento do estado de uma definição de trabalho (*WorkDefinitionState*) tem um comportamento definido, apresentado na seção 4.2.1.

O elemento *ProcessRole* é mapeado com um agente, que efetivamente inicia o trabalho na ASM especificada. O mapeamento para o ASM é apresentado a seguir:

domain *PROCESSROLE* $\subseteq$ *AGENT*

Por ser um agente, para cada instância de um *ProcessRole* existente no vocabulário do ASM, é iniciada uma regra que avalia as precedências das atividades em espera (*WAITING*) e caso a condição seja verdadeira para início da atividade, ela entra em estado (*READY*). Em outras palavras, cada ator de um projeto de software, instanciado a partir de um modelo de processo de software, como um Analista de

Sistemas, um Programador, ou Arquiteto de Software é representado neste trabalho como um agente, cuja semântica é especificada adiante.

Os atributos do elemento *ProcessRole* são mapeados em funções do ASM, como segue:

(a) **dynamic** *State: PROCESSROLE → PROCESSPERFORMERSTATE*

(b) **static** *Name: PROCESSROLE → STRING*

(c) **static** *Perform***:** *PROCESSROLE → ACTIVITY–***sequence**

A função *b* é estática e não sofre modificação, já que define o nome de um ator do processo.

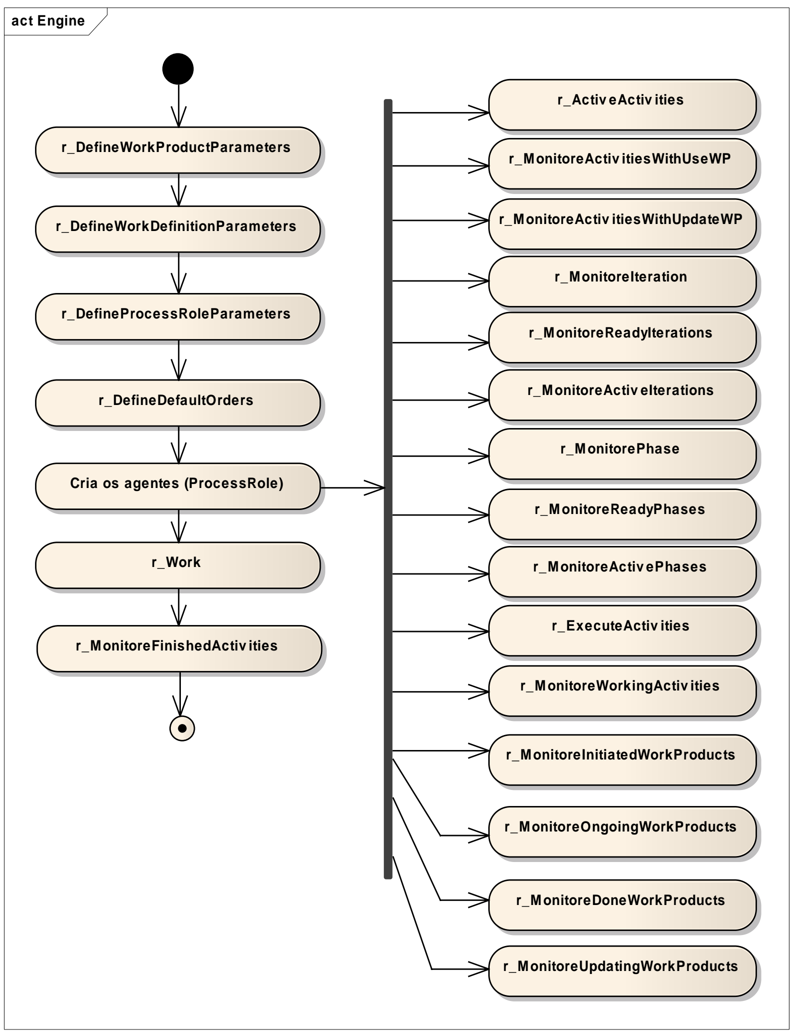
A função *a* controla o estado de um ator do processo (*ProcessRole)* após o início da computação da ASM. A função *c* é estática e deve ser definida no início da computação do ASM como um conjunto de atividades a serem executadas (realizadas) por um *ProcessRole*, ou seja, as atividades que determinado ator realiza durante um projeto de software.

Os demais domínios e funções estruturais são apresentados no Apêndice A e não são discutidas nesta seção. O pacote *ProcessLifeCycle* também faz parte do mapeamento apresentado no apêndice. A seguir é apresentado o mapeamento do comportamento da ASM especificada.

5.2 Mapeamento dos elementos de comportamento

O comportamento do processo é simulado utilizando-se regras do ASM. Cada estado possível para cada domínio é representado pelos domínios estáticos enumerados, associados aos domínios abstratos mapeados para os pacotes *ProcessStructure* e *ProcessLifeCycle* do *SPEMasm*.

Um ambiente de execução de processos deve ser capaz de simular a execução de atividades, que são utilizadas como entrada e produzem produtos de trabalho, ou artefatos. Para que o trabalho da atividade possa ser realizado, é necessário um ator que inicie o trabalho de acordo com o fluxo de execução definido pelo modelo de ciclo de vida do processo de software. Todo o comportamento modelado pelas regras especificadas no ASM proposto neste trabalho é baseado neste cenário de execução do processo. O ciclo de vida do ASM proposto neste trabalho é apresentado na Figura 37.



Fonte: Elaborado pelo autor

Figura 37 – Ciclo de Vida do ASM proposto.

A execução da especificação inicia-se seqüencialmente com as regras que definem os parâmetros iniciais, que são informações da instância do processo de desenvolvimento de software. Logo após, para cada um dos atores que desempenham um papel no processo, é criado um programa ASM que inicia as regras *Work* e *MonitoreFinishedActivities*. Paralelamente todas as regras definidas após a criação dos agentes entram em execução, até que todos os estados definidos em cada uma das regras sejam endereçados, ou seja, alcançados pela computação do ASM. O comportamento global do ASM é apresentado no Quadro 1.

```
Main ≡
        DefineWorkProductParameters
        DefineWorkDefinitionParameters
        DefineProcessRoleParameters
        DefineDefaultOrders

        ∀ pr ∈ PROCESSROLE:
              Program(pr)

      par
              ActiveActivities
              MonitoreActivitiesWithUseWP
              MonitoreActivitiesWithUpdateWP
              MonitoreIteration
              MonitoreReadyIterations
              MonitoreActiveIterations
              MonitorePhase
              MonitoreReadyPhases
              MonitoreActivePhases
              ExecuteActivities
              MonitoreWorkingActivities
              MonitoreInitiatedWorkProducts
              MonitoreOngoingWorkProducts
              MonitoreDoneWorkProducts
              MonitoreUpdatingWorkProducts
      endpar
```

Fonte: Elaborado pelo autor

Quadro 1 – Regras de transição especificadas no ASM.

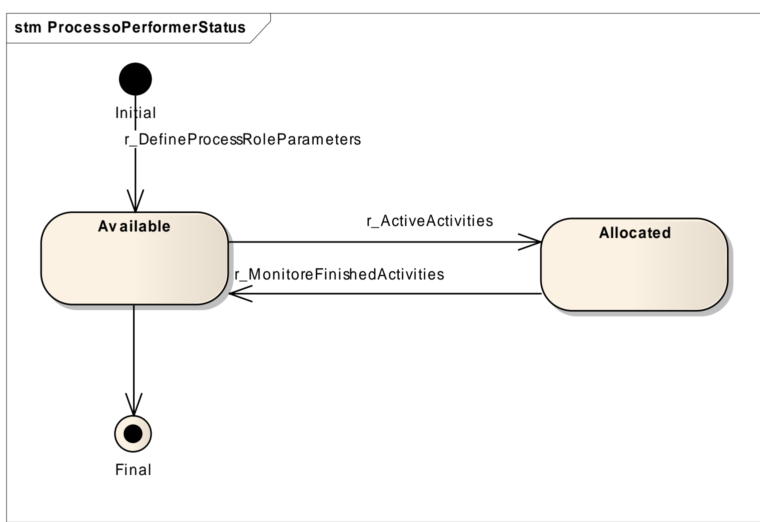
Cada regra apresentada no Quadro 2 é condicionada para iniciar por estados, definidos segundo um ciclo de vida apresentados na seção a seguir.

5.2.1 O Ciclo de Vida do Modelo ASM

Conforme apresentado no Quadro 1, o ASM inicia uma série de regras, que monitoram os estados de todas as instâncias dos domínios declarados na especificação

ASM. As regras iniciadas pelo prefixo "*Define*" inicializam os estados iniciais das funções dinâmicas propostas de acordo com o *SPEMasm*.

O ciclo de vida do *ProcessRole* é apresentado na Figura 38.



Fonte: Elaborado pelo autor

Figura 38 – Ciclo de Vida da *ProcessRole* mapeado no ASM.

O estado inicial do *ProcessRole* é definido pela regra *DefineProcessRoleParameters,* apresentada no Quadro *2*.

DefineProcessRoleParameters ≡ ∀ *pr* ∈ *PROCESSROLE* | *IsUndef(Name(pr))* : *Name(pr)*:= value *Perform(pr)*:= *ACTIVITY*-sequence *State(pr)*:= *AVAILABLEPP*

Fonte: Elaborado pelo autor

Quadro 2 – Regra *DefineProcessRoleParameters*

Para cada instância de *ProcessRole*, é atribuído um nome, as atividades que ele executa e o estado inicial *AVAILABLEPP*. O Quadro 2 apresenta a forma geral da regra. Contudo, os valores para as funções *Name* e *Perform* serão atribuídos para cada instância do modelo de processo de software que se deseja simular, ou seja, devem ser informados manualmente, ou usando a técnica de transformação de modelos, discutida na seção 2.2.2.2 e que é apresentada com o estudo de caso no próximo capítulo. O mesmo vale para toda regra com o prefixo "*Define*", ou seja, seus valores iniciais são informados de acordo com a instância do modelo de processo de software para um projeto específico.

Após a computação da regra *DefineProcessRoleParameters* cada instância de *ProcessRole* está disponível para iniciar o trabalho nas atividades (*state(pr)* = *AVAILABLEPP*). O *ProcessRole* é mapeado como um agente, o que significa que ele

inicia a computação do ASM, após a execução de todas as regras com o prefixo "*Define*". Um programa ASM é criado para cada uma das instâncias, conforme regra especificada no Quadro 1:

$\forall$ *pr* $\in$ *PROCESSROLE*:
  Program(pr)

Quando um Programa ASM é iniciado para um agente, são executadas regras, conforme declaração a seguir:

agent PROCESSROLE :
  Work

Um agente inicia o processo, desde que existam atividades com estado em espera e que o agente esteja disponível. Caso a atividade não seja precedida por outra atividade, seu estado é definido como pronto (*READYWD*) e sua execução é iniciada. Caso haja alguma precedência a atividade em questão só é iniciada se a atividade que a precede tiver sido finalizada. O Quadro 3 apresenta a regra *Work*.

```
Work ≡
    ∀ a ∈ Activity(Self) | State(a) = WAITINGWD :
        if State(a) = AVAILABLEPP then
            if IsUndef(Precede(a)) then
                State(a) := READYWD
            else State(Precede(a)) = FINISHEDWD then
                State(a) := READYWD
            endif
        endif
```

Fonte: Elaborado pelo autor

Quadro 3 – Regra Work para agentes *ProcessRole*

Com a mudança do estado de uma atividade realizada por um agente *ProcessRole* para *READYWD*, a regra *ActiveActivities* encontra o estado que é necessário para que ela entre em execução.

```
ActiveActivities ≡
    ∀ a ∈ ACTIVITY | State(a) = READYWD :
        State(FIRST(sequence (Assistant(a)))) := ALLOCATEDPP
        State(a) := ACTIVEWD
```

Fonte: Elaborado pelo autor

Quadro 4 – Regra *ActiveActivities* para atividades

Todas as atividades que entram no estado *READYWD* passam para o estado *ACTIVEWD* e o ator que executa a atividade fica alocado (*ALLOCATEDPP)* para realizar a atividade em questão.

Quando uma atividade está no estado *ACTIVEWD* e o tempo decorrido (*ElapsedTime*) é menor que a duração estimada, seu *ElapsedTime* é acrescido de 1 unidade de tempo, conforme Quadro 5.

ExecuteActivities ≡ $\forall$ *a* $\in$ *ACTIVITY* | *State(a)* = *ACTIVEWD* $\wedge$ *ElapsedTime(a)* < *Duration(a)* : *ElapsedTime(a)* := *ElapsedTime(a)* + *1* *totalElapsedTime* := *totalElapsedTime* + *1*

Fonte: Elaborado pelo autor

Quadro 5 – Regra ExecuteActivities para atividades

Quando o estado de uma atividade é ativo e o tempo decorrido é igual a duração estimada, a atividade é considerada finalizada.

MonitoreWorkingActivities ≡ $\forall$ *a* $\in$ *ACTIVITY* | *State(a)* = *ACTIVEWD* $\wedge$ *ElapsedTime(a)* = *Duration(a)* : *State(a)* := *FINISHEDWD*

Fonte: Elaborado pelo autor

Quadro 6 – Regra MonitoreWorkingActivities para atividades

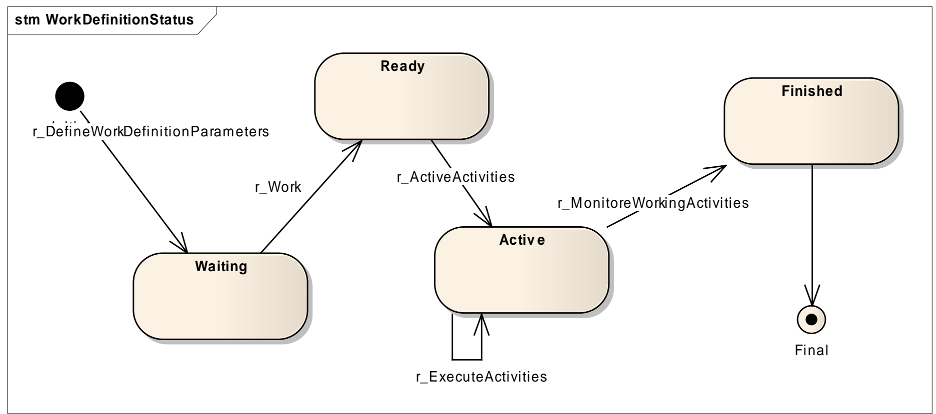
Finalmente, quando as atividades que um ator realiza são finalizadas, o agente ou ator, fica disponível (*AVAILABLEPP*) para trabalhar em outras atividades, completando seu ciclo de vida.

MonitoreFinishedActivities ≡ $\forall$ *a* $\in$ *Activity(Self)* | *State(a)* = *FINISHEDWD* : *State(Self)* := *AVAILABLEPP*

Fonte: Elaborado pelo autor

Quadro 7 – Regra MonitoreFinishedActivities para atividades

O ciclo de vida dos valores de *ACTIVITY,* é resumido na Figura 39.



Fonte: Elaborado pelo autor

Figura 39 – Ciclo de Vida da Activity no ASM.

Conforme foi discutido, regras com prefixo "*Define*" atribuem estados iniciais às funções. A semântica da regra *DefineWorkDefinitionParameters* é apresentada no Quadro 8.

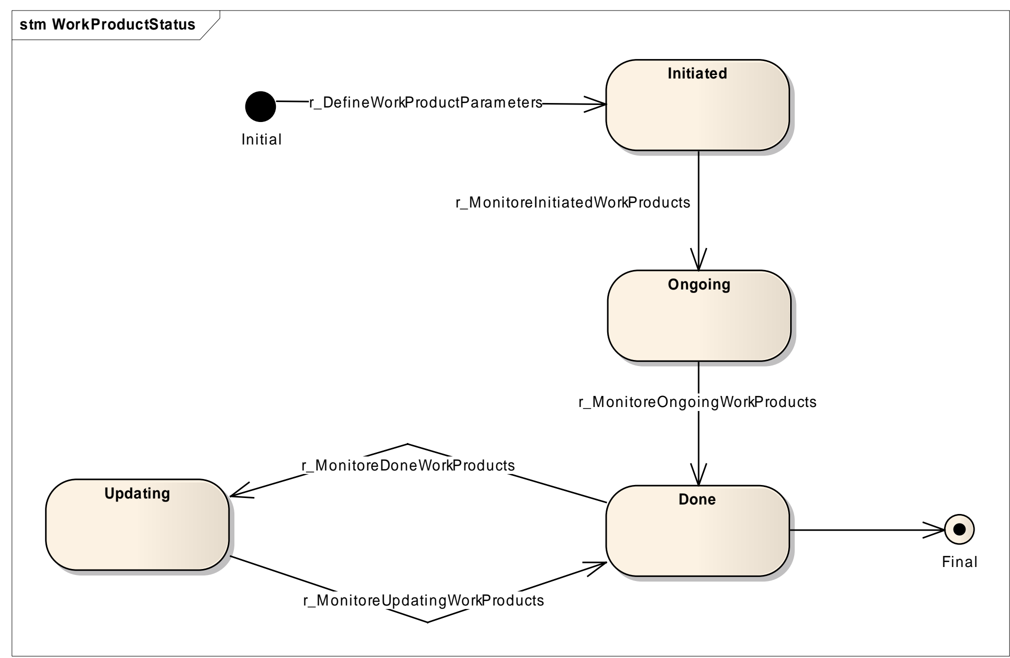
DefineWorkDefinitionParameters ≡
$\forall$ *a* $\in$ *ACTIVITY* | *IsUndef(Name(pr))* :
Name(a):= value1
Duration(a):= value2
State(a):= WAITINGWD (a)
ElapsedTime(a):= 0
ExecOrder(a1):= 1 (b)
Precede(a):= value3
Produce(a):= WORKPRODUCT-sequence (c)
Update(a):= WORKPRODUCT-sequence (d)
Use(a):= WORKPRODUCT-sequence (e)
Name(it):= value4
Activities(it):= ACTIVITY-sequence
Name(ph):= value5
Iterations(ph):= ITERATION-sequence (f)
Precondition(it) := (WORKPRODUCT $\times$ *WORKPRODUCTSTATE)*-sequence (g)

Fonte: Elaborado pelo autor

Quadro 8 – Regra DefineWorkDefinitionParameters para WorkDefinition

Os valores $value_n$ devem ser informados baseando-se nas informações da instância do modelo de processo que estiver sendo realizada para um projeto específico. Os estados iniciais para as funções nas linhas *a, b, f* são obrigatórios, já os estados *c, d, e* são opcionais devem ser informados caso a atividade *a* produza, atualize ou utilize um ou mais produtos de trabalho. O estado *g* deve ser informado caso haja uma precondição para que a iteração inicie.

Com a computação das regras sobre as atividades, o estado dos produtos de trabalho também é alterado conforme apresentado na Figura 40.



Fonte: Elaborado pelo autor

Figura 40 – Ciclo de Vida de um WorkProduct do SPEMasm

Os estados iniciais de cada produto de trabalho especificado no ASM são definidos pela regra *DefineWorkProductParameters,* apresentada no Quadro 9.

DefineWorkProductParameters ≡ ∀ *w* ∈ *WORKPRODUCT* | *IsUndef(Name(pr))* : *Name(pr):=* value1 *State(w):= INITIATED* *IsDeliverable(w):=* value2 *Kind(w):=* value3

Fonte: Elaborado pelo autor

Quadro 9 – Regra *DefineWorkProductParameters*

Os estados *value1* e *value3* devem ser obrigatoriamente informados, neste último caso, usando um elemento do domínio pré-definido *WORKPRODUCTKIND.* O estado *value2* deve ser informado como *true* ou *false,* de acordo com cada produto de trabalho.

Dado um produto de trabalho com estado iniciado e que seja produzido por uma atividade cujo estado esteja ativo, ou seja, em execução, seu estado é definido com em andamento, conforme Quadro 10.

MonitoreInitiatedWorkProducts ≡ ∀ *w* ∈ *WORKPRODUCT* | *State(w) = INITIATED* ∧ *isNotEmpty(ProducedBy(w))* : if *State(FIRST(*sequence*(ProducedBy(w)))= ACTIVEWD* then *State(w) := ONGOING*

endif

Fonte: Elaborado pelo autor

Quadro 10 – Regra MonitoreInitiatedWorkProducts

Todo produto de trabalho que esteja sendo produzido por uma atividade, é concluído quando a atividade é finalizada, de acordo com a regra apresentada no Quadro 11.

```
MonitoreOngoingWorkProducts ≡
∀ w ∈ WORKPRODUCT | State(w) = ONGOING ∧ isNotEmpty(ProducedBy(w)) :
        if State(FIRST(sequence(ProducedBy(w)))= FINISHEDWD then
            State(w) := DONE
        Endif
```

Fonte: Elaborado pelo autor

Quadro 11 – Regra MonitoreOngoingWorkProducts

Todo produto de trabalho que tenha sido finalizado e que seja atualizado por uma atividade que esteja ativa, passa a ser atualizado, como estabelece a regra descrita no Quadro 12.

```
MonitoreDoneWorkProducts ≡
    ∀ w ∈ WORKPRODUCT | State(w) = DONE ∧ isNotEmpty(UpdatedBy(w)) :
        if State(FIRST(sequence(UpdatedBy (w)))= ACTIVEWD  then
                                State(w) := UPDATING
        Endif
```

Fonte: Elaborado pelo autor

Quadro 12 – Regra MonitoreDoneWorkProducts

Todo produto de trabalho que esteja sendo atualizado por uma atividade, é concluído quando a atividade é finalizada, de acordo com a regra estabelecida no Quadro 13.

```
MonitoreUpdatingWorkProducts ≡
∀ w ∈ WORKPRODUCT | State(w) = UPDATING ∧ isNotEmpty(UpdatedBy (w)) :
        if State(FIRST(sequence(UpdatedBy (w)))= FINISHEDWD then
            State(w):= DONE
        endif
```

Fonte: Elaborado pelo autor

Quadro 13 – Regra MonitoreUpdatingWorkProducts

Atividades cuja função *Use* seja estabelecida na regra *DefineWorkDefinitionParameters*, passam por uma avaliação que pode modificar a sua precedência, denominada *precedência de produto de trabalho* definida no Capítulo 3,

mapeada como a regra *MonitoreActivitiesWithUseWP* no ASM apresentado no Quadro 14.

```
MonitoreActivitiesWithUseWP ≡
∀ a ∈ Activity :
    if  isNotEmpty(set Use(a)) then
                let (act = FIRST(sequence (ProducedBy (FIRST(sequence (Use (a))))))) in
                if  ExecOrder (act)  >= ExecOrder(a) then
                        Precede(a) := act
                        ExecOrder(a) := ExecOrder (act) + 1
                        endif
                endlet
    endif
```

Fonte: Elaborado pelo autor

Quadro 14 – Regra MonitoreActivitiesWithUseWP

A regra *MonitoreActivitiesWithUseWP* estabelece que caso uma atividade a_{n+1} use um produto de trabalho que seja produzido por uma atividade a_n da rede de atividades, mas sua ordem de execução seja menor que a ordem de execução da atividade a_n, a nova precedência de a_{n+1}, passa a ser a_n, e sua ordem de execução, igual a *ExecOrder(* a_n *)+1,*descrita na seção 2.2.2.4.

Toda atividade tem sua ordem de execução (*ExecOrder*) definida pela função *DefineDefaultOrders* apresentada no Quadro 15.

```
DefineDefaultOrders ≡
        ∀ a ∈ ACTIVITY
          if IsUndef(ExecOrder(a)) then
                ExecOrder (a) := ExecOrder (Precede(a))  + 1
          endif
```

Fonte: Elaborado pelo autor

Quadro 15 – Regra *DefineDefaultOrders*

Outra avaliação feita pelo ASM é dada pela situação quando a função *Update* de uma atividade é estabelecida na regra *DefineWorkDefinitionParameters.* Neste caso a precedência da atividade cuja função *Update* é definida pode ser modificada, conforme a regra *MonitoreActivitiesWithUpdateWP* descrita no Quadro 16.

```
MonitoreActivitiesWithUpdateWP ≡
∀ a ∈ Activity :
    if  isNotEmpty(set Update(a)) then
                let (act = FIRST(sequence (ProducedBy (FIRST(sequence (Update (a))))))) in
```

```
                    if  ExecOrder (act)  >= ExecOrder(a) then
                              Precede(a) := act
                              ExecOrder(a) := ExecOrder (act) + 1
                              endif
                    endlet
     endif
```

Fonte: Elaborado pelo autor

Quadro 16 – Regra *MonitoreActivitiesWithUpdateWP*

A regra *MonitoreActivitiesWithUpdateWP* estabelece que caso uma atividade a_n atualize um produto de trabalho que seja utilizado por uma atividade a_{n+1} da rede de atividades, e a ordem de execução da atividade a_n seja maior que à ordem de execução da atividade a_{n-1} que inicialmente produziu o produto de trabalho, a nova precedência de a_{n+1}, passa a ser a_n, e sua ordem de execução, igual a *ExecOrder (* a_n *)+1,*descrita na seção 2.2.2.4.

Neste trabalho foi definido que um ator pode desempenhar somente uma atividade de cada vez, portanto não há compartilhamento de tempo para a realização de atividades. Dado um conjunto de atividades $a_1...a_n$ que determinado ator deve desempenhar no projeto, o ator deve realizar a atividade a_2 caso tenha concluído a atividade a_1. A regra *MonitoreActivitiesPerformed* estabelece tal definição e é apresentada a seguir no Quadro 17.

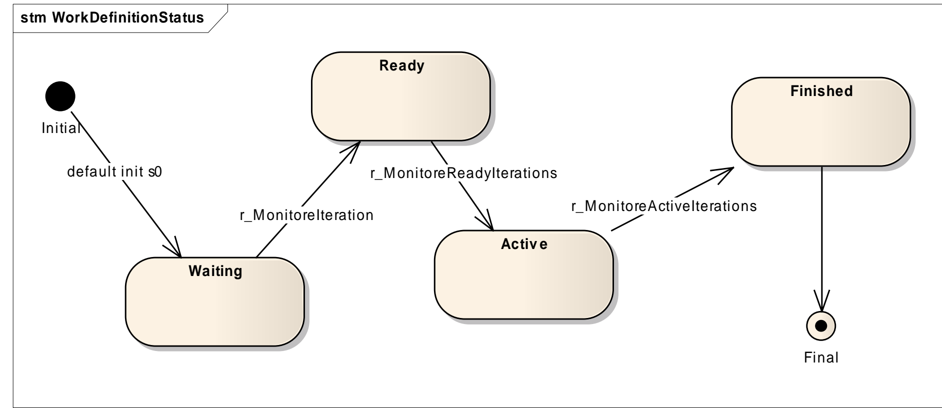
```
MonitoreActivitiesPerformed ≡
∀ a ∈ Activity | sequence Activity(FIRST(sequence(Assistant(a))) ⊃ a :
                    let (act = FIRST(sequence (Activity(self)))) in
                         if  act ≠ a ∧ ExecOrder (act)  = ExecOrder(a) ∧  Assistant (act)  =
          Assistant(a) then
                              Precede(a) := act
                              ExecOrder(a) := ExecOrder (act) + 1
                         endif
                    endlet
     endif
```

Fonte: Elaborado pelo autor

Quadro 17 – Regra MonitoreActivitiesPerformed

Conforme descrito na seção 2.2.1, um conjunto de atividades compõe uma iteração (*Iteration*), e um conjunto de iterações, definem uma fase (*Phase*). Tais elementos também são mapeados para o ASM proposto.

O ciclo de vida da Iteração segue o padrão definido para *WorkDefinition*, conduto as regras de transição são distintas e apresentadas na Figura 41.



Fonte: Elaborado pelo autor

Figura 41 – Ciclo de Vida de uma Iteração do SPEMasm

A regra $init\,S_0$ define o estado inicial da instância de *ITERATION* para WAITINGWD. Para que o estado READYWD seja endereçado, é necessário que pelo menos uma atividade pertencente à iteração avaliada pela regra tenha sido iniciada, conforme a regra *MonitoreIterations,* apresentada no Quadro 18.

MonitoreIterations ≡

  ∀ *it* ∈ *ITERATION* | *State(it)* = *WAITINGWD*:

    if *Activities (it)* ⊃ *FIRST(*sequence *(Activity (it)))* then

      State(it) := READYWD

    endif

Fonte: Elaborado pelo autor

Quadro 18 – Regra MonitoreIterations

Toda iteração que esteja pronta para iniciar, cuja primeira atividade tenha sido iniciada, tem seu estado definido para iniciado, ou ativo, conforme Quadro 19.

MonitoreReadyIterations ≡

  ∀ *it* ∈ *ITERATION* | *State(it)* = *READYWD* :

    if *Activities (it)* ⊃ *FIRST(*sequence *(ActiveActivities (it)))* then

      State(it) := ACTIVEWD

    endif

Fonte: Elaborado pelo autor

Quadro 19 – Regra MonitoreReadyIterations

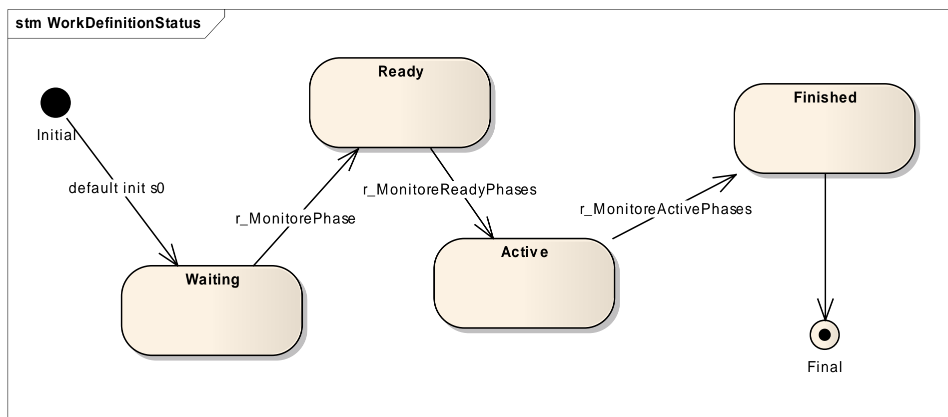
As iterações que estejam em andamento, cuja última atividade tenha sido finalizada, têm seu estado definido como concluído. O Quadro 20 define a semântica da regra *MonitoreActiveIterations*.

MonitoreActiveIterations ≡
∀ *it* ∈ *ITERATION* | *State(it)* = *ACTIVEWD* :
if *LAST(Activities(it))* ⊃ (sequence *(FinishedActivities (it)))* then
State(it) := *FINISHEDWD*
*e*ndif

Fonte: Elaborado pelo autor

Quadro 20 – Regra MonitoreActiveIterations

O ciclo de vida das fases é definido pelos estados e regras apresentados na Figura 42.



Fonte: Elaborado pelo autor

Figura 42 – Ciclo de Vida de uma Fase do SPEMasm

Da mesma forma que ocorre para o domínio ITERATION, a regra *init* S_0 define o estado inicial de todas as instâncias de PHASE para WAITINGWD. Toda fase que esteja com estado inicial (*WAITINGWD*), e a primeira iteração pronta para iniciar, terá seu estado modificado para pronto (*READYWD*). A regra que estabelece tal condição é apresentada no Quadro 21.

MonitorePhase ≡
∀ *ph* ∈ *PHASE* | *State(ph)* = *WAITINGWD* :
if *Iterations(ph)* ⊃ *FIRST(*sequence *(ReadyIterations(ph)))* then
State(ph) := *READYWD*
endif

Fonte: Elaborado pelo autor

Quadro 21 – Regra MonitorePhase

A regra apresentada no Quadro 22 estabelece que quando pelo menos uma iteração é ativada, ou seja, sua execução é iniciada, significa que a fase também foi iniciada.

MonitoreReadyPhases $\equiv$
$\forall$ *ph* $\in$ *PHASE* | *State(ph) = READYWD* :
if *Iterations(ph)* $\supset$ *FIRST(*sequence *(ActiveIterations (ph)))* then
State(ph) := ACTIVEWD
endif

Fonte: Elaborado pelo autor

Quadro 22 – Regra MonitoreReadyPhases

Com o decorrer do tempo das atividades, as iterações são finalizadas e a regra apresentada no Quadro 23, faz a seguinte avaliação: Toda fase com estado ativo, cuja última iteração, seja igual à última iteração finalizada, deve ser finalizada.

MonitoreActivePhases $\equiv$
$\forall$ *ph* $\in$ *PHASE* | *State(ph) = ACTIVEWD* :
if *LAST(*sequence*(Iterations(ph)) = LAST(*sequence *(allFinishedIterations))* then
State(ph) := FINISHEDWD
endif

Fonte: Elaborado pelo autor

Quadro 23 – Regra *MonitoreActivePhases*

5.3 Mapeamento de Restrições sobre o modelo

Para que o modelo possa ser instanciado, algumas premissas precisam ser verdadeiras, isso é, predicados ou axiomas são acionados e a execução da máquina só é iniciada caso eles sejam verdadeiros. As restrições são apresentadas no Quadro 24.

Todas as atividades precisam ter uma duração definida e maior que zero.
constraint over *Duration*
$\forall$ *a* $\in$ *ACTIVITY* | *(Duration(a) > 0* $\wedge$ *IsDef(Duration(a))*
Toda atividade deve ser executada por um ator.
constraint over *Assistant*
$\forall$ *a* $\in$ *ACTIVITY* | *Size(Assistant (a)) > 0*
Toda iteração precisa ter pelo menos uma atividade associada.
constraint over *Activities*
$\forall$ *it* $\in$ *ITERATION* | *Size(Activities (it)) > 0*

Toda fase precisa ter pelo menos uma iteração associada.

constraint over *Iterations*

$$\forall\ ph \in PHASE\ |\ Size(Iterations(ph)) > 0$$

Fonte: Elaborado pelo autor

Quadro 24 – Restrições sobre o Modelo

5.4 Considerações Finais

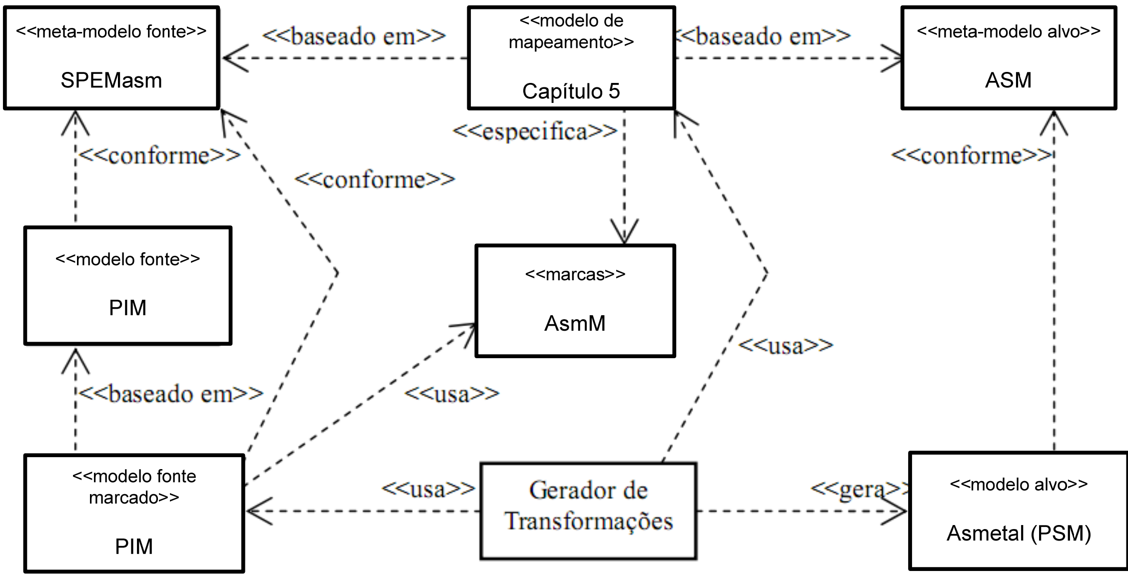
O modelo de mapeamento apresentado neste capítulo é também uma formalização que pode ser utilizada nas fases de validação de modelos de processo de software, já que estabelece o ciclo de vida de cada elemento do processo, e do processo como um todo. O mapeamento pode ser estendido com informações para dar suporte à validação de outras abordagens de modelagem de processos.

A formalização definida também pode ser útil como fonte de requisitos para a elaboração de outros formalismos.

No capítulo seguinte o mapeamento realizado neste capítulo é utilizado na transformação de um modelo independente de plataforma especificado com o *SPEMasm* para máquina de estado abstrata.

6 TRANSFORMAÇÃO DE MODELOS DE PROCESSOS PARA MÁQUINAS DE ESTADO ABSTRATAS

Este capítulo apresenta a transformação do modelo independente de plataforma - PIM marcado (modelo fonte marcado) para o modelo específico de plataforma (modelo alvo). A Figura 43 apresenta o esquema geral do mapeamento de instâncias utilizando marcas definidas adiante e discutidos na seção 2.2.



Fonte: Adaptado pelo autor de (CALIARI, 2003)

Figura 43 - Mapeamento de instâncias.

O *SPEMasm* é utilizado como meta-modelo fonte e especifica elementos de processo de software e suas associações utilizados como referência para a criação do modelo fonte (modelo independente de plataforma – PIM). O PIM apresentado neste trabalho utiliza como referência um modelo de processo adaptado do *OpenUP/Basic* (KROLL, 2006).

Conforme apresentado na Figura 43 o meta-modelo alvo é a teoria de máquinas de estado abstratas (*ASM*), referência para a criação do modelo específico de plataforma (modelo alvo), a partir da transformação realizada manualmente, sem a utilização de um gerador de transformações (OMG, 2003). A transformação do modelo fonte marcado utiliza como referência o modelo de mapeamento, que define os elementos do PIM que devem ser transformados de forma específica em elementos do PSM, de acordo com a plataforma definida para o PSM (ASM), dada uma correspondência entre o elemento do PIM e um ou mais tipos do meta-modelo PSM.

O modelo de mapeamento especifica as marcas usadas para definir como os elementos do PIM devem ser transformados de forma específica em elementos do PSM. Este trabalho utiliza como marcas parte dos elementos do meta-modelo *AsmM*, detalhados adiante neste capítulo, (GARGANTINI, RICCOBENE, SCANDURRA, 2006).

A transformação é realizada em 2 etapas

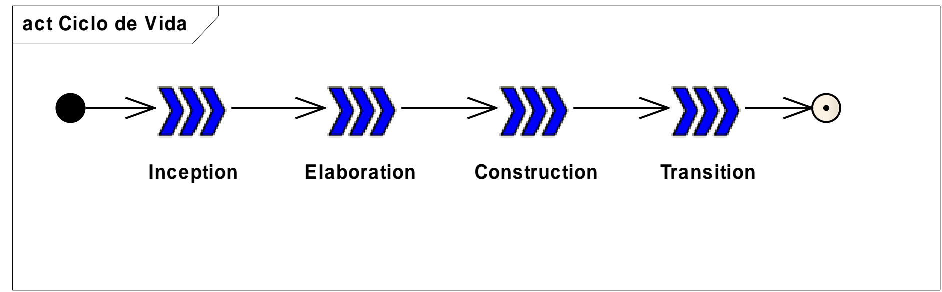
1. Elementos estruturais: são as definições de domínios, funções estáticas e dinâmicas
2. Instâncias: representam os valores para as funções estáticas e os valores iniciais para funções dinâmicas de uma máquina de estado abstrata.

A seguir é realizada a descrição do modelo fonte necessário para a definição do modelo fonte marcado, entrada para o processo de transformação..

6.1 O modelo fonte

Como já foi visto, modelos de processo de software trazem informações sobre o ciclo de vida, tais como fases, atividades, atores, produtos de trabalho e o fluxo sugerido para a execução do trabalho.

O modelo de processo utilizado neste trabalho foi adaptado do *OpenUp/Basic* , um *framework* de processo de domínio público projetado para equipes pequenas em parte baseado na abordagem ágil para desenvolvimento de software (KROLL, 2006) . O ciclo de vida do *OpenUp/Basic* é apresentado no diagrama de atividades da UML, na Figura 44.

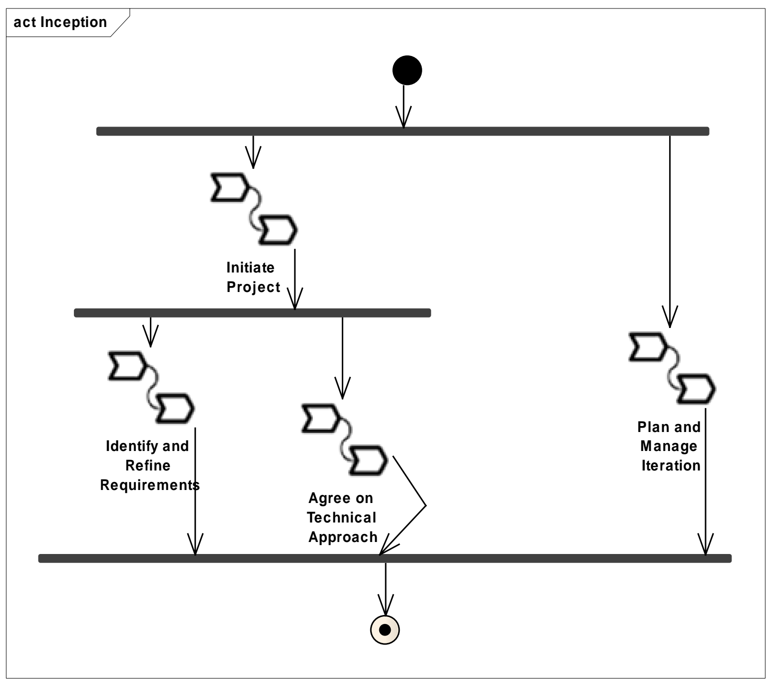


Fonte: Adaptado de (IBM, 2008)

Figura 44 – Ciclo de Vida do *OpenUp/Basic*

O *OpenUp/Basic* possui quatro fases principais, que são organizadas em iterações e atividades. Como se trata de um *framework*, não apresenta o conteúdo de cada iteração, mas um modelo ou *template,* que deve ser adaptado para cada

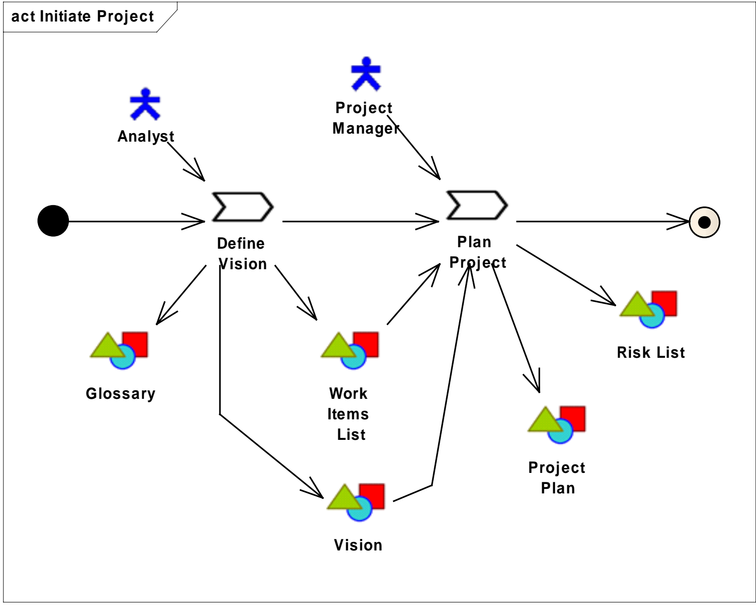
organização ou equipe dada sua realidade. O *template* da fase *Inception* é ilustrado na Figura 45.



Fonte: Adaptado de (IBM, 2008)

Figura 45 – Composição da fase *Inception*

As iterações são organizadas em um conjunto de atividades executadas por atores que desempenham papéis no processo. Tais atividades utilizam, atualizam e produzem produtos de trabalho de variados tipos, tais como, documentos de texto, diagramas, protótipos e código fonte. A Figura 46 apresenta um diagrama de atividades com a organização da iteração *Initiate Project.*



Fonte: Adaptado de (IBM, 2008)

Figura 46 – Composição da iteração *Initiate Project*

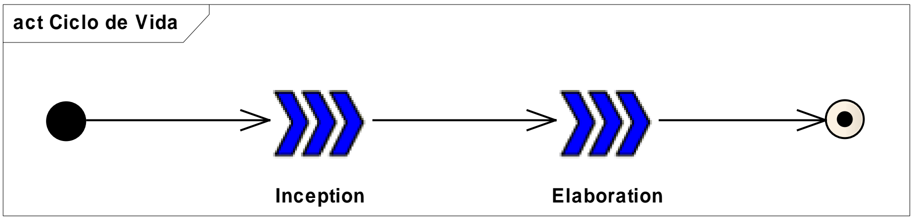
A atividade *Define Vision* é realizada pelo ator do processo, que desempenha o papel (ProcessRole) de Analista de Sistemas (*Analyst*).

O estudo de caso apresentado a seguir utiliza uma adaptação do ciclo de vida apresentado no *OpenUP/Basic*. Iterações das fases *Inception* e *Iteration* foram adaptadas para a realização deste estudo e são apresentadas a seguir.

Para a elaboração do estudo de caso foi realizada uma adaptação do *OpenUp/Basic,* para demonstrar o mapeamento, como também a transformação para a máquina de estado abstrata especificada. O modelo adaptado representa o nível M1 do MOF, podendo ser instanciado para um ou mais projetos de software, representando assim o PIM (*Platform Independent Model).*

Na fase *Inception* acontecem os primeiros contatos com os *stakeholders (*envolvidos com o projeto) e as informações para início do projeto são levantadas, para serem utilizadas como subsídio na elaboração do planejamento e da visão geral sobre o projeto.

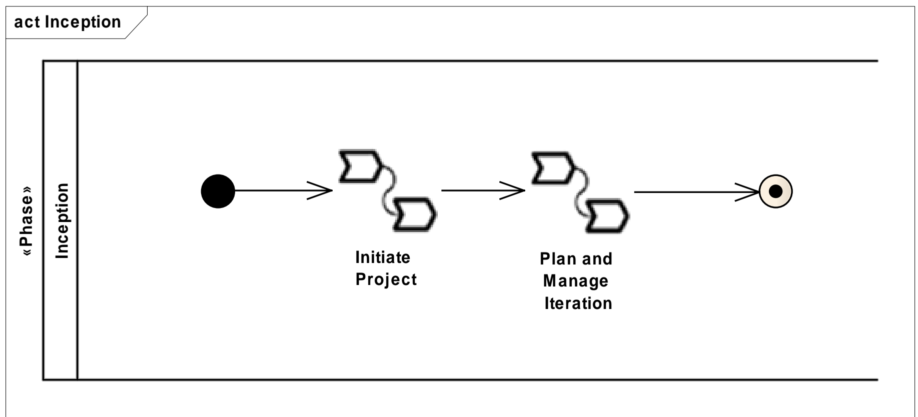
Na fase *Elaboration* são propostas soluções ao problema apresentado na fase *Inception*, ou seja, são realizadas a especificação dos requisitos e dos Casos de Uso, e posteriormente a primeira versão da arquitetura é desenvolvida. A Figura 47 apresenta a seqüência com as fases do modelo de processo utilizado neste estudo de caso, caracterizado como modelo *Cascata*.



Fonte: Adaptado de (IBM, 2008)

Figura 47 – Ciclo de Vida do processo utilizado no estudo de caso

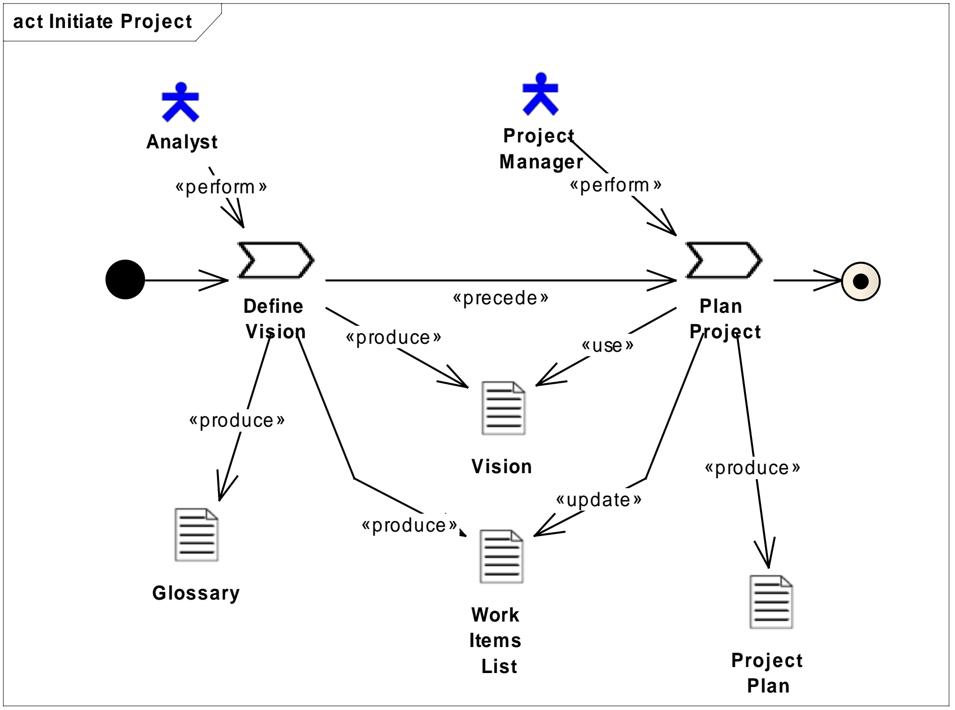
A fase *Inception* adaptada é composta das iterações mostradas na Figura 48, representadas pelo estereótipo *WorkDefinition* do SPEM. A raia (*swimlane*) do diagrama apresentado foi estereotipada como uma fase (*Phase*) do SPEM.



Fonte: Adaptado de (IBM, 2008)

Figura 48 – Composição da fase Inception.

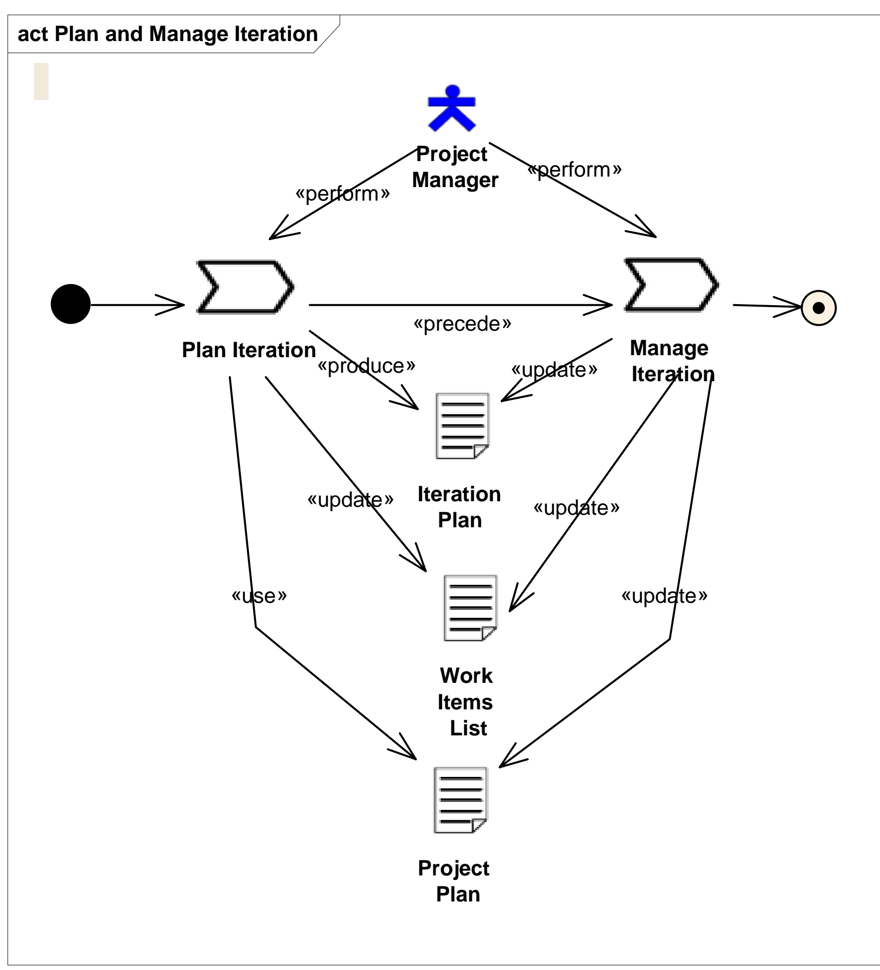
A iteração é composta por atividades, produtos de trabalho e atores (papéis) do processo. Os atores desempenham papéis e são responsáveis por executar (*perform*) as atividades (*Activity*). Conforme já foi visto, uma atividade pode utilizar (*use*), atualizar (*update*), e produzir (*produce)* artefatos, ou produtos de trabalho utilizados durante o projeto. Uma atividade pode preceder *(precede)* outras, ou seja, para que a atividade *n + 1* ocorra, é necessário que a atividade *n* seja finalizada. A notação utilizada no elemento *Glossary* é a representação de um documento (*Document*), uma especialização de produto de trabalho (*WorkProduct*). A composição da iteração *Initiate Project* é representada pelo diagrama na Figura 49.



Fonte: Adaptado de (IBM, 2008)

Figura 49 – Iteração Initiate Project

A iteração *Plan and Manage Iteration*, que faz parte da fase *Inception*, tem como foco permitir que as iterações do projeto sejam planejadas e gerenciadas. A Figura 50 mostra a iteração.

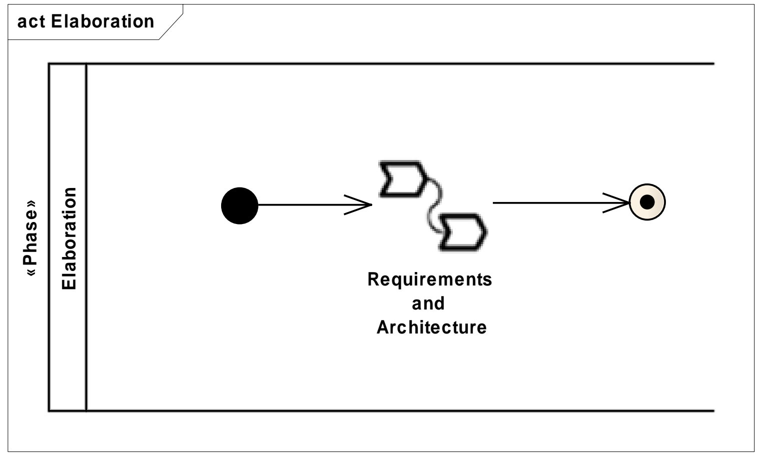


Fonte: Adaptado de (IBM, 2008)

Figura 50 – Plan and Manage Iteration

A atividade *Plan Iteration* produz o documento *Iteration Plan*, baseando-se no documento *Project Plan* (use), e atualiza o documento *Work Items List*. Em seguida a atividade *Manage Iteration* atualiza os documentos da iteração de acordo com o acompanhamento realizado do projeto. É importante notar que a atividade *Manage Iteration* ocorre somente após a atividade *Plan Iteration*, já que tem uma relação de precedência, representada pelo estereótipo *precede.*

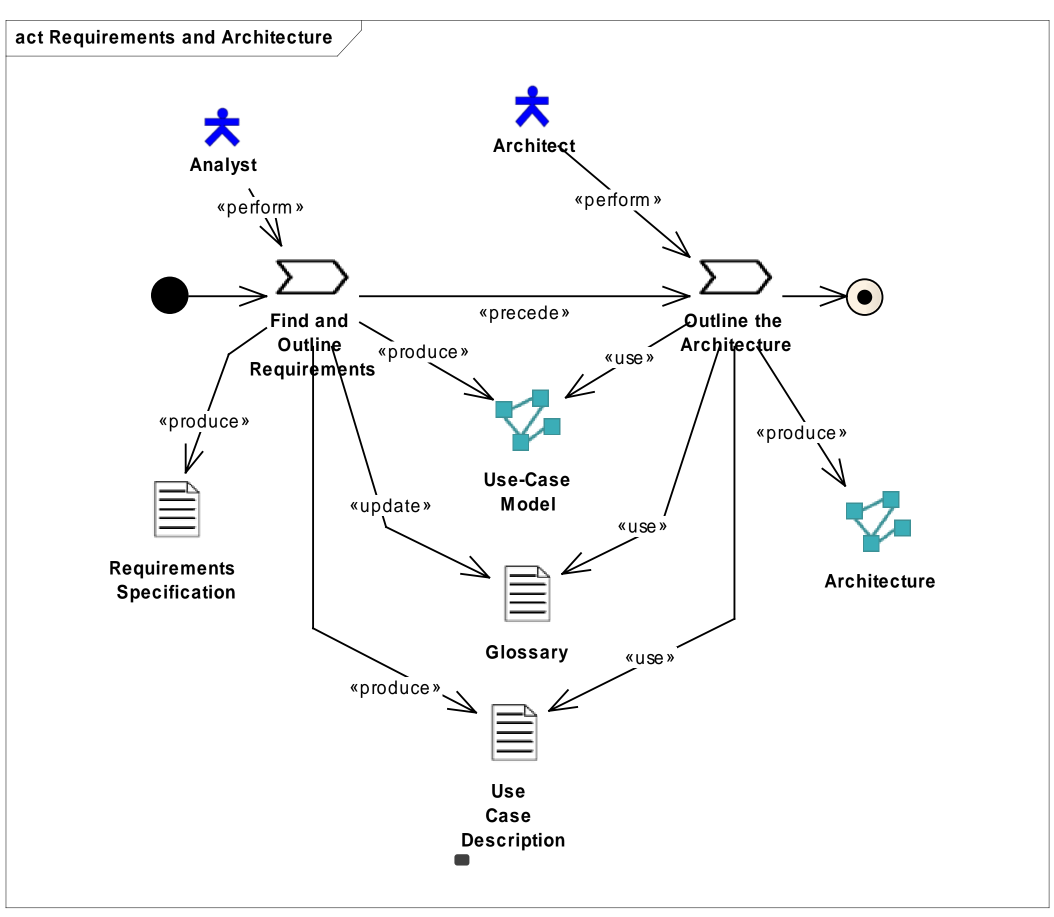
A fase *Elaboration*, foi adaptada para este trabalho e é composta de uma iteração, representada na Figura 51.



Fonte: Adaptado de (IBM, 2008)

Figura 51 – Composição da fase Elaboration

A iteração *Requirements and Architecture* da fase *Elaboration* é detalhada na Figura 52.



Fonte: Adaptado de (IBM, 2008)

Figura 52 – Iteração Requirements and Architecture

Os produtos de trabalho *Use-Case Model* e *Architecture* são especializações para representar diagramas UML (*UMLModel*) de *WorkProduct* do SPEM.

O modelo fonte (modelo independente de plataforma) é especificado utilizando um Diagrama de Atividades da UML, estereotipado pelo meta-modelo fonte *SPEMasm.* Por exemplo, uma atividade representada em um diagrama de atividades, cujo estereótipo seja *Activity*, utiliza como valores etiquetados os atributos da classe *Activity* do *SPEMasm,* exemplificado na Tabela 3.

Tabela 3 – Exemplo de um modelo fonte marcado

Elemento	MOF	MDA	Representação
Classe *Activity*	Nível M2	Meta-modelo Fonte	class Activity +precede «SPEMAsm» 1 Activity +/ assistant: ProcessRole + name: String «SPEMAsm» + duration: int + elapsedTime: int + execOrder: int +/ precede: Activity +/ state: WorkDefinitionState
Atividade do modelo de processo conforme *SPEMasm*	Nível M1	Modelo fonte (Modelo Independente de Plataforma – PIM)	*Find and Outline Requirements* *tags* *assistant =* *duration =* *elapsedTime =* *execOrder =* *id = a1* *precede =* *state =*

Fonte: Elaborado pelo Autor

No exemplo apresentado na Tabela 3, a atividade *Find and Outline Requirements* tem como estereótipo o elemento *Activity* do meta-modelo fonte, representado pela classe *Activity,* e que utiliza a notação SPEM (OMG, 2005). Os atributos da classe *Activity* são usados na atividade como valores etiquetados (*tagged values)* já que de acordo com a definição da UML, atividades não possuem atributos (OMG, 2008c). A atividade “*Find and Outline Requirements*” apresentada na Tabela 3 utiliza a notação (gráfica) proposta pelo SPEM (OMG, 2005), no entanto, pode utilizar como alternativa o estereótipo textual <<*Activity*>>, apresentado na seção 2.2.2.

A Tabela 4 apresenta os valores etiquetados adicionados ao modelo fonte marcado para a iteração *Initiate Project,* ilustrada no diagrama de atividades na Figura 48.

Tabela 4 – Valores etiquetados na iteração *Initiate Project*

Elemento do Diagrama	Valores etiquetados	
Define Vision	*id*	
	name	*Define Vision*
	duration	
	elapsedTime	
	execOrder	
	precede	
	state	*WAITINGWD*
	assistant	*Analyst*
	produce	*Glossary, Vision, Work Itens List*
Analyst	*id*	
	name	*Analyst*
	state	*AVAILABLEPP*
	Perform	*Define Vision*
Glossary	*id*	
	name	*Glossary*
	isDeliverable	*True*
	kind	*TextDocument*
	responsibleRole	*Analyst*
	state	
Vision	*id*	
	name	*Vision*
	isDeliverable	*True*
	kind	*TextDocument*
	responsibleRole	*Analyst*
	state	
Work Itens List	*id*	
	name	*Work Itens List*
	isDeliverable	*False*
	kind	*TextDocument*
	responsibleRole	*Analyst*
	state	
Plan Project	*Id*	
	name	*Plan Project*
	duration	
	elapsedTime	
	execOrder	
	precede	
	state	*WAITINGWD*
	assistant	*Project Manager*
	produce	*Project Plan*
	use	*Vision*
	update	*Work Itens List*
Project Manager	*id*	
	name	*Project Manager*
	state	*AVAILABLEPP*
	perform	*Plan Project*
Project Plan	*id*	
	name	*Project Plan*
	isDeliverable	*True*
	kind	*TextDocument*
	responsibleRole	*Project Manager*
	state	

Fonte: Elaborado pelo Autor

Na Tabela 4, as informações definidas na coluna da direita dos valores etiquetados são necessárias no modelo fonte, já os valores em branco são gerados no momento da transformação utilizando as informações do modelo fonte marcado.

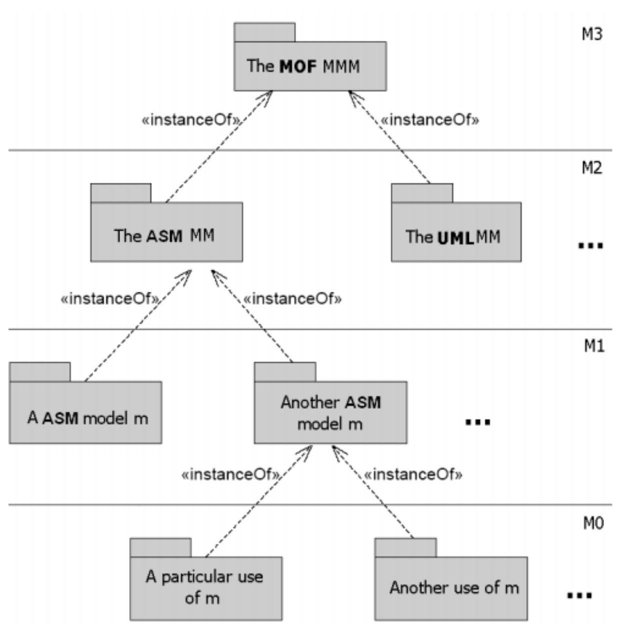
6.2 Marcas

Esta seção apresenta marcas especificadas a partir do modelo de mapeamento baseado nos conceitos de máquinas de estado abstratas (*Abstract State Machine – ASM*) descrito no Capítulo 5. Conforme já discutido na seção 2.2.2.3, a *MDA* define que mapeamentos podem ser usados para transformar PIM (modelos independentes de plataforma) em PSM (modelos específicos de plataforma) por meio do mapeamento de instâncias utilizando marcas.

De acordo com Gargantini, Riccobene e Scandurra (2006) o AsmM pode ser visto como a primeira iniciativa para a definição de um padrão de família de linguagens para ASM e é formado por:

- Definição do meta-modelo baseado no MOF, representando uma notação visual para representação dos conceitos de ASM;
- Um sintaxe de intercambio, baseada em XMI, derivada do *AsmM* para intercâmbio de modelos ASM entre ferramentas;
- Uma sintaxe concreta;

Sua organização é baseada no MOF, e é apresentada na Figura 53:



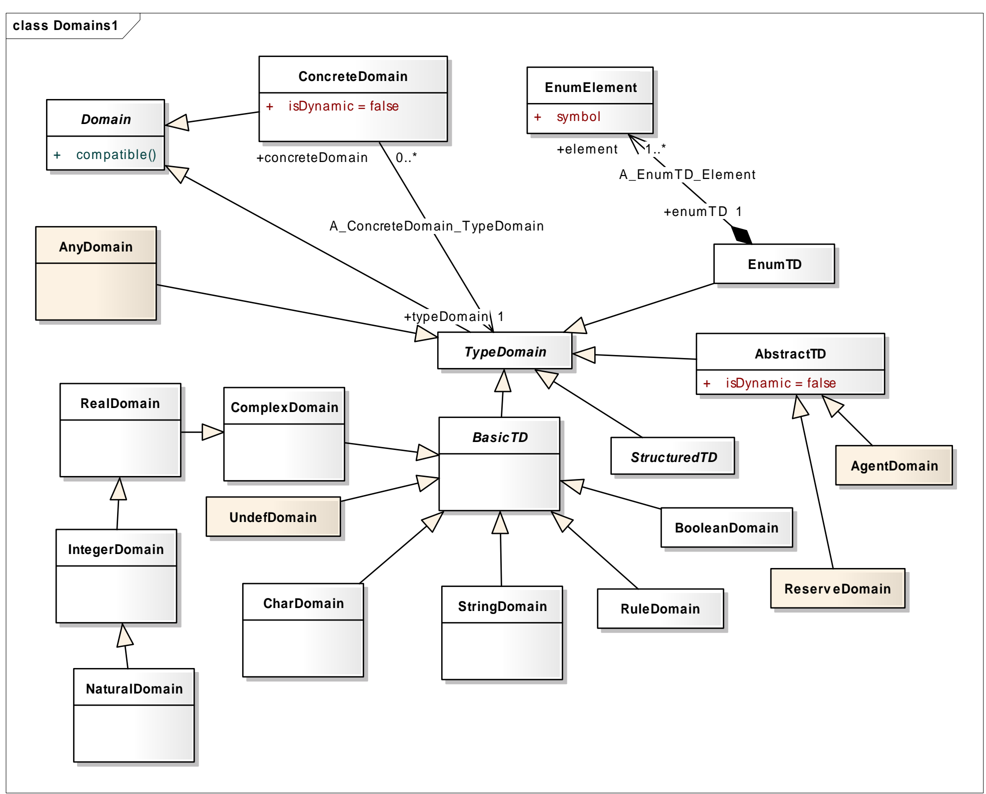
Fonte: (GARGANTINI, RICCOBENE, SCANDURRA, 2006)

Figura 53 – Estrutura baseada no MOF

De acordo com a Figura 52, o meta-modelo *AsmM* é definido no nível M2, e utiliza os construtores do MOF. No nível M1, modelos ASM podem ser especificados usando o *AsmM*, e instanciados com valores para um modelo no nível M0.

No nível M2, o *AsmM* utiliza os conceitos do MOF, como classes, atributos e associações para representar os conceitos de Máquinas de Estado Abstratas (GUREVICH, 1995).

Não é objetivo deste trabalho apresentar detalhadamente o meta-modelo *AsmM*, mas apresentar os principais conceitos definidos de acordo com os construtores do MOF, e discutidos no Capítulo 5. A Figura 54 traz a organização dos tipos de domínio definidos para máquinas de estado abstratas, no meta-modelo *AsmM*.



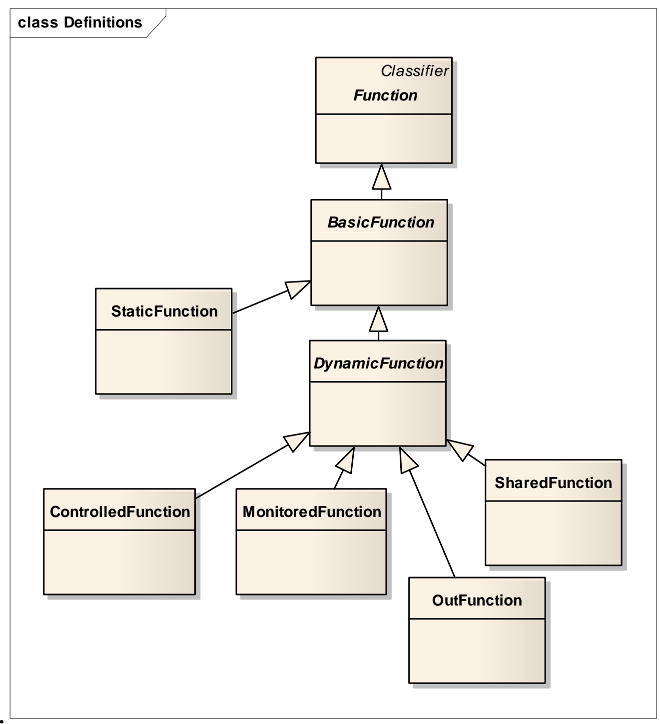
Fonte: (GARGANTINI, RICCOBENE, SCANDURRA, 2006)

Figura 54 – Domínios

Para especificação de processos de software, a máquina de estado abstrata especificada neste trabalho utiliza os domínios *AbstractTD*, *AgentDomain* e *EnumTD,* e domínios para tipos de dados básicos, tais como *IntegerDomain* para representação de números inteiros, *StringDomain*, para representação de palavras ou cadeia de caracteres, *BooleanDomain*, que representam valores booleanos e *RuleDomain*, para regras de transição.

De acordo com os conceitos de máquinas de estado abstratas apresentados no Capítulo 2, domínios abstratos são representados no meta-modelo *AsmM* pela classe *AbstractTD.* Ainda de acordo com tais conceitos, agentes ASM devem ser definidos pela classe *AgentDomain* e domínios que representam conjuntos pré-definidos, ou enumerados, são representados pela classe *EnumTD*.

Funções ASM são organizadas no meta-modelo *AsmM* no pacote denominado *Definitions*, apresentado na Figura 55 (GARGANTINI, RICCOBENE, SCANDURRA, 2006)



Fonte: (GARGANTINI, RICCOBENE, SCANDURRA, 2006)

Figura 55 – Funções

As marcas representadas pelas classes do *AsmM* descritas anteriormente são especificadas pelo modelo de mapeamento conforme Tabela 5.

Tabela 5 – Modelo de Mapeamento e Marcas

Modelo de Mapeamento			**Marcas**
SPEMasm		**ASM**	**Meta-modelo *AsmM***
Estrutura do Processo	*WorkDefitinitionState, WorkProductState, ProcessPerformState, WorkProductKind*	Domínio estático enumerado	Classe *EnumTD*
	Activity, WorkProduct	Domínios abstratos	Classe *AbstractTD*
	ProcessRole	Agente	Classe *AgentDomain*
	Atributos de *Activity*,	Funções	Classes

	WorkProduct e *ProcessRole*	estáticas e dinâmicas	*StaticFunction* e *DynamicFunction*
	Associações	Funções estáticas	Classe *StaticFunction*
	Instâncias	Funções estáticas	Classe *StaticFunction*
Ciclo de Vida do Processo	*Iteration, Phase*	Domínios abstratos	Classe *AbstractTD*
	Precondition e Goal	Funções dinâmicas	Classe *StaticFunction*
	Associações	Funções estáticas	Classe *StaticFunction*

Adaptado de (PARK *et al*, 2007)

As classes do *AsmM* apresentadas na Tabela 5 não são instanciadas para nenhum propósito neste trabalho, mas são utilizadas como valores etiquetados (*tagged values)* no modelo fonte marcado. Valores etiquetados são uma seqüência de caracteres (*string)*, originada de um perfil UML e aplicáveis a elementos da UML estereotipados (OMG, 2008c). A Tabela 6 demonstra detalhadamente como cada elemento do modelo fonte é marcado pelo *AsmM*.

Tabela 6 – Modelo de Mapeamento e Marcas detalhado

Modelo de Mapeamento			**Marcas do *AsmM***
SPEMasm			
Estrutura do Processo	***Elemento***	**Atributo (valor etiquetado)**	**Classe**
	Activity	*id*	*StaticFunction*
		name	*StaticFunction*
		duration	*StaticFunction*
		elapsedTime	*ControlledFunction*
		execOrder	*ControlledFunction*
		precede	*ControlledFunction*
		state	*ControlledFunction*
		assistant	*StaticFunction*
		produce	*StaticFunction*
	ProcessRole	*id*	*StaticFunction*
		name	*StaticFunction*
		state	*ControlledFunction*
		perform	*StaticFunction*
	WorkProduct	*id*	*StaticFunction*
		name	*StaticFunction*
		isDeliverable	*StaticFunction*
		kind	*StaticFunction*
		responsibleRole	*StaticFunction*
		state	*ControlledFunction*
	Elementos de		*EnumTerm*

	WorkProductState, WorkDefinitionState, ProcessPerformerState		
Ciclo de Vida do Processo	*Iteration*	*id*	*StaticFunction*
		name	*StaticFunction*
		state	*ControlledFunction*
		precondition	*StaticFunction*
		goal	*StaticFunction*
		activities	*StaticFunction*
	Phase	*id*	*StaticFunction*
		name	*StaticFunction*
		state	*ControlledFunction*
		precondition	*StaticFunction*
		goal	*StaticFunction*
		iterations	*StaticFunction*

Fonte: Elaborado pelo Autor

6.3 O modelo fonte marcado

Conforme foi visto o modelo fonte apresentado na seção anterior representa o modelo independente de plataforma (*PIM),* que pode ser transformado para qualquer plataforma específica, pois apresenta os conceitos fundamentais de processo de software e indica as associações entre esses conceitos. A Tabela 7 apresenta o modelo fonte marcado da iteração *Initiate Project* adaptado do modelo fonte (representado na Tabela 5).

Tabela 7 – Modelo fonte marcado para a *Initiate Project*

Elemento do Diagrama	Valores etiquetados		Marcas do *AsmM*
Define Vision	*id*		*StaticFunction*
	name	*Define Vision*	*StaticFunction*
	duration		*StaticFunction*
	elapsedTime		*ControlledFunction*
	execOrder		*ControlledFunction*
	precede		*ControlledFunction*
	state	*WAITINGWD*	*ControlledFunction*
	assistant	*Analyst*	*StaticFunction*
	produce	*Glossary, Vision, Work Itens List*	*StaticFunction*
Analyst	*id*		*StaticFunction*
	name	*Analyst*	*StaticFunction*
	state	*AVAILABLEPP*	*ControlledFunction*
	Perform	*Define Vision*	*StaticFunction*
	id		*StaticFunction*
	name	*Glossary*	*StaticFunction*
	isDeliverable	*True*	*StaticFunction*

Glossary	kind	TextDocument	StaticFunction
	responsibleRole	Analyst	StaticFunction
	state		ControlledFunction
Vision	id		StaticFunction
	name	Vision	StaticFunction
	isDeliverable	True	StaticFunction
	kind	TextDocument	StaticFunction
	responsibleRole	Analyst	StaticFunction
	state		ControlledFunction
Work Itens List	id		StaticFunction
	name	Work Itens List	StaticFunction
	isDeliverable	False	StaticFunction
	kind	TextDocument	StaticFunction
	responsibleRole	Analyst	StaticFunction
	state		ControlledFunction
Plan Project	Id		StaticFunction
	name	Plan Project	StaticFunction
	duration		StaticFunction
	elapsedTime		ControlledFunction
	execOrder		ControlledFunction
	precede		ControlledFunction
	state	WAITINGWD	ControlledFunction
	assistant	Project Manager	StaticFunction
	produce	Project Plan	StaticFunction
	use	Vision	StaticFunction
	update	Work Itens List	StaticFunction
Project Manager	id		StaticFunction
	name	Project Manager	StaticFunction
	state	AVAILABLEPP	ControlledFunction
	perform	Plan Project	StaticFunction
Project Plan	id		StaticFunction
	name	Project Plan	StaticFunction
	isDeliverable	True	StaticFunction
	kind	TextDocument	StaticFunction
	responsibleRole	Project Manager	StaticFunction
	state		ControlledFunction

Fonte: Elaborado pelo Autor

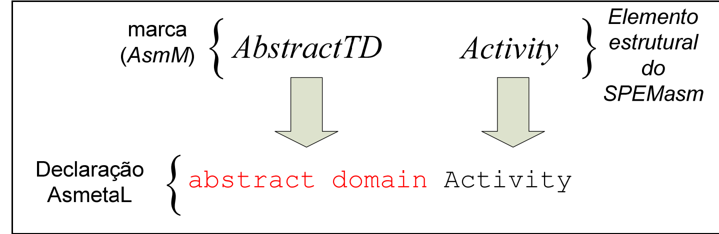
A representação em colunas apresentada na Tabela 7 foi uma opção para dar maior legibilidade ao modelo, já que a representação no diagrama de atividades, com a representação dos valores etiquetados (*tagged values*) não seria adequada, pela quantidade de elementos gráficos e textuais necessários. Ferramentas para construção de diagramas UML possuem recursos necessários para que os valores etiquetados sejam informados, como também para que as marcas sejam aplicadas como estereótipos.

6.4 Transformação do modelo fonte marcado

Esta seção apresenta a transformação do modelo fonte marcado para o modelo alvo na linguagem *AsmetaL*. *AsmetaL* é também definida como notação textual para o meta-modelo *AsmM* usada para descrição de modelos ASM de maneira textual (GARGANTINI, RICCOBENE, SCANDURRA, 2007a). É complemetamente independente de plataforma já que sua notação utiliza os conceitos das máquinas de estado abstratas cuja sintaxe é próxima da notação matemática, o que permite uma abstração de detalhes tecnológicos (GARGANTINI, RICCOBENE, SCANDURRA, 2007a). Contudo, a *AsmetaL* é adotada como modelo específico de plataforma neste trabalho, uma vez que é a plataforma alvo para transformação do modelo independente de plataforma (modelo fonte marcado) que utiliza a notação gráfica do diagrama de atividades da UML. Tal abordagem não impede que a especificação escrita em *AsmetaL* seja adotada como modelo independente de plataforma para transformação para outras plataformas, por exemplo, linguagens de programação como Java, C#, C++ dentre outras.

É importante ressaltar que o modelo representa o nível M1 do MOF e não representa a instância para um projeto de software específico. Uma organização desenvolvedora de software que adota um modelo de processo pode utilizá-lo no desenvolvimento de software para seus clientes. A transformação descrita é realizada no nível M1 do MOF, contudo, pode ser realizada também para uma instância de um projeto.

A seguir os elementos estruturais do *SPEMasm* mapeados como domínios, funções estáticas e dinâmicas na teoria de máquinas de estado abstratas, marcado com o *AsmM* na Tabela 5 são transformados para a linguagem *AsmetaL*. A Figura 56 ilustra a transformação do elemento *Activity*.



Fonte: Elaborado pelo autor

Figura 56 – Declaração do domínio abstrato *Activity*.

As setas da Figura 55 representam a transformação realizada do elemento *Activity*, marcado como *AbstractTD*.

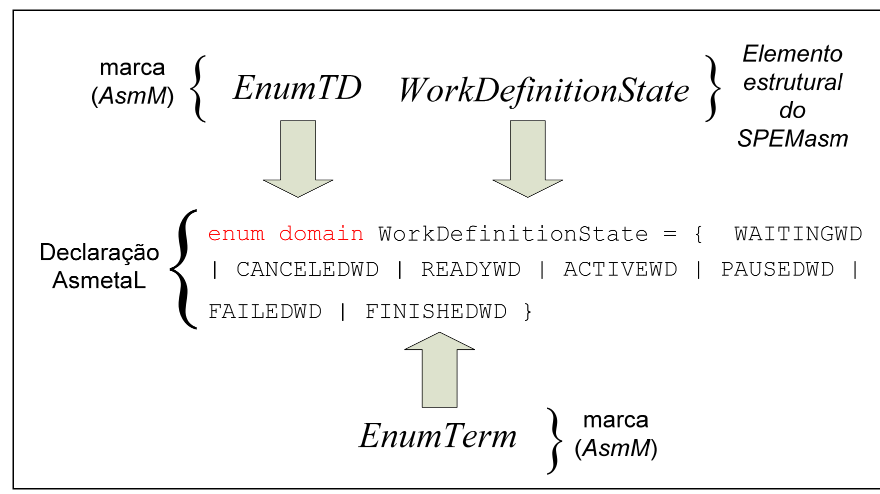
Conforme já foi discutido o elemento estrutural *Activity* do *SPEMasm* apresenta atributos no nível M2 do MOF e quando utilizado no nível M1 no diagrama de atividades, tais atributos são utilizados como valores etiquetados (*tagged value).* A transformação dos atributos do elemento *Activity* é apresentada na Tabela 8.

Tabela 8 – Transformação do elemento estrutural *Activity*

Valores etiquetados	Marcas do *AsmM*	Declaração com *AsmetaL*
name	*StaticFunction*	`static name: Activity -> String`
duration	*StaticFunction*	`static duration: Activity -> Integer`
elapsedTime	*ControlledFunction*	`dynamic controlled elapsedTime: Activity -> Integer`
execOrder	*ControlledFunction*	`dynamic controlled execOrder: Activity -> Integer`
precede	*ControlledFunction*	`dynamic controlled precede: Activity -> Activity`
state	*ControlledFunction*	`dynamic controlled state: Activity -> WorkDefinitionState`
assistant	*StaticFunction*	`static assistant: Activity -> Agent`
produce	*StaticFunction*	`static produce: Activity -> Seq(WorkProduct)`

Fonte: Elaborado pelo Autor

Os atributos da classe *Activity* são considerados na transformação, por exemplo, o atributo *duration* do tipo inteiro é representado na função estática transformada, cujo valor é um domínio inteiro. O estado da atividade é definido pelo elemento *WorkDefinitionState* apresentado na Figura 57.



Fonte: Elaborado pelo autor

Figura 57 – Declaração do domínio abstrato *WorkDefinitionState.*

O atributo *produce* indica os produtos de trabalho que são gerados em uma atividade, já a associação *assistant* é transformada para uma função estática cujo valor é

um agente ASM (*Agent*) e portanto um ator do processo de software (*ProcessRole*), cuja declaração é apresentada a seguir:

```
domain ProcessRole subsetof Agent
```

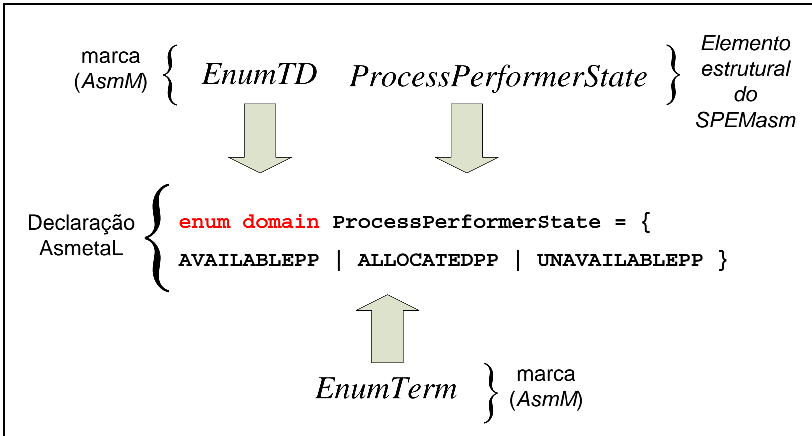
O domínio concreto Agent é controlado pela máquina de estado abstrata e pode instanciar de forma paralela um ASM por agente. Isso significa que, no exemplo em questão, os atores do processo *Analyst* e *Project Manager* iniciam o conjunto de regras especificados na Figura 36 de forma paralela, de modo que com o término das atividades cada ator inicia o trabalho em seguida, sem precisar seguir uma seqüência pré-definida. A transformação do elemento *ProcessRole* é apresentada na Tabela 9.

Tabela 9 – Transformação do elemento estrutural *ProcessRole*

Valores etiquetados	Marcas do *AsmM*	Declaração com *AsmetaL*
Name	*StaticFunction*	`static name: ProcessRole -> String`
State	*ControlledFunction*	`dynamic controlled state: ProcessRole -> ProcessPerformerState`
Perform	*StaticFunction*	`static perform: ProcessRole -> Seq(Activity)`

Fonte: Elaborado pelo Autor

A transformação do elemento *ProcessPerformerState,* que representa os estados possíveis para um ator durante a execução do processo de software é apresentado na Figura 58.



Fonte: Elaborado pelo autor

Figura 58 – Declaração do domínio abstrato *ProcessPerformerState.*

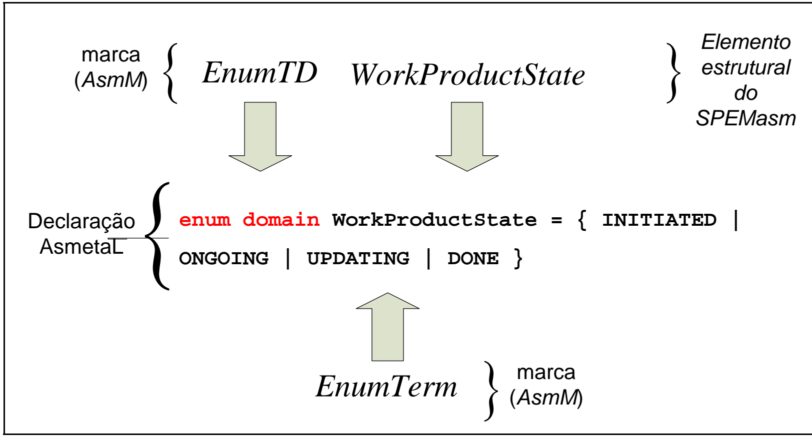
Conforme já foi visto, atividades executadas pelos atores geram, atualizam e usam produtos de trabalho representados no *SPEMasm* pelo elemento estrutural *WorkProduct*, cuja transformação é realizada na Tabela 10.

Tabela 10 – Transformação do elemento estrutural *WorkProduct*

Valores etiquetados	Marcas do *AsmM*	Declaração com *AsmetaL*
name	*StaticFunction*	`static name: WorkProduct -> String`
isDeliverable	*StaticFunction*	`static isDeliverable: WorkProduct -> Boolean`
kind	*StaticFunction*	`static kind: WorkProduct -> WorkProductKind`
responsibleRole	*StaticFunction*	`static responsibleRole: WorkProduct -> ProcessRole`
state	*ControlledFunction*	`dynamic controlled state: WorkProduct -> WorkProductState`

Fonte: Elaborado pelo Autor

A transformação do elemento *WorkProductState* é apresentado na Figura 59.



Fonte: Elaborado pelo autor

Figura 59 – Declaração do domínio abstrato *WorkProductState*.

A segunda etapa da transformação é realizada para as instâncias dos domínios declarados nas Figuras 57, 58, 59 e nas Tabelas 8, 9 e 10. As instâncias não fazem parte das informações do nível M1 do MOF, mas do nível M0. Deve haver uma distinção entre o modelo fonte marcado e a instância deste modelo, já que essa etapa realiza a transformação considerando a geração de valores iniciais para o simulador apresentado no próximo capítulo.

A tabela 11 apresenta a transformação realizada na iteração *Initiate Project.*

Tabela 11 – Transformação da instância do modelo fonte marcado para a iteração *Initiate Project*

Elemento do Diagrama	Valores etiquetados		Declaração com *AsmetaL*
	id	*a1*	`static a1: Activity`

Vision	Name	Define Vision	`name(a1) := "Define Vision"`
	Duration		`duration(a1) := 3`
	elapsedTime		`elapsedTime($a) := 0`
	execOrder		`execOrder(a1) := 1`
	precede		`precede(a1) := undef`
	state	WAITINGWD	`state($a) := WAITINGWD`
	assistant	Analyst	
	produce	Glossary, Vision, Work Itens List	`produce(a1) := [wp1, wp2, wp3]`
Analyst	id	pr1	`static pr1: ProcessRole`
	name	Analyst	`name(pr1) := "Analyst"`
	state	AVAILABLEPP	`state($pr) := AVAILABLEPP`
	perform	Define Vision	`perform(pr1):= [a1]`
Glossary	id	wp1	`static wp1: WorkProduct`
	name	Glossary	`name(wp1) := "Glossary"`
	isDeliverable	True	`isDeliverable(wp1) := true`
	kind	TextDocument	`kind(wp1) := TEXTDOCUMENT`
	responsibleRole	Analyst	
	state		`state($w) := INITIATED`
Vision	id	wp2	`static wp2: WorkProduct`
	name	Vision	`name(wp2) := "Vision"`
	isDeliverable	True	`isDeliverable(wp2) := true`
	kind	TextDocument	`kind(wp2) := TEXTDOCUMENT`
	responsibleRole	Analyst	
	state		
Work Itens List	id	wp3	`static wp3: WorkProduct`
	name	Work Itens List	`name(wp3) := "Work Items List"`
	isDeliverable	False	`isDeliverable(wp3) := false`
	kind	TextDocument	`kind(wp3) := TEXTDOCUMENT`
	responsibleRole	Analyst	
	state		
Plan Project	id		`static a2: Activity`
	name	Plan Project	`name(a2) := "Plan Project"`
	duration		`duration(a2) := 3`
	elapsedTime		`elapsedTime($a) := 0`
	execOrder		`execOrder(a2) := undef`
	precede		`precede(a2) := a1`
	state	WAITINGWD	
	assistant	Project Manager	
	produce	Project Plan	`produce(a2) := [wp4]`
	use	Vision	`update(a2):=[wp3]`
	update	Work Itens List	`use(a2) := [wp2]`
Project	id	pr2	`static pr2: ProcessRole`
	name	Project Manager	`name(pr2) := "Project Manager"`
	state	AVAILABLEPP	

Manager	perform	Plan Project	`perform(pr2):= [a2]`
Project Plan	id	wp4	`static wp4: WorkProduct`
	name	Project Plan	`name(wp4) := "Project Plan"`
	isDeliverable	True	`isDeliverable(wp4) := false`
	kind	TextDocument	`kind(wp4) := TEXTDOCUMENT`
	responsibleRole	Project Manager	
	state		

Fonte: Elaborado pelo autor

Na Tabela 11 a coluna “Declaração com *AsmetaL”* exibe o resultado da transformação, que usa os valores etiquetados como referência. O valor *id* é transformado em uma função estática cujo identificador é a instância do elemento, por exemplo a atividade *Define Vision* foi criada a seguinte declaração:

```
static a1: Activity
```

O identificador *a1* é utilizado como referência para definir os valores dos demais atributos.

Os atributos *responsibleRole* e *assistant* não são transformados, mas são mapeados como operações no simulador. A operação mapeada *responsibleRole* fornece como resultado o ator responsável pela atividade que gera o produto de trabalho.

A operação *assistant* resulta no ator que está executando uma atividade.

As funções estáticas *state* são aplicadas a variáveis iniciadas por *$w* uma só vez no modelo, para os primeiros elementos transformados.

Os valores dos atributos *perform* e *precede* são atribuídos analisando o fluxo de objetos no diagrama de atividades (OMG, 2008c) do ator para a atividade, e de uma atividade para outra atividade respectivamente. Já os atributos *produce, use* e *update* referem-se aos fluxos de objeto das atividades para os produtos de trabalho, usando o mecanismo de estereótipo para definir o tipo da relação (produção, uso ou atualização).

6.5 Considerações Finais

O mapeamento de instâncias, utilizado como abordagem de mapeamento neste trabalho permitiu o uso do meta-modelo *AsmM* como modelo de marcas, definindo uma nova aplicação para o meta-modelo proposto por Gargantini et al (2007a).

De acordo com pesquisas bibliográficas realizadas a transformação manual não é uma abordagem utilizada com freqüência na MDA, apesar de prevista pela OMG

(OMG, 2005). Contudo com este trabalho foi possível verificar que pode ser utilizada para validar e apoiar a especificação de regras de transformação automatizadas por linguagens de transformação de modelos como ATL[6] (*Atlas Transformation Language).*

A máquina de estado abstrata produzida pela transformação manual é utilizada no capítulo seguinte em um estudo de caso para: validar as regras elaboradas no Capítulo 5; simular a execução do modelo fonte, representação estática de um modelo de processo.

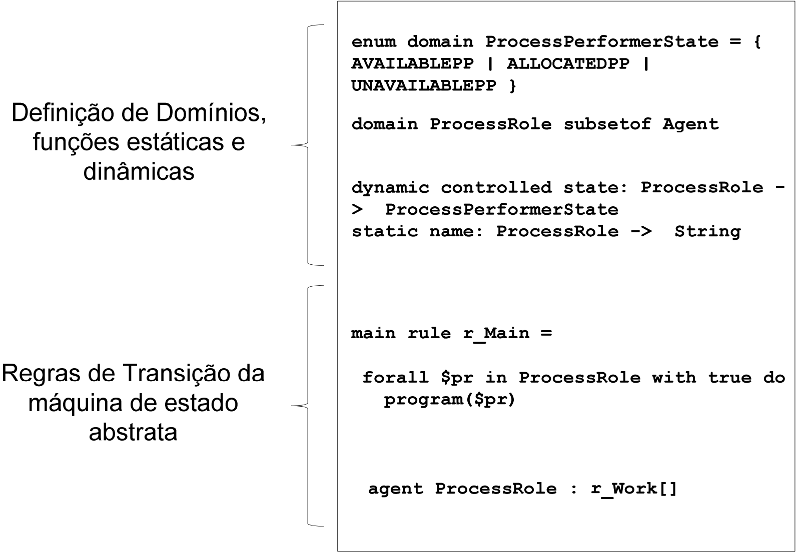
[6] http://www.eclipse.org/m2m/atl/

7 SIMULAÇÃO DE PROCESSOS DE SOFTWARE COM O MODELO ALVO

Este capítulo apresenta um estudo onde é feita a simulação do processo de software descrito com máquinas de estado abstratas. A simulação utiliza os dados do processo instanciado e pode auxiliar o Engenheiro de Processo a verificar e validar o processo projetado.

A máquina de estado abstrata implementada neste trabalho é composta pelas regras de transição que definem e comportamento do processo, baseado nos conceitos do *SPEMasm*, e as transformações realizadas no Capítulo 6. Sua implementação completa é apresentada no Apêndice B.

A estrutura da máquina de estado abstrata é ilustrada na Figura 60.

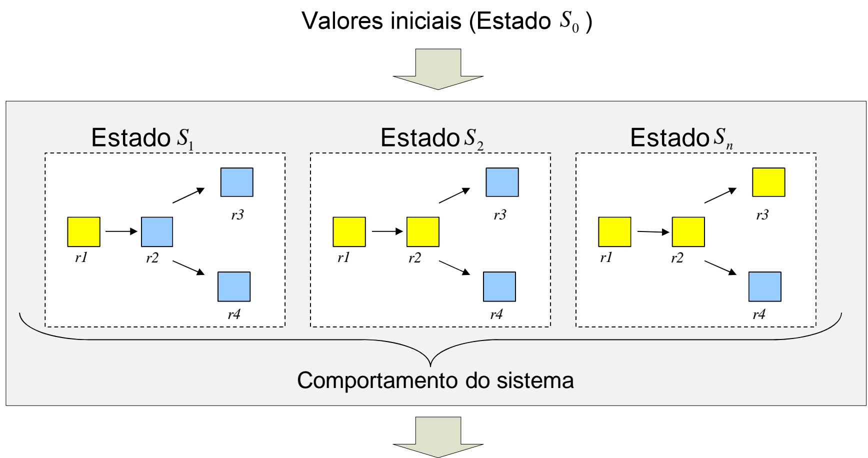


Fonte: Elaborado pelo autor

Figura 60 – Estrutura de uma ASM

A transformação é realizada somente para definição de domínios, funções estáticas e dinâmicas, já as regras de transição não fazem parte da transformação, pois representam o comportamento da máquina de estado abstrata. A máquina utiliza como entrada as informações transformadas (domínios, funções estáticas e dinâmicas) e dados de instância (de um projeto de software) e realiza a transição dos estados segundo regras definidas, até que todos os estados sejam alcançados, ou seja, desde a data de início de um projeto de software, até sua finalização, quando todas as atividades forem finalizadas por todos os atores do processo.

A Figura 61 demonstra um detalhamento da Figura 60.



Fonte: Elaborado pelo autor

Figura 61 – Funcionamento da máquina de estado abstrata.

Os valores iniciais (estados S_0) produzidos pela transformação manual no Capítulo 6 são utilizados como entrada para a simulação, e são modificados pelas regras de transição, gerando um registro de todos os estados desde o inicial (estado S_0) até o estado final (estado S_n).

7.1 Informações para simulação: o projeto

A simulação utiliza informações de um projeto específico de software para produzir o registro da transição de estados de todos os elementos do projeto. O registro de cada transição permite ao Engenheiro de Processo identificar eventuais erros na alocação de pessoal, na determinação de prazos para as atividades e na dependência de produtos de trabalho para a realização de atividades no projeto.

O modelo de processo apresentado no capítulo anterior, denominado modelo fonte, foi instanciado para um projeto denominado Estudo de Caso, cujas informações são apresentadas na Figura 62.

Id	Nome da tarefa	Duração	Início	Término	Predecessoras	Nomes dos recursos
1	**Estudo de Caso**	**14 dias**	**Qua 2/1/08**	**Seg 21/1/08**		
2	**Inception**	**10 dias**	**Qua 2/1/08**	**Ter 15/1/08**		
3	**Initiate Project**	**6 dias**	**Qua 2/1/08**	**Qua 9/1/08**		
4	Define Vision	3 dias	Qua 2/1/08	Sex 4/1/08		Analyst
5	Plan Project	3 dias	Seg 7/1/08	Qua 9/1/08	4	Project Manager
6	**Plan and Manage Iteration**	**4 dias**	**Qui 10/1/08**	**Ter 15/1/08**		
7	Plan Iteration	2 dias	Qui 10/1/08	Sex 11/1/08	5	Project Manager
8	Manage Iteration	2 dias	Seg 14/1/08	Ter 15/1/08	7	Project Manager
9	**Elaboration**	**6 dias**	**Seg 14/1/08**	**Seg 21/1/08**		
10	**Requirements and Architecture**	**6 dias**	**Seg 14/1/08**	**Seg 21/1/08**		
11	Find and Outline Requirements	6 dias	Seg 14/1/08	Seg 21/1/08	7	Analyst
12	Outline the Architecture	5 dias	Seg 14/1/08	Sex 18/1/08	7	Architect

Fonte: Elaborado pelo autor

Figura 62 – Informações do projeto Estudo de Caso.

Na Figura 62, as linhas 1 e 9 representam as fases do modelo fonte, composta por iterações, apresentadas nas linhas 3, 6 e 10. As iterações *Initiate Project* e *Plan and Manage Iteration* compõem a fase *Inception*, conforme apresentado no modelo fonte no capítulo anterior. No projeto proposto a unidade de tempo utilizada é o *dia,* e a data inicial do projeto foi estabelecida para o dia 02/01/2008. A determinação da data de término é dada pela soma da data inicial da atividade com a duração em dias, no entanto a máquina de estado abstrata proposta não realiza cálculo de datas. As precedências de atividades são apresentadas nas linhas 5, 7, 8, 11 e 12. Por exemplo a atividade *Plan Project* inicia-se após o término da atividade *Define Vision.*

A determinação da duração do projeto e da data final é realizada pelo software *Microsoft Project*, onde as informações do projeto da Figura 63 foram inseridas. Tais informações são inseridas na máquina de estado abstrata como valores iniciais (estados S_0) através da transformação já discutida no capítulo anterior.

Os produtos de trabalho produzidos, utilizados e atualizados durante o projeto são descritos no modelo de processo, ou modelo fonte que é utilizado como referência pela equipe e não faz parte da Figura 63 e são apresentados na Tabela 12.

Tabela 12 – Resumo dos produtos de trabalho produzidos, atualizados e utilizados no projeto Estudo de Caso.

Fase	Iteração	Atividade	Produtos de Trabalho		
			Produz	**Atualiza**	**Utiliza como referência**
Inception	Initiate Project	Define Vision	Glossary, Vision, Work Items List	-	-
		Plan Project	Project Plan	Work Items List	Vision

	Plan and Manage Iteration	Plan Iteration	Iteration Plan	Work Items List	Project Plan
		Manage Iteration	-	Iteration Plan, Project Plan, Work Items List	-
Elaboration	Requirements and Architecture	Find and Outline Requirements	Requirements Specification, Use Case Description, Use-Case Model	Glossary, Work Items List	Vision
		Outline the Architecture	Architecture	-	Glossary, Use Case Description, Use-Case Model

Fonte: Elaborado pelo autor

As informações apresentadas nesta seção são utilizadas pela máquina de estado abstrato para simulação apresentada na próxima seção.

7.2 Simulação do projeto: Estudo de Caso

Esta seção apresenta a simulação realizada com as informações de estados iniciais inseridas na máquina de estado abstrata. Apesar da linguagem *AsmetaL* permitir a modularização, isso é, a separação das regras de transição, das definições de domínios e funções e instâncias em arquivos distintos (GARGANTINI, RICCOBENE, SCANDURRA, 2007a), tal recurso não foi funcionou satisfatoriamente, impossibilitando a simulação com o código modularizado. Desta forma, os valores iniciais (estados S_0) foram informados no mesmo módulo onde as regras de transição são definidas. O quadro 25 apresenta parte das informações descritas na máquina de estado abstrata :

```
// Duração das atividades
duration(a1) := 3
duration(a2) := 3
duration(a3) := 2
duration(a4) := 2
duration(a5) := 6
duration(a6) := 5

// Tempo decorrido
elapsedTime($a) := 0

// Ordem de execução
execOrder(a1) := 1
execOrder(a2) := undef
execOrder(a3) := undef
execOrder(a4) := undef
execOrder(a5) := undef
execOrder(a6) := undef
```

```
// Precedência
precede(a1) := undef
precede(a2) := a1
precede(a3) := a2
precede(a4) := a3
precede(a5) := a3
precede(a6) := a3

// Produtos de trabalho produzidos, atualizados e utilizados pelas atividades.
produce(a1) := [wp1, wp2, wp3]

produce(a2) := [wp4]
update(a2):=[wp3]
use(a2) := [wp2]

produce(a3) := [wp5]
update(a3):=[wp3]
use(a3) := [wp4]

update(a4):=[wp5, wp4, wp3]

produce(a5) := [wp6, wp7, wp8]
update(a5):=[wp1, wp3]
use(a5) := [wp2]

produce(a6) := [wp9]
use(a6) := [wp8, wp7, wp1]
```

Fonte: Elaborado pelo autor

Quadro 25 – Informações do estudo de caso.

A função *duration* estabelece a duração estimada das atividades estabelecidas para o projeto Estudo de Caso. A função *elapsedTime* é inicializada com 0 (zero) já que é responsável por armazenar o tempo decorrido de cada atividade, após o início do projeto.

A função *execOrder* representa a ordem de execução das atividades é definida apenas para a primeira atividade da lista de atividades para que seja possível identificar a atividade inicial que o ator (agente) precisa trabalhar. A ordem de execução das demais atividades é calculada pela máquina de estado abstrata avaliando a precedência de recurso de trabalho e produtos de trabalho descrita anteriormente. A precedência da atividade *a1* não é definida, já que sua ordem de execução é definida.

A precedência informada para as demais atividades pode ser modificada com a execução da simulação conforme descrito na seção 4.1.1, já que avalia a disponibilidade do ator para executar uma atividade da fila de atividades, e se produtos de trabalho necessários são atualizados em atividades que sucedem a atividade informada na precedência inicial.

Na Tabela 13 são apresentados os estados que representam o tempo decorrido (*elapsedTime*) das atividades do estudo de caso.

Tabela 13 – Registro dos estados para o tempo decorrido das atividades.

Estado	*elapsedTime(a1)*	*elapsedTime(a2)*	*elapsedTime(a3)*	*elapsedTime(a4)*	*elapsedTime(a5)*	*elapsedTime(a6)*
S0	0	0	0	0	0	0
S1	1	0	0	0	0	0
S2	2	0	0	0	0	0
S3	3	0	0	0	0	0
S4	3	0	0	0	0	0
S5	3	0	0	0	0	0
S6	3	1	0	0	0	0
S7	3	2	0	0	0	0
S8	3	3	0	0	0	0
S9	3	3	0	0	0	0
S10	3	3	0	0	0	0
S11	3	3	1	0	0	0
S12	3	3	2	0	0	0
S13	3	3	2	0	0	0
S14	3	3	2	0	0	0
S15	3	3	2	1	1	0
S16	3	3	2	2	2	0
S17	3	3	2	2	3	0
S18	3	3	2	2	4	0
S19	3	3	2	2	5	0
S20	3	3	2	2	6	0
S21	3	3	2	2	6	0
S22	3	3	2	2	6	0
S23	3	3	2	2	6	1
S24	3	3	2	2	6	2
S25	3	3	2	2	6	3
S26	3	3	2	2	6	4
S27	3	3	2	2	6	5
S28	3	3	2	2	6	5
S29	3	3	2	2	6	5
S30	3	3	2	2	6	5
S31	3	3	2	2	6	5

Fonte: Elaborado pelo autor

No estado S_0 nenhuma atividade iniciou a execução, ou seja, teve o tempo decorrido acrescido de uma unidade de tempo, já que outros valores para as funções apresentados no Quadro 25 são atribuídos a simulação neste estado. A atividade *Outline the Architecture* (a6) inicia sua execução no estado S_{23}, após a atividade *Find and Outline Requirements* (a5). Neste caso a precedência inicial definida na Figura 62 foi alterada com a aplicação das regras de transição, considerando que a atividade *Find and Outline Requirements* (a5) atualiza o documento *Glossary* (wp1), utilizado pela atividade *Outline the Architecture* (a6). A mudança da precedência é realizada no estado S_2 apresentado na Tabela 14.

Tabela 14 – Registro dos estados da precedência.

Estado	*precede(a1)*	*precede(a2)*	*precede(a3)*	*precede(a4)*	*precede(a5)*	*precede(a6)*
S0	undef	a1	a2	a3	a3	a3
S1	undef	a1	a2	a3	a3	a3
S2	undef	a1	a2	a3	a3	a5
S3	undef	a1	a2	a3	a3	a5

Fonte: Elaborado pelo autor

No estado S_2 a condição estabelecida pela regra *MonitoreActivitiesWithUpdateWP* é verdadeira, já que uma atividade (a5) atualiza (*Update*) o documento utilizado na atividade cuja função é aplicada (a6). O Quadro 26 apresenta a regra.

```
MonitoreActivitiesWithUpdateWP ≡
∀ a ∈ Activity :
    if  isNotEmpty(set Update(a)) then
            let (act = FIRST(sequence (ProducedBy (FIRST(sequence (Update (a))))))) in
              if  ExecOrder (act)  >= ExecOrder(a) then
                      Precede(a) := act
                      ExecOrder(a) := ExecOrder (act) + 1
              endif
            endlet
    endif
```

Fonte: Elaborado pelo autor

Quadro 26 – Regra *MonitoreActivitiesWithUpdateWP*

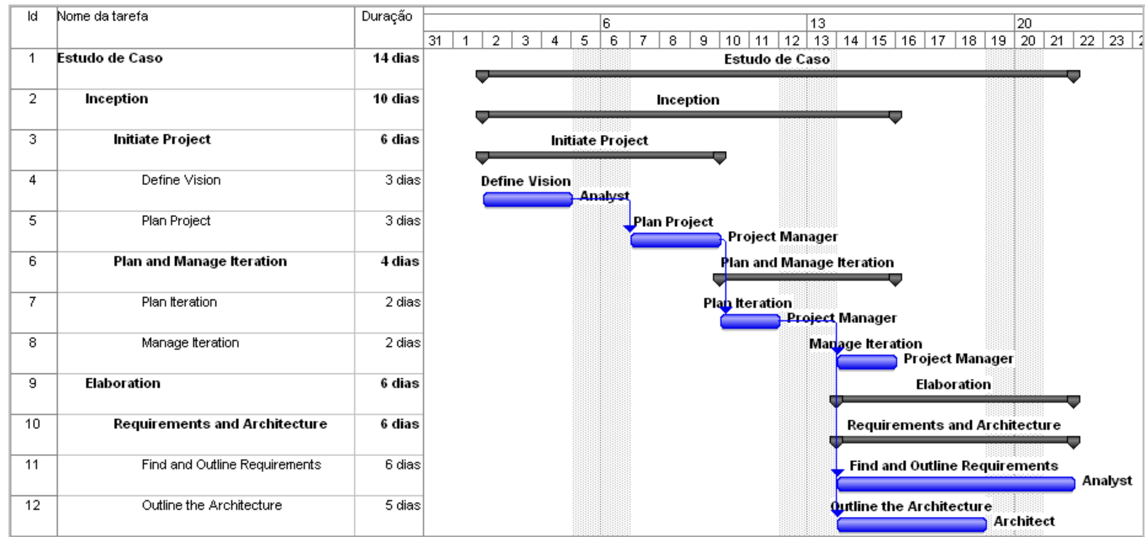
A regra modifica a precedência (função *precede*), estabelecendo como atividade antecessora a atividade que realiza a atualização no documento utilizado pela atividade sucessora. A ordem de execução da atividade *Outline the Architecture* (a6) é modificada pela regra *MonitoreActivitiesWithUpdateWP* no estado , conforme tabela 14.

Tabela 15 – Registro dos estados da ordem de execução.

Estado	*execOrder(a1)*	*execOrder(a2)*	*execOrder(a3)*	*execOrder(a4)*	*execOrder(a5)*	*execOrder(a6)*
S0	1	2	undef	undef	Undef	undef
S1	1	2	3	undef	Undef	undef
S2	1	2	3	4	4	5

Fonte: Elaborado pelo autor

A ordem de execução é informada apenas para a primeira atividade da cadeia de atividades de um projeto, no entanto, considerando a precedência declarada na Figura 62 é possível definir que a atividade *Outline the Architecture* (a6) seria executada em paralelo com as atividades a5 e a4, conforme demonstrado no gráfico de *Gantt* na Figura 63.



Fonte: Elaborado pelo autor

Figura 63 – Gráfico de Gantt do projeto

Na Tabela 16 são apresentados todos os estados que são alcançados pela modificação dos valores das funções da atividade *Define Vision* (a1).

Tabela 16 – Registro dos estados da atividade *Define Vision* da ordem de execução.

Estado	*ET*	*execOrder*	*state*	*state(pr1)*	*state(wp1)*	*state(wp2)*	*state(wp3)*	*TET*
S0	0	1	ACTIVEWD	ALLOCATEDPP	INITIATED	INITIATED	INITIATED	*0*
S1	1	1	ACTIVEWD	ALLOCATEDPP	ONGOING	ONGOING	ONGOING	1
S2	2	1	ACTIVEWD	ALLOCATEDPP	ONGOING	ONGOING	ONGOING	2
S3	3	1	ACTIVEWD	ALLOCATEDPP	ONGOING	ONGOING	ONGOING	3
S4	3	1	FINISHEDWD	AVAILABLEPP	ONGOING	ONGOING	ONGOING	3
S5	3	1	FINISHEDWD	AVAILABLEPP	DONE	DONE	DONE	3

Fonte: Elaborado pelo autor[7]

O ator *Analyst* (pr1) é alocado no estado S_0 da atividade, já que ele inicia o trabalho do projeto e no próximo estado, o tempo decorrido da atividade (ET) é acrescido de uma unidade de tempo. A função *state* para o domínio *WorkProduct* define o ciclo de vida dos produtos de trabalho. A coluna *state(wp1)* apresenta o ciclo de vida do produto de trabalho *Work Items List*. No estado S_1 todos os produtos de trabalho produzidos pela atividade estão em andamento (ONGOING), quando a atividade é finalizada (FINISHEDWD), os produtos de trabalho são concluídos.

Como é a primeira atividade a ser iniciada (poderia haver outras), o TET, tempo total decorrido do projeto também é 1 no estado S_1. A regra *ExecuteActivities* realiza as modificações no tempo decorrido da atividade e no tempo total decorrido no projeto, conforme Quadro 27.

---

[7] ET – e*lapsedTime* (Tempo decorrido da atividade)
TET – totalElapsedTime (Tempo total decorrido do projeto)

ExecuteActivities $\equiv$

$\forall\ a \in ACTIVITY \mid State(a) = ACTIVEWD\ \wedge ElapsedTime(a) < Duration(a)$:

$ElapsedTime(a) := ElapsedTime(a) + 1$

$totalElapsedTime := totalElapsedTime + 1$

Fonte: Elaborado pelo autor

Quadro 27 – Regra *ExecuteActivities* para atividades

A regra determina que toda atividade cujo *state* seja ativo, e o tempo decorrido seja menor que a sua duração, deve ter seu tempo acrescido de 1 (uma) unidade de tempo, como também o tempo total do projeto. As demais regras de transição que atualizam os valores das funções em cada estado foram descritas no Capítulo 5.

Após a simulação realizada neste estudo de caso, o tempo total decorrido do projeto foi alterado para 19 dias (ou unidade de tempo) conforme Figura 64, considerando a modificação de precedências ocorrida na atividade *Outline the Architecture* (a6).

Id	Nome da tarefa	Duração da linha de base	Duração	Variação da duração	Início	Variação do término	Término da linha de base	Término	Predecessoras	Nomes dos recursos
1	**Estudo de Caso**	**14 dias**	**19 dias**	**5 dias**	**Qua 2/1/08**	**5 dias**	**Seg 21/1/08**	**Seg 28/1/08**		
2	**Inception**	**10 dias**	**10 dias**	**0 dias**	**Qua 2/1/08**	**0 dias**	**Ter 15/1/08**	**Ter 15/1/08**		
3	**Initiate Project**	**6 dias**	**6 dias**	**0 dias**	**Qua 2/1/08**	**0 dias**	**Qua 9/1/08**	**Qua 9/1/08**		
4	Define Vision	3 dias	3 dias	0 dias	Qua 2/1/08	0 dias	Sex 4/1/08	Sex 4/1/08		Analyst
5	Plan Project	3 dias	3 dias	0 dias	Seg 7/1/08	0 dias	Qua 9/1/08	Qua 9/1/08	4	Project Manager
6	**Plan and Manage Iteration**	**4 dias**	**4 dias**	**0 dias**	**Qui 10/1/08**	**0 dias**	**Ter 15/1/08**	**Ter 15/1/08**		
7	Plan Iteration	2 dias	2 dias	0 dias	Qui 10/1/08	0 dias	Sex 11/1/08	Sex 11/1/08	5	Project Manager
8	Manage Iteration	2 dias	2 dias	0 dias	Seg 14/1/08	0 dias	Ter 15/1/08	Ter 15/1/08	7	Project Manager
9	**Elaboration**	**6 dias**	**11 dias**	**5 dias**	**Seg 14/1/08**	**5 dias**	**Seg 21/1/08**	**Seg 28/1/08**		
10	**Requirements and Architecture**	**6 dias**	**11 dias**	**5 dias**	**Seg 14/1/08**	**5 dias**	**Seg 21/1/08**	**Seg 28/1/08**		
11	Find and Outline Requirements	6 dias	6 dias	0 dias	Seg 14/1/08	0 dias	Seg 21/1/08	Seg 21/1/08	7	Analyst
12	Outline the Architecture	5 dias	5 dias	0 dias	Ter 22/1/08	6 dias	Sex 18/1/08	Seg 28/1/08	11	Architect

Fonte: Elaborado pelo autor

Figura 64 – Gráfico de Gantt do projeto atualizado com dados da simulação.

A coluna Duração da linha de base apresenta os valores iniciais, definidos na Figura 62 para o projeto. Com a execução da simulação a duração foi atualizada (e representada na coluna Duração), já que a precedência foi modificada. A variação foi de 5 dias e é apresentada na coluna Variação da duração, ocorrida na atividade *Outline the Architecture.*

7.3 Considerações Finais

O estudo de caso realizado utilizando a máquina de estado abstrata obtida neste trabalho pela transformação de modelos demonstrou o registro dos estados e valores das funções ocorridos após a simulação do processo instanciado para um projeto. O registro produzido pode ser utilizado para correção da decisão na alocação de recursos, mas

também como arquivo de entrada para uma ferramenta CASE capaz de mostrar passo a passo o processo em execução, permitindo ao Engenheiro de Processo retroceder, avançar, e obter informações visuais para subsidiar a decisão de realizar correções no modelo do processo ou na alocação de recursos para o projeto.

8 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

O estabelecimento de um modelo de processo em uma organização desenvolvedora de software não é uma tarefa trivial. Envolve especialistas, tecnologia, ferramentas, maturidade da equipe e do modelo de processo adotado e muitas são as pesquisas que envolvem processos de software e abordagens para estabelecimento de padrões, de linguagens de representação, modelagem e simulação (LONCHAMP, 1993), (LAHOZ, 2004), (OMG, 2005), (OMG, 2008b), (PARK, 2007).

A transformação manual foi realizada neste trabalho a partir da definição do meta-modelo fonte, foi estabelecido o modelo fonte. Com a seleção do meta-modelo alvo foi possível estabelecer o modelo de mapeamento dos elementos do meta-modelo fonte para o meta-modelo alvo, alcançando o objetivo específico 3 da pesquisa e possibilitanto a especificação das marcas. O uso das marcas foi essencial para realização do mapeamento de instâncias, passo final para a transformação para o modelo alvo. O modelo de marcas é independente do meta-modelo e do modelo fonte, portanto não alteram seu nível de abstração. Com isso, o objetivo geral, proposto nesta pesquisa, foi atingido.

Diante dos estudos e pesquisas realizadas, foi feita a adaptação de um meta-modelo para definição de processos de software de maneira estática e seu comportamento foi definido em uma máquina de estado abstrata. O meta-modelo originado da adaptação fornece os elementos necessários para que o modelo de processo seja definido de forma estática, atingindo assim o primeiro objetivo específico.

A utilização das máquinas de estado abstratas se mostrou uma abordagem adequada ao problema proposto, já que permite a execução de instâncias de modelos de processos e vai além, permitindo que os aspectos teóricos relacionados com o estabelecimento e uso de processos tenham mais destaque, já que o formalismo matemático é independente de tecnologia, alcançando assim o segundo objetivo específico desta pesquisa. Também por este motivo a implementação gerada pela transformação de modelos pode ser implementada em linguagens de programação ou ainda, utilizada como fonte de especificação requisitos na especificação de ambientes dirigidos a processo. Não é pretensão deste trabalho que a máquina de estado abstrata

para processos de software seja utilizada em ambiente profissional, nem mesmo que traga a solução para a simulação de processos de software.

Ficou claro que a separação de interesses entre o modelo estático e dinâmico do processo fornece uma visão mais clara, independente de tecnologia sobre os conceitos relacionados à definição e a execução (*enactment)* do processo.

O estudo de caso conduzido demonstrou que a definição e o conhecimento acerca das regras de funcionamento de um processo são essenciais no momento da alocação de recursos e validou o modelo alvo gerado pela transformação realizada, concluindo o quarto objetivo específico estabelecido. O estudo de caso foi importante ainda para demonstrar que o modelo de processo deve ser modificado e revisto constantemente, já que se configura a cada projeto de maneira ímpar, e o tempo disponível para avaliar os modelos de processo e propor melhorias é curto, sobretudo nas organizações onde não há melhoria processo e modelos de maturidade estabelecidos.

Acredita-se que uma contribuição importante foi o uso da abordagem MDA para a transformação de modelos de processo de software, permitindo a separação dos modelos em diferentes níveis de abstração, simplificando a definição do mapeamento e transformação do modelo.

Como consideração final sobre as contribuições desta pesquisa, acredita-se que os resultados são relevantes do ponto de vista da teoria de processos de software.

8.1 Sugestões e Trabalhos Futuros

O modelo *SPEMasm* pode ser estendido para permitir a incorporação de outros elementos e conceitos do SPEM, sobretudo da versão 2 (OMG, 2008b). Outros atributos podem ser incorporados e mapeados para máquinas de estado abstratas, para melhorar os resultados da simulação.

As regras de transição definidas formalmente podem ser modificadas e novas regras criadas para suportar a simulação de processos de software em outros contextos.

Uma ferramenta CASE visual para interpretar o registro (*log*) produzido pela simulação pode ser desenvolvida de modo que permita ao Engenheiro de Processo visualizar, parar, avançar, retroceder a simulação de maneira iconográfica. Tal ferramenta pode ser utilizada inicialmente no ambiente acadêmico e servir de protótipo para iniciativas comerciais.

APÊNDICE A – Mapeamento estático

A1. Representação Estática dos elementos do SPEMasm para ASM

Atividades (*Activity*), Produtos de Trabalho (*WorkProduct*), Iteração (*Iteration*) e Fase (*Phase*) são representados como domínios abstratos, já que nenhuma regra ou restrição são estabelecidas fora do modelo proposto na especificação ASM para processos de software.

abstract domain *ACTIVITY*

abstract domain *WORKPRODUCT*

abstract domain *ITERATION*

abstract domain *PHASE*

O conceito de ator que representa um papel no processo (*ProcessRole*) é mapeado neste trabalho como um agente, já que agentes em ASM podem iniciar programas ASM de forma paralela. Desta forma, se um modelo de processo define que existem os papéis de gerente de projeto, programador e analista de sistemas, em sua instanciação para um programa ASM utilizando o mapeamento definido a seguir, cada um destes iniciará um programa ASM de forma paralela, de modo que caso não existam restrições de precedência de atividades, o trabalho possa ser iniciado. Em outras palavras, cada agente instanciado começa a executar as atividades definidas para ele desde que não haja dependência de recursos que impeçam o início desta atividade. Desde modo, o ator é um sub-conjunto ou uma especialização de um agente ASM:

domain $PROCESSROLE \subseteq AGENT$

O domínio enumerado *WorkDefinitionState* é utilizado para controlar o ciclo de vida dos elementos especializados de definição de trabalho, como: *Activity*, *Phase* e *Iteration*. Neste tipo de domínio, um conjunto é definido, de modo que não é possível alterá-los durante a computação da ASM.

enum domain *WORKDEFINITIONSTATE = { WAITINGWD | READYWD | ACTIVEWD | FINISHEDWD }*

Um ator de um processo desempenha papéis durante o trabalho em um determinado projeto de software (instância de processo). Durante o período em que o projeto está em andamento, o ator é alocado para atividades de acordo com suas habilidades e experiência. Ao finalizar determinada atividade, o ator fica disponível para trabalhar em outra atividade, desde que todas as restrições para o início da próxima atividade sejam atendidas. Para representar o ciclo de vida do ator durante a execução (computação) da máquina de estado abstrata, foi definido o domínio enumerado *ProcessPerformerState,* cujos valores são: - *ALOCATEDPP* : quando o ator está trabalhando em uma atividade; - *AVAILABLEPP*: quando o ator está disponível para trabalhar em outra atividade. O estado inicial de todos os atores no início da computação do ASM é *AVAILABLEPP.* A representação do mapeamento é apresentada a seguir:

enum domain *PROCESSPERFORMERSTATE = { AVAILABLEPP | ALLOCATEDPP}*

Durante o andamento do processo de software, os atores que desempenham papéis executando as atividades, produzem artefatos ou produtos de trabalhos que são o resultados propriamente dito destas atividades. Sendo assim, tais produtos de trabalho tem um ciclo de vida definido, desde o início (INITIATED) da sua produção, passando pelo seu desenvolvimento (ONGOING), até a finalização (DONE). Um produto de trabalho que tenha sido produzido por uma atividade, pode ser atualizado por outra (UPDATING). A representação em ASM para o ciclo de vida de produtos de trabalhos é apresentada pela declaração:

enum domain *WORKPRODUCTSTATE = { INITIATED | ONGOING | UPDATING | DONE }*

Os produtos de trabalho desenvolvidos durante um projeto de software tem tipos variados e que são estabelecidos de acordo com o modelo de processo de software definido. Tais tipos podem ser representados por domínios enumerados, como na declaração a seguir, que neste trabalho não possuem semântica definida e são utilizados para representação estática durante a computação da ASM:

enum domain *WORKPRODUCTKIND = { TEXTDOCUMENT | UMLMODEL | EXECUTABLE | CODELIBRARY }*

Definições para o elemento *ProcessRole*

A seguir são apresentadas as definições de atributos, associações e operações do domínio *ProcessRole*.

Atributos

Atributos da UML são representados em ASM por funções estáticas, quando seu valor não é modificado durante a computação (execução) do ASM, ou dinâmicas, quando há modificação de valores (estados) durante a execução.

O atributo State de um ator (*ProcessRole*) é mapeado em uma função dinâmica *State*, cujo valor que define o ciclo de vida, ou o estado é obtido a partir do domínio enumerado *ProcessPerformerState:*

dynamic *State: PROCESSROLE* $\rightarrow$ *PROCESSPERFORMERSTATE*

O estado de um ator no processo é modificado de acordo com a semântica das regras definidas em 4.2.1.

O nome de um ator representa a descrição dada pelo modelo de processo para determinado papel e é representado pela função estática *Name*:

static *Name: PROCESSROLE* $\rightarrow$ *STRING*

Um ator é responsável pela execução de várias atividades durante um projeto de software e no mapeamento realizado, a função estática *Perform* recebe uma seqüência de atividades $(a_1...a_n)$ que o ator deve executar durante a computação do ASM:

static *Perform*: *PROCESSROLE* $\rightarrow$ *ACTIVITY*–**sequence**

Associações

Associações são úteis por permitir que um conceito seja relacionado a outro. No ASM, as associações são representas por funções cuja definição é estabelecida para que valores do domínio mapeado sejam retornados. A função w*orkProduct* definida para

um *ProcessRole* retorna os produtos de trabalho sob a responsabilidade de determinado ator. Sua declaração e definição são apresentadas a seguir:

static *workProduct: PROCESSROLE* $\rightarrow$ *WORKPRODUCT*-**sequence** $\equiv$
  if *isEmpty(Perform(PROCESSROLE)*-**set***)* **then []**
  else
    let(*activity = CHOOSEONE(Perform(PROCESSROLE)))* **in**
      union(*workProduct(PROCESSROLE, Produce(activity))*
    endlet
  endif

Para determinar os produtos de trabalho sob a responsabilidade de determinado ator, a função *workProduct* faz obtém as atividades desempenhadas por um ator e adiciona os produtos de trabalho produzidos para cada uma dessas atividades ao conjunto de produtos de trabalho obtidos pela função.

As atividades sob responsabilidade de um ator são obtidas pela função estática *Activity,* definida a seguir:

static *Activity: PROCESSROLE* $\rightarrow$ *ACTIVITY* –**set** $\equiv$
  Perform(PROCESSROLE)-**set**

Operações

O conceito de operações pode ser mapeado da mesma forma que associações, contudo, operações são definidas neste trabalho como funções auxiliares ao modelo estrutural proposto no *SPEMasm* e são utilizadas como simplificar o funcionamento da máquina de estado abstrata proposto neste trabalho. Algumas regras utilizam tais operações para definir seu escopo, o que simplifica sua abstração no modelo geral proposto.

A operação *ActiveActivities* retorna o conjunto de atividades em andamento no projeto de software, dada a execução da máquina de estado abstrata. A verificação é realizada pelo atributo (função dinâmica) *State* de cada atividade do conjunto

ACTIVITY, retornando um conjunto de atividades cujo estado (*State*) é igual a ativo (*ACTIVEWD*):

static *ActiveActivities*: *PROCESSROLE* $\rightarrow$ *ACTIVITY* –**set** $\equiv$

$\forall\ a \in ACTIVITY{:}\ a \in ACTIVITY \mid State(a) = ACTIVEWD \rightarrow a$

Já a operação *FinishedActivities* retorna o conjunto de atividades finalizadas no projeto de software:

static *FinishedActivities*: *PROCESSROLE* $\rightarrow$ *ACTIVITY* –**set** $\equiv$

$\forall\ a \in ACTIVITY{:}\ a \in ACTIVITY \mid State(a) = FINISHEDWD \rightarrow a$

Definições para o elemento *Activity*

A seguir são apresentadas as definições de atributos e associações do domínio abstrato Activity.

Atributos

A função dinâmica State é mapeada a partir do atributo State do *SPEMasm* e controla o ciclo de vida de uma atividade durante o projeto. Seus valores (estados) possíveis são definidos pelo domínio enumerado *WORKDEFINITIONSTATE.* Os estados são modificados de acordo com o trabalho realizado pelos atores durante o projeto. A definição é apresentada a seguir:

dynamic *State: ACTIVITY* $\rightarrow$ *WORKDEFINITIONSTATE*

Uma atividade possui um nome que descreve o que o ator deve realizar durante um projeto de software. O nome *Name* é definido como uma função estática, já que seu valor (estado) não sofre modificação com a computação realizada pela máquina. Sua definição é apresentada a seguir:

static *Name: ACTIVITY* $\rightarrow$ *STRING*

Para que uma atividade possa ser desenvolvida é importante que o ator tenha conhecimento de sua duração, definida principalmente para projetos de preço fechado.

A duração é uma função estática, já que sua definição é feita somente no início de um projeto de software. Sua definição é representada por:

static *Duration: ACTIVITY* $\rightarrow$ *INTEGER*

A duração apresentada neste trabalho é estimada, mas em outros contextos pode ser negociada de acordo com o andamento do projeto, contudo não é foco deste trabalho definir regras e condições para que tais situações sejam tratadas neste trabalho.

Durante o projeto, o tempo de uma atividade avança a cada unidade de tempo, seja em horas, dias, ou semanas, desta forma, o atributo *ElapsedTime* é mapeado para uma função dinâmica na máquina proposta, já que seu valor muda de acordo com o avanço do projeto.

dynamic *ElapsedTime: ACTIVITY* $\rightarrow$ *INTEGER*

A precedência inicial de uma atividade é definida pela função dinâmica *Precede*, que é definida de forma dinâmica já que além de um valor inicial de precedência definida, seu valor pode mudar de acordo com as precedências definidas na seção 3.1.1.1. Uma atividade é precedida somente por uma atividade, conforme definição:

dynamic *Precede: ACTIVITY* $\rightarrow$ *ACTIVITY*

O atributo *ExecOrder* é utilizado para controlar a ordem de execução e é estabelecido pela máquina de estado abstrata de acordo com as precedências definidas na seção 3.1.1.1. O mapeamento é realizado para a função dinâmica *ExecOrder*, por um número inteiro. Assim, uma atividade pode ter seu ExecOrder definido como 2, significando que é a segunda atividade a ser executada pela máquina, podendo haver uma ou mais atividades com *ExecOrder* com o mesmo valor, neste caso, sendo executadas de forma paralela.

dynamic *ExecOrder: ACTIVITY* $\rightarrow$ *INTEGER*

Associações

As associações de uma atividade para produtos de trabalho podem ser 3 : Produção (*Produce*), Utilização(*Use*) e Atualização (*Update).* Associações de produção (*Produce*) determinam quais produtos de trabalho são desenvolvidos ou produzidos por determinada atividade e são mapeadas como funções estáticas por representarem valores iniciais que devem ser informados como entrada para a máquina de estado abstrata. Sua definição é apresentada a seguir :

static *Produce: ACTIVITY* $\rightarrow$ *WORKPRODUCT–***sequence**

Atividades podem utilizar como entrada documentos, diagramas ou qualquer produto de trabalho que tenha sido desenvolvido em atividades anteriores. Para realizar a atividade de Testes em uma versão de um software para entrega ao cliente, é necessário que o produto de trabalho *Software* tenha sido desenvolvido (produzido) anteriormente e este deve ser utilizado (*use*) como entrada para a atividade de testes. A função *Use* define tal relação, e trata-se de uma função estática, já que seu valor é definido somente uma vez na máquina de estado abstrata e não é modificado durante sua execução (computação). Sua definição é apresentada a seguir:

static *Use: ACTIVITY* $\rightarrow$ *WORKPRODUCT–***sequence**

Um produto de trabalho pode ser atualizado durante um projeto de software. Nas fases iniciais de um projeto de software, a equipe do projeto tem uma visão inicial das necessidades do cliente, e pode produzir uma versão inicial da especificação de requisitos, que pode ser atualizada assim que os requisitos sejam especificados e detalhados pela equipe e o cliente, com o avançar do projeto. Na máquina de estado abstrata proposta, é necessário definir previamente quais produtos de trabalho serão atualizados por cada atividade do projeto de software, e o mapeamento que estabelece tal função estática é apresentado a seguir:

static *Update: ACTIVITY* $\rightarrow$ *WORKPRODUCT–***sequence**

Uma atividade pode ser executada por um ou mais atores, é representada conforme segue:

static *Assistant*: *ACTIVITY* $\rightarrow$ *AGENT* –**set** ≡

$\forall$ *pr* $\in$ *PROCESSROLE: pr* $\in$ *PROCESSROLE | Perform(ACTIVITY)* $\supset$ *pr: pr*

Definições para o elemento *WorkProduct*

A seguir são apresentadas as definições de atributos e associações do domínio abstrato WorkProduct.

Atributos

Assim como os demais domínios da máquina de estado abstrata, o estado de um produto de trabalho também deve ser especificado, de modo que seja possível conhecer seu estado em cada momento do projeto. A função dinâmica *State* é mapeada para conter o estado de um produto de trabalho, cujos valores (estados) são definidos pelo domínio enumerado *WorkProductState*, como segue:

dynamic *State*: *WORKPRODUCT* $\rightarrow$ *WORKPRODUCTSTATE*

Da mesma forma, o nome de um produto de trabalho deve ser definido, para identificar cada um dos produtos de trabalho produzidos, utilizados e atualizados por cada atividade. Tal definição é realizada na máquina de estados abstrata proposta neste trabalho pela função estática *Name*, apresentada a seguir:

static *Name*: *WORKPRODUCT* $\rightarrow$ *STRING*

O atributo *IsDeliverable* do *SPEMasm*, derivado do SPEM (OMG, 2005), mapeado pela função estática *IsDeliverable* é utilizado para especificar se o produto de trabalho em questão deve ser entregue ao cliente (valor *true*), ou se é utilizado durante o projeto como entrada para outra atividade (valor *false*):

static *IsDeliverable*: *WORKPRODUCT* $\rightarrow$ *BOOLEAN*

Diversos são os tipos de produtos de trabalho produzidos durante um projeto de software, dada a natureza do software a ser desenvolvido, como também o modelo d e processo que esteja sendo utilizado. Neste sentido, o tipo (*Kind*) de um produto de trabalho é especificado pela função dinâmica *Kind*, cuja declaração é apresentada a seguir:

static *Kind*: *WORKPRODUCT* $\rightarrow$ *WORKPRODUCTKIND*

Associações

A associação *ResponsibleRole* é mapeada como uma função estática que retorna o ator responsável por determinado produto de trabalho. Sua declaração é apresentada a seguir:

static *ResponsibleRole*: *WORKPRODUCT* $\rightarrow$ *PROCESSROLE*
FIRST(Assistant(FIRST(ProducedBy(WORKPRODUCT))))

As funções estáticas *ProducedBy*, retorna a atividade que produz o produto de trabalho em questão, já as funções *UsedBy* e *UpdatedBy* retornam as atividades que, respectivamente utilizam e atualizam o produto de trabalho. Sua declaração é apresentada a seguir:

static *ProducedBy*: *WORKPRODUCT* $\rightarrow$ *ACTIVITY* –**set** $\equiv$
$\forall a \in ACTIVITY\!: \; a \in ACTIVITY \mid \; Produce(ACTIVITY) \supset w\!: a$

static *UsedBy*: *WORKPRODUCT* $\rightarrow$ *ACTIVITY* –**set** $\equiv$
$\forall a \in ACTIVITY\!: \; a \in ACTIVITY \mid \; Use(ACTIVITY) \supset w\!: a$

static *UpdatedBy*: *WORKPRODUCT* $\rightarrow$ *ACTIVITY* –**set** $\equiv$
$\forall a \in ACTIVITY\!: \; a \in ACTIVITY \mid \; Update(ACTIVITY) \supset w\!: a$

Definições para o elemento *Iteration*

A seguir são apresentadas as definições de atributos e associações do domínio abstrato *Iteration*.

Atributos

Uma iteração é uma especialização de definição de trabalho (*WorkDefinition*), portanto tem seu estado (*State*) controlado pela máquina de estado abstrata e seus possíveis estados são definidos pelo domínio enumerado *WORKDEFINITIONSTATE.* O nome da iteração também deve ser especificado, porém de forma estática, já que seu valor não é modificado pela mudança de estados. Os mapeamentos realizados para o estado e nome de uma iteração é apresentado a seguir:

dynamic *State*: *ITERATION* $\rightarrow$ *WORKDEFINITIONSTATE*

static *Name*: *ITERATION* $\rightarrow$ *STRING*

Associações

A iteração é formada por um conjunto de atividades e seu mapeamento é feito para uma função estática, cujo valor é definido somente uma vez na computação da máquina:

static *Activities*: *ITERATION* $\rightarrow$ *ACTIVITY* –**sequence**

Para que uma iteração seja iniciada, uma pré-condição deve ser estabelecida (OMG, 2005). A determinação da pré-condição deve ser realizada utilizando-se como referência o modelo de processo adotado pela organização ou para um projeto específico e seu ciclo de vida pode ser em cascata, iterativo incremental, espiral, dentre outros. Um projeto instanciado a partir de um modelo de processo, cujo ciclo de vida é o cascata, não tem pré-condições estabelecidas, já que a ordem de execução das atividades é seqüencial e é definida pela função dinâmica *Precede*. Já um modelo iterativo incremental, pode utilizar como pré-condição para início de uma iteração, o estado de produtos de trabalho desenvolvidos anteriormente. Sendo assim, uma iteração $n+1$ pode ser iniciada somente se os produtos de trabalho $w1$ e $w2$ estejam prontos (*DONE*). Tais produtos são produzidos por atividades presentes na iteração n. A definição da pré-condição é uma restrição incluída nas restrições de precedência.

static *Precondition*: *ITERATION* $\rightarrow$ *(WORKPRODUCT* $\times$ *WORKPRODUCTSTATE)*–**sequence**

Ao contrário da pré-condição, que define que uma atividade seja iniciada somente se os critérios sejam atendidos, a meta (*Goal)* estabelece que uma iteração deve atender a critérios para que seja finalizada, ou seja, seu estado seja endereçado como *FINISHEDWD:*

static *Goal*: *ITERATION* $\rightarrow$ *(WORKPRODUCT* $\times$ *WORKPRODUCTSTATE)*–**sequence**

Para simplificar os critérios optou-se por utilizar o estado de produtos de trabalho para definição de pré-condições e meta, contudo, outros critérios como por exemplo, estado de atividades podem ser adicionados ao modelo,

A especificação das funções é realizada, contudo sua implementação na máquina de estado abstrata não é realizada, por limitações encontradas na versão da linguagem utilizada, apresentadas no capítulo do Estudo de Caso.

Definições para o elemento *Phase*

A seguir são apresentadas as definições de atributos e associações do domínio abstrato *Phase*.

Atributos

A semântica da fase é a mesma da iteração já que ambas são especializações de *WorkDefinition*, diferindo apenas por uma fase conter uma ou mais iterações, conforme declaração da associação *Iterations,* mapeada como função estática, já que é definida uma vez na máquina*:*

static *Iterations*: *PHASE* $\rightarrow$ *ITERATION* –**sequence**

Os mapeamentos dos atributos *Name* e *State* são realizados da mesma forma que no domínio *Iteration*, conforme a seguir:

dynamic *State*: *PHASE* $\rightarrow$ *WORKDEFINITIONSTATE*

static *Name*: *PHASE* $\rightarrow$ *STRING*

A semântica do mapeamento realizado para pré-condições e metas no domínio *Iteration* é válido para o domínio *Phase*:

static *Precondition*: *PHASE* $\rightarrow$ *(WORKPRODUCT* $\times$ *WORKPRODUCTSTATE)*–**sequence**

static *Goal*: *PHASE* $\rightarrow$ *(WORKPRODUCT* $\times$ *WORKPRODUCTSTATE)*–**sequence**

Operações

As operações apresentadas nesta seção são utilizadas pela máquina de estado abstrata para controlar a computação da máquina e fornecer informações ao usuário da máquina, ou seja, quem a executa.

A operação *ReadyIterations* é mapeada como uma função estática que retorna um conjunto de iterações de uma fase, cujo estado seja *READYWD,* ou seja, pronto para iniciar. O estado *READYWD* é dado para uma iteração quando todas as suas atividades estão prontas para serem iniciadas.

static *ReadyIterations*: *PHASE* $\rightarrow$ *ITERATION* –**set** $\equiv$

$\forall$ *it* $\in$ *ITERATION: it* $\in$ *ITERATION* | *Iterations(PHASE)* $\supset$ *it* $\wedge$ *State(it)* = *READYWD: it*

As iterações que estão ativas para uma fase são retornadas pela função estática *ActiveIteration,* quando pelo menos uma das atividades da iteração estiver sendo executada por um ator. O mapeamento é apresentado a seguir:

static *ActiveIterations*: *PHASE* $\rightarrow$ *ITERATION* –**set** $\equiv$

$\forall$ *it* $\in$ *ITERATION: it* $\in$ *ITERATION* | *Iterations(PHASE)* $\supset$ *it* $\wedge$ *State(it)* = *ACTIVEWD: it*

O estado de uma iteração de uma fase passa a ser finalizado (*FINISHEDWD*), quando todas as atividades desta iteração são finalizadas com o andamento do projeto. A função estática *FinishedIterations* fornece um conjunto de iterações de determinada fase, cujo estado seja finalizado. O mapeamento é apresentado a seguir:

static *FinishedIterations*: *PHASE* $\rightarrow$ *ITERATION* –**set** $\equiv$

$\forall$ *it* $\in$ *ITERATION: it* $\in$ *ITERATION* | *Iterations(PHASE)* $\supset$ *it* $\wedge$ *State(it)* = *FINISHEDWD: it*

Definições para o escopo das máquinas de estado abstratas

Outros mapeamentos foram realizados para o escopo da máquina de estado abstrata, ou seja, para o vocabulário, independente de qualquer domínio.

A função dinâmica *TotalElapsedTime* é iniciada com o valor 0 (zero), contudo a partir do momento que a máquina inicia sua computação, ou seja, é executada, seu valor é acrescido de uma unidade de tempo, desde a primeira atividade, até a última, somando assim o tempo total do projeto. O mapeamento é apresentado a seguir:

dynamic *TotalElapsedTime*: *INTEGER*

Ainda para fornecer suporte a execução da máquina, a função estática *AllFinishedIterations* retorna todas as iterações que já tenham sido finalizadas, conforme mapeamento:

static *AllFinishedIterations*: *ITERATION* –**set** $\equiv$

$\forall$ *it* $\in$ *ITERATION: it* $\in$ *ITERATION* | *State(it)* = *FINISHEDWD: it*

APÊNDICE B

B1. Modelo Específico de Plataforma na linguagem *AsmetaL* utilizada no estudo de Caso.

```
asm estudoDeCaso

import StandardLibrary

signature:

	// Estrutural

	enum domain WorkDefinitionState = {  WAITINGWD | CANCELEDWD | READYWD | ACTIVEWD
| PAUSEDWD | FAILEDWD | FINISHEDWD }
	enum domain ProcessPerformerState = { AVAILABLEPP | ALLOCATEDPP | UNAVAILABLEPP }
	enum domain WorkProductState = { INITIATED | ONGOING | UPDATING | DONE }
	enum domain WorkProductKind = { TEXTDOCUMENT | UMLMODEL | EXECUTABLE |
CODELIBRARY }

	domain ProcessRole subsetof Agent
	abstract domain Activity
	abstract domain WorkProduct
	abstract domain Iteration
	abstract domain Phase

	dynamic controlled state: ProcessRole ->  ProcessPerformerState
	dynamic controlled name: ProcessRole ->  String

	dynamic controlled totalElapsedTime: Integer

	dynamic controlled state: Activity ->  WorkDefinitionState

	dynamic controlled name: Activity ->  String
	dynamic controlled duration: Activity ->  Integer
	dynamic controlled elapsedTime: Activity ->  Integer
	dynamic controlled precede: Activity ->  Activity
	dynamic controlled execOrder: Activity ->  Integer

	dynamic controlled produce: Activity ->  Seq(WorkProduct)
	dynamic controlled use: Activity ->  Seq(WorkProduct)
	dynamic controlled update: Activity ->  Seq(WorkProduct)

	dynamic controlled state: WorkProduct ->  WorkProductState
	dynamic controlled name: WorkProduct ->  String
	dynamic controlled  isDeliverable: WorkProduct -> Boolean
	dynamic controlled  kind: WorkProduct ->  WorkProductKind

	static  pausedActivities:  Powerset(Activity)
```

```
	static  updateWorkProductList:  Activity -> Powerset(WorkProduct)
	static  assistant: Activity -> Powerset(Agent)

	dynamic controlled state: Iteration ->  WorkDefinitionState
	dynamic controlled name : Iteration -> String
	dynamic controlled activities : Iteration -> Seq(Activity)
	dynamic controlled precondition: Iteration -> 
Seq(Prod(WorkProduct,WorkProductState))
	dynamic controlled goal: Iteration -> Seq(Prod(WorkProduct,WorkProductState))

	dynamic controlled iterations : Phase -> Seq(Iteration)
	dynamic controlled name : Phase -> String
	dynamic controlled state: Phase ->  WorkDefinitionState
	dynamic controlled precondition: Phase -> Seq(Prod(WorkProduct,WorkProductState))
	dynamic controlled goal: Phase -> Seq(Prod(WorkProduct,WorkProductState))

	static  responsibleRole: WorkProduct -> ProcessRole
	static  producedBy: WorkProduct -> Powerset(Activity)
	static  usedBy: WorkProduct -> Powerset(Activity)
	static  updatedBy: WorkProduct -> Powerset(Activity)
	static  workProduct: ProcessRole -> Seq(WorkProduct)
	static  activity: ProcessRole -> Powerset(Activity)
	dynamic controlled  perform: ProcessRole -> Seq(Activity)

	static  activeActivities: Powerset(Activity)
	static  finishedActivities: Powerset(Activity)

	static  readyIterations: Phase -> Powerset(Iteration)
	static  activeIterations: Phase -> Powerset(Iteration)
	static  finishedIterations: Phase -> Powerset(Iteration)

	static  allFinishedIterations: Powerset(Iteration)

	// Instâncias
	static ph1: Phase
	static ph2: Phase

	static it1: Iteration
	static it2: Iteration
	static it3: Iteration

	static pr1: ProcessRole
	static pr2: ProcessRole
	static pr3: ProcessRole

	static a1: Activity
	static a2: Activity
	static a3: Activity
	static a4: Activity
	static a5: Activity
	static a6: Activity
```

```
        static wp1: WorkProduct
        static wp2: WorkProduct
        static wp3: WorkProduct
        static wp4: WorkProduct
        static wp5: WorkProduct
        static wp6: WorkProduct
        static wp7: WorkProduct
        static wp8: WorkProduct
        static wp9: WorkProduct

definitions:

        function activity($pr in ProcessRole) = {$a in Activity | state($a) = WAITINGWD :
$a }

        function activeActivities = {$a in Activity | state($a) = ACTIVEWD : $a }
        function finishedActivities = {$a in Activity | state($a) = FINISHEDWD : $a }

        function readyIterations($ph in Phase) = {$it in Iteration |
contains(iterations($ph), $it) = true and state($it) = READYWD : $it }

        function activeIterations($ph in Phase) = {$it in Iteration |
contains(iterations($ph), $it) = true and state($it) = ACTIVEWD : $it }

        function finishedIterations($ph in Phase) = {$it in Iteration | state($it) =
FINISHEDWD : $it }

        function allFinishedIterations = {$it in Iteration | state($it) = FINISHEDWD :
$it }

        function workProduct($pr in ProcessRole) =
                        produce(first(perform($pr)))

        function responsibleRole($w in WorkProduct) =
                first(asSequence(assistant(first(asSequence(producedBy($w))))))

        function pausedActivities = {$a in Activity| state($a) = PAUSEDWD: $a }

        function updateWorkProductList($a in Activity) = {$w in WorkProduct | state($w) =
UPDATING : $w }

        function assistant($a in Activity) = {$pr in ProcessRole | contains(perform($pr),
$a) = true : $pr}

        function producedBy($w in WorkProduct) = {$a in Activity | contains(produce($a),
$w) = true : $a }

        function updatedBy($w in WorkProduct) = {$a in Activity | contains(update($a),
$w) = true : $a }
```

```
	function usedBy($w in WorkProduct) = {$a in Activity | contains(use($a), $w) =
true : $a }

	// Regras
	rule r_DefineWorkProductParameters =
		forall $w in WorkProduct with isUndef(name($w)) = true  do
		par
			name(wp1) := "Glossary"
			name(wp2) := "Vision"
			name(wp3) := "Work Items List"
			name(wp4) := "Project Plan"
			name(wp5) := "Iteration Plan"
			name(wp6) := "Requirements Specification"
			name(wp7) := "Use Case Description"
			name(wp8) := "Use-Case Model"
			name(wp9) := "Architecture"

			state($w) := INITIATED

			isDeliverable(wp1) := true
			isDeliverable(wp2) := true

			kind(wp1) := TEXTDOCUMENT
			kind(wp2) := UMLMODEL
			kind(wp3) := TEXTDOCUMENT
			kind(wp4) := TEXTDOCUMENT

		endpar

	 rule r_DefineWorkDefinitionParameters =
		forall $a in Activity with isUndef(name($a)) = true  do
		seq
			name(a1) := "Define Vision"
			name(a2) := "Plan Project"
			name(a3) := "Plan Iteration"
			name(a4) := "Manage Iteration"
			name(a5) := "Find and Outline Requirements"
			name(a6) := "Outline the Architecture"

			state($a) := WAITINGWD

			duration(a1) := 3
			duration(a2) := 3
			duration(a3) := 2
			duration(a4) := 2
			duration(a5) := 6
			duration(a6) := 5

			elapsedTime($a) := 0

			execOrder(a1) := 1
```

```
                precede(a1) := undef
     precede(a2) := a1
                precede(a3) := a2
                precede(a4) := a3
                precede(a5) := a3
                precede(a6) := a3

                produce(a1) := [wp1, wp2, wp3]

                produce(a2) := [wp4]
                update(a2):=[wp3]
                use(a2) := [wp2]

                produce(a3) := [wp5]
                update(a3):=[wp3]
                use(a3) := [wp4]

                update(a4):=[wp5, wp4, wp3]

                produce(a5) := [wp6, wp7, wp8]
                update(a5):=[wp1, wp3]
                use(a5) := [wp2]

                produce(a6) := [wp9]
                use(a6) := [wp8, wp7, wp1]

                name(it1):= "Initiate Project"
                name(it2):= "Plan and Manage Iteration"
                name(it3):= "Requirements and Architecture"

                activities(it1) := [a1, a2]
                activities(it2) := [a3, a4]
                activities(it3) := [a5, a6]

                name(ph1):= "Inception"
                name(ph2):= "Elaboration"

                iterations(ph1) := [it2, it1]
                iterations(ph2) := [it3]

                totalElapsedTime := 0
        endseq

rule r_DefineProcessRoleParameters =
        forall $pr in ProcessRole with isUndef(name($pr)) = true do
        seq
                name(pr1) := "Analyst"
                name(pr2) := "Project Manager"
                name(pr3) := "Architect"
```

```
                    perform(pr1):= [a1, a5]
                    perform(pr2):= [a2, a3, a4]
                    perform(pr3):= [a6]

                    state($pr) := AVAILABLEPP
              endseq

       rule r_DefineDefaultOrders =
              forall $a in Activity with isUndef(execOrder($a)) = true do
                     execOrder($a) := execOrder(precede($a)) + 1

       rule r_MonitoreActivitiesWithUseWP =
           forall $a in Activity with true  do
              if isEmpty(asSet(use($a))) = false then
                let ($act = first(asSequence(producedBy(first(use($a)))))) in
                     if (execOrder($act) >= execOrder($a)) = true then
                       seq
                            precede($a) := $act
                            execOrder($a) := execOrder($act) + 1
                       endseq
                     endif
                 endlet
            endif

       rule r_MonitoreActivitiesWithUpdateWP =
           forall $a in Activity with true  do
              if isDef(asSet(update($a))) = false then
                let ($act = first(asSequence(producedBy(first(update($a)))))) in
                   if (execOrder($act) >= execOrder($a)) = true then
                       seq
                            precede($a) := $act
                            execOrder($a) := execOrder($act) + 1
                       endseq
                   endif
                endlet
             endif

       rule r_MonitoreActivitiesPerformed =
         forall $a in activity(self) with
contains(asSequence(activity(first(asSequence(assistant($a))))), $a) = true do
            let ($act = first(asSequence(activity(self)))) in
                if ((execOrder($act) = execOrder($a)) = true and assistant($act) =
assistant($a) = true and eq($a, $act) = false) = true  then
                 seq
                      precede($a) := $act
                      execOrder($a) := execOrder($act) + 1
                 endseq
              endif
           endlet

    rule r_MonitorePhase =
```

```
     forall $ph in Phase with state($ph) = WAITINGWD  do
       if contains(iterations($ph), first(asSequence(readyIterations($ph)))) = true then
         state($ph) := READYWD
       endif

    rule r_MonitoreReadyPhases =
     forall $ph in Phase with state($ph) = READYWD  do
      if contains(iterations($ph), first(asSequence(activeIterations($ph)))) = true then
         state($ph) := ACTIVEWD
      endif

    rule r_MonitoreActivePhases =
     forall $ph in Phase with state($ph) = ACTIVEWD  do
      if eq(first(iterations($ph)), first(asSequence(allFinishedIterations)))= true then
         state($ph) := FINISHEDWD
      endif

    rule r_MonitoreIteration =
     forall $it in Iteration with state($it) = WAITINGWD  do
      if contains(activities($it), first(asSequence(activity(self)))) = true then
          state($it) := READYWD
      endif

    rule r_MonitoreReadyIterations =
     forall $it in Iteration with state($it) = READYWD  do
      if contains(activities($it), first(asSequence(activeActivities))) = true  then
          state($it) := ACTIVEWD
      endif

    rule r_MonitoreActiveIterations =
     forall $it in Iteration with state($it) = ACTIVEWD  do
      if isEmpty(allFinishedIterations) = true then
          if eq(asSet(activities($it)), finishedActivities) = true then
              state($it) := FINISHEDWD
          endif
           else if isEmpty(allFinishedIterations) = false then
              if contains(asSequence(finishedActivities), first(activities($it))) =
true then
                     state($it) := FINISHEDWD
              endif
           endif
      endif

       rule r_ActiveActivities =
              forall $a in Activity with  state($a) = READYWD do
              seq
                     state(first(asSequence(assistant($a)))) := ALLOCATEDPP
                     state($a) := ACTIVEWD
              endseq

       rule r_ExecuteActivities =
```

```
		forall $a in Activity with  state($a) = ACTIVEWD and (elapsedTime($a) <
duration($a)) = true do
		seq
			elapsedTime($a) :=  elapsedTime($a) + 1
			totalElapsedTime :=  totalElapsedTime + 1
		endseq

	rule r_MonitoreWorkingActivities =
		forall $a in Activity with  state($a) = ACTIVEWD and (elapsedTime($a) =
duration($a)) = true  do
		seq
			state($a) := FINISHEDWD
			state(first(asSequence(assistant($a)))) := AVAILABLEPP
		endseq

	rule r_MonitoreInitiatedWorkProducts =
		forall $w in WorkProduct with state($w) = INITIATED and
isEmpty(producedBy($w)) = false do
			if  state(first(asSequence(producedBy($w)))) = ACTIVEWD then
				state($w) := ONGOING
			endif

	rule r_MonitoreInitiatedWorkProducts2 =
		forall $w in WorkProduct with state($w) = INITIATED and
isEmpty(updatedBy($w)) = false do
			if  state(first(asSequence(updatedBy($w)))) = ACTIVEWD then
				state($w) := ONGOING
			endif

	rule r_MonitoreDoneWorkProducts =
		forall  $w in WorkProduct with state($w) = DONE and
isEmpty(updatedBy($w)) = false do
			if state(first(asSequence(updatedBy($w)))) = ACTIVEWD  then
				state($w) := UPDATING
			endif

	rule r_MonitoreUpdatingWorkProducts =
		forall $w in WorkProduct with state($w) = UPDATING and
isEmpty(updatedBy($w)) = false do
			if state(first(asSequence(updatedBy($w)))) = FINISHEDWD  then
				state($w) := DONE
			endif
	rule r_MonitoreOngoingWorkProducts =
		forall $w in WorkProduct with state($w) = ONGOING and
isEmpty(producedBy($w)) = false do
			if  state(first(asSequence(producedBy($w)))) = FINISHEDWD then
				state($w) := DONE
			endif



	// Todos os agentes ProcessRole começam a trabalhar
```

```
	macro rule r_Work =
		forall $a in Activity with state($a) = WAITINGWD do
			if state(self) = AVAILABLEPP  then
				if isUndef(precede($a)) = true  then
					state($a) :=  READYWD
				else if state(precede($a)) = FINISHEDWD then
					state($a) :=  READYWD
					endif
				endif
			endif


	// A duração de todas as atividades deve ser maior que 0
	axiom inv_Activity_duration over duration:
		 (forall $a in Activity with duration($a) > 0)

	// Toda atividade deve ser executada por algum ProcessRole
	axiom inv_Activity_assistant over assistant:
		 (forall $a in Activity with size(assistant($a)) > 0)

	axiom inv_Iteration_activities over activities:
		 (forall $it in Iteration with length(activities($it)) > 0)

	axiom inv_Phase_iterations over iterations:
		 (forall $ph in Phase with length(iterations($ph)) > 0)



main rule r_Main =
seq
		r_DefineWorkProductParameters[]
		r_DefineWorkDefinitionParameters[]
		r_DefineProcessRoleParameters[]
		r_DefineDefaultOrders[]
		r_MonitoreActivitiesWithUseWP[]
		r_MonitoreActivitiesWithUpdateWP[]



	forall $pr in ProcessRole with true do
		program($pr)

		par
			r_ActiveActivities[]
			r_MonitoreIteration[]
			r_MonitoreReadyIterations[]
			r_MonitoreActiveIterations[]
			r_MonitorePhase[]
			r_MonitoreReadyPhases[]
			r_MonitoreActivePhases[]
			r_ExecuteActivities[]
			r_MonitoreWorkingActivities[]
			r_MonitoreInitiatedWorkProducts[]
```

```
                      r_MonitoreInitiatedWorkProducts2[]
                      r_MonitoreOngoingWorkProducts[]
                      r_MonitoreDoneWorkProducts[]
                      r_MonitoreUpdatingWorkProducts[]
                      r_MonitoreActivitiesPerformed[]
               endpar
endseq


default init s0:

       function state($it in Iteration) = WAITINGWD
       function state($ph in Phase) = WAITINGWD


agent ProcessRole : r_Work[]
```

APÊNDICE C

C1. Registros dos estados do estudo de caso[8].

Estado	*S0*	*S1*	*S2*	*S3*	*S4*	*S5*	*S6*	*S7*	*S8*	*S9*
elapsedTime(a1)	0	1	2	3	3	3	3	3	3	3
elapsedTime(a2)	0	0	0	0	0	0	1	2	3	3
elapsedTime(a3)	0	0	0	0	0	0	0	0	0	0
elapsedTime(a4)	0	0	0	0	0	0	0	0	0	0
elapsedTime(a5)	0	0	0	0	0	0	0	0	0	0
elapsedTime(a6)	0	0	0	0	0	0	0	0	0	0
execOrder(a1)	1	1	1	1	1	1	1	1	1	1
execOrder(a2)	2	2	2	2	2	2	2	2	2	2
execOrder(a3)	undef	3	3	3	3	3	3	3	3	3
execOrder(a4)	undef	undef	4	4	4	4	4	4	4	4
execOrder(a5)	undef	undef	4	4	4	4	4	4	4	4
execOrder(a6)	undef	undef	5	5	5	5	5	5	5	5
precede(a1)	undef	undef	undef	undef	undef	undef	undef	undef	undef	undef
precede(a2)	a1	a1	a1	a1	a1	a1	a1	a1	a1	a1
precede(a3)	a2	a2	a2	a2	a2	a2	a2	a2	a2	a2
precede(a4)	a3	a3	a3	a3	a3	a3	a3	a3	a3	a3
precede(a5)	a3	a3	a3	a3	a3	a3	a3	a3	a3	a3
precede(a6)	a3	a3	a5	a5	a5	a5	a5	a5	a5	a5
state(a1)	AWD	AWD	AWD	AWD	FWD	FWD	FWD	FWD	FWD	FWD
state(a2)	WWD	WWD	WWD	WWD	WWD	AWD	AWD	AWD	AWD	FWD
state(a3)	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD
state(a4)	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD
state(a5)	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD
state(a6)	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD
state(it1)	RWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD
state(it2)	undef	undef	undef	undef	undef	RWD	RWD	RWD	RWD	RWD

[8] AWD = ACTIVEWD
WWD – WAITINGWD
RWD – READYWD
FWD – FINISHEDWD
ALPP – ALLOCATEDPP
AVPP – AVAILABLEPP

state(it3)	undef	undef	undef	undef	undef	undef	undef	undef	undef	undef
state(ph1)	undef	RWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD
state(ph2)	undef	undef	undef	undef	undef	undef	undef	undef	undef	undef
state(pr1)	ALPP	ALPP	ALPP	ALPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP
state(pr2)	AVPP	AVPP	AVPP	AVPP	AVPP	ALPP	ALPP	ALPP	ALPP	AVPP
state(pr3)	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP
state(wp1)	I	O	O	O	O	D	D	D	D	D
state(wp2)	I	O	O	O	O	D	D	D	D	D
state(wp3)	I	O	O	O	O	D	U	U	U	U
state(wp4)	I	I	I	I	I	I	O	O	O	O
state(wp5)	I	I	I	I	I	I	I	I	I	I
state(wp6)	I	I	I	I	I	I	I	I	I	I
state(wp7)	I	I	I	I	I	I	I	I	I	I
state(wp8)	I	I	I	I	I	I	I	I	I	I
state(wp9)	I	I	I	I	I	I	I	I	I	I
totalElapsedTime	*0*	1	2	3	3	3	4	5	6	6

Estado	*S10*	*S11*	*S12*	*S13*	*S14*	*S15*	*S16*	*S17*	*S18*	*S19*
elapsedTime(a1)	3	3	3	3	3	3	3	3	3	3
elapsedTime(a2)	3	3	3	3	3	3	3	3	3	3
elapsedTime(a3)	0	1	2	2	2	2	2	2	2	2
elapsedTime(a4)	0	0	0	0	0	1	2	2	2	2
elapsedTime(a5)	0	0	0	0	0	1	2	3	4	5
elapsedTime(a6)	0	0	0	0	0	0	0	0	0	0
execOrder(a1)	1	1	1	1	1	1	1	1	1	1
execOrder(a2)	2	2	2	2	2	2	2	2	2	2
execOrder(a3)	3	3	3	3	3	3	3	3	3	3
execOrder(a4)	4	4	4	4	4	4	4	4	4	4
execOrder(a5)	4	4	4	4	4	4	4	4	4	4
execOrder(a6)	5	5	5	5	5	5	5	5	5	5
precede(a1)	undef	undef	undef	undef	undef	undef	undef	undef	undef	undef
precede(a2)	a1	a1	a1	a1	a1	a1	a1	a1	a1	a1
precede(a3)	a2	a2	a2	a2	a2	a2	a2	a2	a2	a2
precede(a4)	a3	a3	a3	a3	a3	a3	a3	a3	a3	a3
precede(a5)	a3	a3	a3	a3	a3	a3	a3	a3	a3	a3
precede(a6)	a5	a5	a5	a5	a5	a5	a5	a5	a5	a5
state(a1)	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(a2)	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(a3)	AWD	AWD	AWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(a4)	WWD	WWD	WWD	WWD	AWD	AWD	AWD	FWD	FWD	FWD
state(a5)	WWD	WWD	WWD	WWD	AWD	AWD	AWD	AWD	AWD	AWD
state(a6)	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD	WWD
state(it1)	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD

state(it2)	RWD	AWD	AWD	AWD	FWD	FWD	FWD	FWD	FWD	FWD
state(it3)	undef	undef	undef	undef	RWD	RWD	RWD	RWD	AWD	AWD
state(ph1)	AWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD
state(ph2)	undef	undef	undef	undef	undef	RWD	RWD	RWD	RWD	AWD
state(pr1)	AVPP	AVPP	AVPP	AVPP	ALPP	ALPP	ALPP	ALPP	ALPP	ALPP
state(pr2)	ALPP	ALPP	ALPP	AVPP	ALPP	ALPP	ALPP	AVPP	AVPP	AVPP
state(pr3)	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP
state(wp1)	D	D	D	D	D	U	U	U	U	U
state(wp2)	D	D	D	D	D	D	D	D	D	D
state(wp3)	D	D	D	D	D	D	D	D	D	D
state(wp4)	D	D	D	D	D	U	U	U	D	D
state(wp5)	I	O	O	O	D	U	U	U	D	D
state(wp6)	I	I	I	I	I	O	O	O	O	O
state(wp7)	I	I	I	I	I	O	O	O	O	O
state(wp8)	I	I	I	I	I	O	O	O	O	O
state(wp9)	I	I	I	I	I	I	I	I	I	I
totalElapsedTime	6	7	8	8	8	9	10	11	12	13

Estado	*S20*	*S21*	*S22*	*S23*	*S24*	*S25*	*S26*	*S27*	*S28*	*S29*
elapsedTime(a1)	3	3	3	3	3	3	3	3	3	3
elapsedTime(a2)	3	3	3	3	3	3	3	3	3	3
elapsedTime(a3)	2	2	2	2	2	2	2	2	2	2
elapsedTime(a4)	2	2	2	2	2	2	2	2	2	2
elapsedTime(a5)	6	6	6	6	6	6	6	6	6	6
elapsedTime(a6)	0	0	0	1	2	3	4	5	5	5
execOrder(a1)	1	1	1	1	1	1	1	1	1	1
execOrder(a2)	2	2	2	2	2	2	2	2	2	2
execOrder(a3)	3	3	3	3	3	3	3	3	3	3
execOrder(a4)	4	4	4	4	4	4	4	4	4	4
execOrder(a5)	4	4	4	4	4	4	4	4	4	4
execOrder(a6)	5	5	5	5	5	5	5	5	5	5
precede(a1)	undef	undef	undef	undef	undef	undef	undef	undef	undef	undef
precede(a2)	a1	a1	a1	a1	a1	a1	a1	a1	a1	a1
precede(a3)	a2	a2	a2	a2	a2	a2	a2	a2	a2	a2
precede(a4)	a3	a3	a3	a3	a3	a3	a3	a3	a3	a3
precede(a5)	a3	a3	a3	a3	a3	a3	a3	a3	a3	a3
precede(a6)	a5	a5	a5	a5	a5	a5	a5	a5	a5	a5
state(a1)	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(a2)	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(a3)	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(a4)	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(a5)	AWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(a6)	WWD	WWD	AWD	AWD	AWD	AWD	AWD	AWD	FWD	FWD

state(it1)	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(it2)	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(it3)	AWD	AWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(ph1)	AWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD	AWD
state(ph2)	AWD	AWD	AWD	FWD	FWD	FWD	FWD	FWD	FWD	FWD
state(pr1)	ALPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP
state(pr2)	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP	AVPP
state(pr3)	AVPP	AVPP	ALPP	ALPP	ALPP	ALPP	ALPP	ALPP	AVPP	AVPP
state(wp1)	U	U	D	D	D	D	D	D	D	D
state(wp2)	D	D	D	D	D	D	D	D	D	D
state(wp3)	D	D	D	D	D	D	D	D	D	D
state(wp4)	D	D	D	D	D	D	D	D	D	D
state(wp5)	D	D	D	D	D	D	D	D	D	D
state(wp6)	O	O	D	D	D	D	D	D	D	D
state(wp7)	O	O	D	D	D	D	D	D	D	D
state(wp8)	O	O	D	D	D	D	D	D	D	D
state(wp9)	I	I	I	O	O	O	O	O	O	D
totalElapsedTime	14	14	14	15	16	17	18	19	19	19

Estado	*S30*	*S31*
elapsedTime(a1)	3	3
elapsedTime(a2)	3	3
elapsedTime(a3)	2	2
elapsedTime(a4)	2	2
elapsedTime(a5)	6	6
elapsedTime(a6)	5	5
execOrder(a1)	1	1
execOrder(a2)	2	2
execOrder(a3)	3	3
execOrder(a4)	4	4
execOrder(a5)	4	4
execOrder(a6)	5	5
precede(a1)	undef	undef
precede(a2)	a1	a1
precede(a3)	a2	a2
precede(a4)	a3	a3
precede(a5)	a3	a3
precede(a6)	a5	a5
state(a1)	FWD	FWD
state(a2)	FWD	FWD
state(a3)	FWD	FWD
state(a4)	FWD	FWD
state(a5)	FWD	FWD
state(a6)	FWD	FWD

state(it1)	FWD	FWD
state(it2)	FWD	FWD
state(it3)	FWD	FWD
state(ph1)	AWD	AWD
state(ph2)	FWD	FWD
state(pr1)	AVPP	AVPP
state(pr2)	AVPP	AVPP
state(pr3)	AVPP	AVPP
state(wp1)	D	D
state(wp2)	D	D
state(wp3)	D	D
state(wp4)	D	D
state(wp5)	D	D
state(wp6)	D	D
state(wp7)	D	D
state(wp8)	D	D
state(wp9)	D	D
totalElapsedTime	19	19

REFERÊNCIAS BIBLIOGRÁFICAS

ASMETA. Disponível em: http://asmeta.sourceforge.net/index.html. Acesso em: 05 out. 2007.

BENDRAOU, Réda et al. Definition of an Executable SPEM 2.0. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, 14., 2007, Nagoya. **Proceedings of the 14th Asia-Pacific Software Engineering Conference**.Washington: Ieee Computer Society, 2007. p. 390 - 397.

BÖRGER, Egon; STÄRK, Robert. **Abstract State Machines:** A Method for High-Level System Design and Analysis. Berlin: Springer-verlag, 2003. 481 p.

CALIARI, Giuliano Luz Pigatti.**Transformações e mapeamentos da MDA e sua implementação em três ferramentas.** 2007. 2 f. Dissertação (Mestrado) - Curso de Engenharia, Universidade de São Paulo, São Paulo, 2007. Cap. 2.

CAVARRA; A., RICCOBENE, E.; SCANDURRA, P.: **A framework to simulate UML models: moving from a semi-formal to a formal environment**. SAC 2004: 1519-1523

COREASM. Disponível em: http://www.coreasm.org/. Acesso em: 19 set. 2007.

CUNHA, Leonardo M.; LUCENA, Carlos J. P. de. **Proposta de Um Método para o Desenvolvimento de Aplicações para a Web Semântica Utilizando MDA**. PUC-Rio, 2004.

DI IORIO, Vladimir Oliveira. **Avaliação Parcial de Máquinas de Estado Abstratas.** 2001. 163 f. Tese (Doutorado) - Curso de Ciência Da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, 2001. Cap. 2.

DI NITTO, E., LAVAZZA, L., SCHIAVONI, M., TRACANELLA, E., TROMBETTA, M. **Deriving executable process descriptions from UML**. In: Proceedings of the 24th international Conference on Software Engineering . Orlando, Florida: ICSE, 2002. p. 155-165.

DUC, Bui Minh. **Real-Time Object Uniform Design.** Quebec: Springer, 2007. 480 p.

ECLIPSE. Disponível em: http://www.eclipse.org/. Acesso em: 05 set. 2006.

FAVRE, Jean-Marie. **Towards a Basic Theory to Model Driven Engineering**, Université Joseph Fourier, Grenoble, France, 2004.

FUGGETTA, A**. Software process: a roadmap**. In: Proceedings of the Conference on the Future of Software Engineering(Limerick, Ireland, June 04 - 11, 2000). New York, ICSE '00. ACM, 2000. p.25-34.

GARGANTINI, A.; RICCOBENE, E.; SCANDURRA, P. **A Metamodel-based Simulator for ASMs** in The 14th International ASM Workshop, Grimstad, Norwegen, May 2008 (Eds. Andreas Prinz) (2007)

GARGANTINI, A.; RICCOBENE, E.; SCANDURRA, P. **AsmEE: an Eclipse plug-in in a metamodel based framework for the Abstract State Machines** in First International Conference on Eclipse Technologies ECLIPSE-IT 2007Cuzzolin Editore (2007).

GARGANTINI, Angelo; RICCOBENE, Elvinia; SCANDURRA, Patrizia. **Metamodelling a Formal Method:** Applying MDE to Abstract State Machines. Technical. 97. ed. Milão: University Of Milan, 2006. 52 p.

GRIESKAMP, W., NACHMANSON, L., TILLMANN, N., VEANES, M., **Test Case Generation from AsmL Specifications - Tool Overview**, 10th International Workshop on Abstract State Machines, 2003.

GUREVICH, Y.; **Evolving Algebras** 1993: Lipari Guide. In E. Börger, editor, Specification and Validation Methods, pages 9–36. Oxford University Press, 1995.

IBM (Org.). **OpenUP/Basic.** Disponível em: <http://epf.eclipse.org/wikis/openup/>. Acesso em: 12 nov. 2008.

KLEPPE, Anneke; WARMER, Jos; BAST, Wim. **MDA Explained - The Model Driven Architecture:** Practice and Promise. Addison Wesley, 2003. 192 p.

KROLL, Per; MACISAAC, Bruce. **Agility and Discipline Made Easy**: Practices from OpenUP and RUP: Addison-wesley, 2006. 448 p. (Professional).

KRUCHTEN, Philippe. **The Rational Unified Process**: An Introduction. 3. ed: Addison-wesley, 2003. 336 p. (Professional).

LAHOZ, Carlos Henrique N. **Uma abordagem para a gerência da qualidade em um ambiente de engenharia de software centrado em processo**. INPE, Dissertação de Mestrado, 2004.

LONCHAMP, J. **A Structured Conceptual and Terminological Framework for Software Process Engineering**. In: International Conference on the Software Process, 2. Proceedings... Berlin: IEEE Press, 1993

LOPES, D.; HAMMOUDI, S.; BÉZIVIN, J.; JOUAULT, F. **Mapping Specification in MDA: from Theory to Practice**. Proceedings of the First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA'2005). 2005.

MAUDE . Disponível em: http://maude.cs.uiuc.edu/papers/. Acesso em: 12 abr. 2007.

OBER, Ileana Stan. **Harmonizing Design Languages with Object-Oriented Extensions and an Executable Semantics.** 2001. 207 f. Tese (Doutorado) - Institut National Polytechnique de Toulouse, 2001. Cap. 8.

OMG. **Business Process Modeling Notation (BPMN) - version 1.1**. 2008. Disponível em: <http://www.bpmn.org/Documents/BPMN%201-1%20Specification.pdf>.

OMG. **Software Process Engineering Metamodel Specification (SPEM) - version 1.1**. 2005. Disponível em: < www.omg.org/docs/formal/05-01-06.pdf >.

OMG. **Software Process Engineering Metamodel Specification (SPEM)** - version 2. 2008. Disponível em: < www.omg.org/cgi-bin/apps/doc?formal/08-04-01.pdf >.

OMG. **Unified Modeling Language: Infrastrutucture - version 2.2.** 2008. Disponível em: < http://www.omg.org/docs/ptc/08-05-04.pdf>.

OMG. **Unified Modeling Language: Superstructure - version 2.1.1**. 2006. Disponível em: < http://www.omg.org/cgi-bin/apps/doc?formal/07-02-05.pdf>.

OMG; **MDA Guide** Version 1.0.1. 2003. Disponível em: <www.omg.org/cgi-bin/doc?omg/03-06-01>

PARK, SeungHun et al. **Deriving Software Process Simulation Model from SPEM-based Software Process Model**. APSEC 2007: 382-389

PENKER, Magnus; ERIKSSON, Hans-erik. **Business Modeling With UML**: Business Patterns at Work (Paperback): Wiley, 2000. 480 p.

REIS, Carla Alessandra. **Uma Abordagem Flexível para Execução de Processos de Software Evolutivos.** 2003. 267 f. Tese (Doutorado) - Curso de Ciência Da Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2003.

RUÍZ, F., VIZCAÍNO, A., GARCÍA, F., PIATTINI, M. **Using XMI and MOF for Representation and Interchange of Software Processes**. In: Proceedings of the 14th international Workshop on Database and Expert Systems Applications. Washington: IEEE Computer Society, 2003. p. 739.

RUMBAUGH, J; JACOBSON, I ; BOOCH, G. **UML : Guia do Usuário**. Rio de Janeiro, Campus, 2000. (ISBN: 8535205624).

SENDALL, S. **Combining Generative and Graph Transformation Techniques for Model Transformation: An Effective Alliance?** Workshop on Generative Techniques in the context of Model Driven Architecture, 2nd OOPSLA , Anaheim. 2003.

SHEN, W; LOW, W. L. "**Using Abstract State Machines to Support UML Model Instantiation Checking**," in *Proceeding of The LASTED International Conference on SOFTWARE ENGINEERING*, Feb., 2005.

SOMMERVILLE, Ian. **Engenharia de Software**. 6. ed. São Paulo: Pearson Education, 2003. 585 p.

STANDISH GROUP. **Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%**. Disponível em: <http://www.standishgroup.com/press/article.php?id=2>. Acesso em 15. Jun. 2007.

STÄRK, R. F., SCHMID, J., BÖRGER, E. **Java and the Java Virtual Machine Definition, Verification, Validation**. ISBN: 978-3-540-42088-0, 2001.

OUIMET, M., LUNDQVIST, K. **The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering**. In: Proceedings of the 14th International Workshop on Abstract State Machines (ASM '07), Grimstad, Norway, 2007.

VALENTE, M. T. O; BIGONHA, R. S.; MAIA, M. A.; LOUREIRO, A. A. F. **Aplicação de ASM na Especificação de Sistemas Móveis**. II Workshop de Métodos Formais (WMF), p. 60-69. 1999.