UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Dissertação de Mestrado

MARISA-MDD: Uma Abordagem para Transformações entre Modelos Orientados a Aspectos: dos Requisitos ao Projeto Detalhado

Ana Luisa Ferreira de Medeiros

Natal/RN
Junho de 2008

ANA LUISA FERREIRA DE MEDEIROS

MARISA-MDD: Uma Abordagem para Transformações entre Modelos Orientados a Aspectos: dos Requisitos ao Projeto Detalhado

Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como requisito para a obtenção do grau de Mestre em Sistemas e Computação (MSc.).

Profª. Drª. Thaís Vasconcelos Batista

Orientadora

Natal/RN

Junho de 2008

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial
Especializada do Centro de Ciências Exatas e da Terra – CCET.

Medeiros, Ana Luisa Ferreira de.

MARISA-MDD : uma abordagem para transformações entre modelos orientados a aspectos : dos requisitos ao projeto detalhado / Ana Luisa Ferreira de Medeiros. – Natal, 2008.

104 f. : il.

Orientadora: Profa. Dra. Thaís Vasconcelos Batista.
Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Desenvolvimento de software – Dissertação. 2. Modelos de requisitos – Dissertação. 3. Modelos orientados a aspectos – Dissertação. 4. Arquitetura de software – Dissertação. 5. Projeto detalhado – Dissertação. I. Batista, Thaís Vasconcelos. II. Título

RN/UF/BSE-CCET CDU: 681.3.06

“O correr da vida embrulha tudo.
A vida é assim: esquenta, e esfria, aperta, daí afrouxa;
sossega e depois desinquieta.
O que ela quer da gente é CORAGEM”.
(José Guimarães Rosa)

Agradecimentos

A Deus, por sempre ter me dado força superar todas as dificuldades e fraquezas as quais me deparei durante todo o mestrado.

A meus pais, Ana Beatriz e Lamech, que sempre me apoiaram em todos os momentos, e que são a base e a razão de tudo em minha vida. E, principalmente, por entenderem a minha ausência e distância.

A Thais, pelas orientações sempre precisas, pela segurança, competência e confiança que tem e me passou nos momentos que mais necessitei. Pelos conhecimentos fornecidos, pelas muitas oportunidades dadas, pela paciência, atenção, amizade, apoio e compreensão.

A Minora pela amizade e incentivo. E por ter me dado as primeiras oportunidades profissionais, e confiado no meu trabalho.

A Lyrene pela atenção, pelos muitos contribuições e aprendizados fornecidos.

A Liginha, que sempre me apoiou, incentivou. Pela sua amizade e companheirismo e força em TODOS os momentos ao longo do mestrado e da minha vida em si.

Aos meus amigos, Rafinha, Camilla, Carla, Guilherme, Clarissa, Ninho, Diego e tantos outros, por estarem presentes as horas de dificuldades e alegrias.

Aos amigos do Projeto GEDIG, Fabíola, Lirisnei, Fred e Raul, pelo companheirimo e pelo apoio nesse período do mestrado. E, não poderia deixar de agradecer a Leonardo, pela compreensão que teve em entender essa fase de elaboração de Dissertação.

A instituição UFRN e seus professores, pelo ensino, aprendizado e pelo amadurecimento que obtive enquanto pessoa e profissional.

RESUMO

As abordagens orientadas a aspectos relacionadas a diferentes atividades do processo de desenvolvimento de software são, em geral, independentes e os seus modelos e artefatos não estão alinhados ou inseridos em um processo coerente. No desenvolvimento orientado a modelos, os diversos modelos e a correspondência entre eles são especificados com rigor. Com a integração do desenvolvimento orientado a aspectos (DSOA) e o desenvolvimento baseado em modelos (MDD) pode-se automaticamente propagar modelos de uma atividade para outra atividade, evitando a perda de informações e de decisões importantes estabelecidas em cada atividade. Este trabalho apresenta MARISA-MDD, uma estratégia baseada em modelos que integra as atividades de requisitos, arquitetura e projeto detalhado orientado a aspectos, usando as linguagens AOV-graph, AspectualACME e aSideML, respectivamente. MARISA-MDD define, para cada atividade, modelos representativos (e metamodelos correspondentes) e um conjunto de transformações entre os modelos de cada linguagem. Tais transformações foram especificadas e implementadas em ATL (*Atlas Definition Language*), no ambiente Eclipse. MARISA-MDD permite a propagação automática entre modelos AOV-graph, AspectualACME e aSideML. Para validar a abordagem proposta dois estudos de caso, o *Health Watcher* e o *Mobile Media* foram usados no ambienteMARISA-MDD para geração automática dos modelos AspectualACME e aSideML, a partir do modelo AOV-graph.

Palavras-chaves: Desenvolvimento de Software Orientado a Aspectos, Desenvolvimento de Software Orientado a Modelos, Modelos de Requisitos, Arquitetura de Software, Projeto Detalhado, Transformações.

ABSTRACT

Aspect Oriented approaches associated to different activities of the software development process are, in general, independent and their models and artifacts are not aligned and inserted in a coherent process. In the model driven development, the various models and the correspondence between them are rigorously specified. With the integration of aspect oriented software development (DSOA) and model driven development (MDD) it is possible to automatically propagate models from one activity to another, avoiding the loss of information and important decisions established in each activity. This work presents MARISA-MDD, a strategy based on models that integrate aspect-oriented requirements, architecture and detailed design, using the languages AOV-graph, AspectualACME and aSideML, respectively. MARISA-MDD defines, for each activity, representative models (and corresponding metamodels) and a number of transformations between the models of each language. These transformations have been specified and implemented in *ATL (Atlas Definition Language),* in the Eclipse environment. MARISA-MDD allows the automatic propagation between AOV-graph, AspectualACME, and aSideML models. To validate the proposed approach two case studies, the Health Watcher and the Mobile Media have been used in the MARISA-MDD environment for the automatic generation of AspectualACME and aSideML models, from the AOV-graph model.

Keywords: Aspect-Oriented Software Development, Model Driven Software Development, Requirements Models, Software Archictecture, Detailed Design, Transformations.

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

Apesar dos inúmeros avanços da Engenharia de Software, ainda há discussões a respeito da qualidade dos resultados produzidos pelas atividades do processo de desenvolvimento de software (elicitação e análise de requisitos, arquitetura, projeto detalhado, implementação, testes, etc), e do quão essas atividades estão integradas, a fim de permitir uma maior manutenibilidade e evolução dos artefatos produzidos, preservação das informações na passagem de uma atividade para outra, e a identificação de características que se encontram transversais ao longo das atividades.

Nos últimos anos, o *Desenvolvimento de Software Orientado a Aspectos* (DSOA) (FILMAN ET AL., 2005) vem se consolidando como uma alternativa viável para a modularização e composição de características transversais ao longo de todo o processo de desenvolvimento e evolução de software. As características transversais são partes de um sistema que estão entrelaçadas e/ou sobrepostas aos elementos básicos do sistema, influenciando ou restringindo umas as outras, tornando o sistema complexo e difícil de analisar. Isso se reflete no número crescente de abordagens e inovações orientadas a aspectos associadas às mais diversas atividades do processo de desenvolvimento de software, que buscam através do uso de aspectos (ou conceitos similares) melhorar a compreensão, modularidade, reusabilidade e evolução do software (AOSD, 2008). Entretanto, há diversas barreiras e desafios a serem superados até que aspectos sejam adotados longamente pela indústria de software e as tecnologias de DSOA tornem-se parte dos processos de software típicos.

Todavia essas abordagens orientadas a aspectos são independentes e os novos modelos e artefatos gerados não estão naturalmente alinhados ou inseridos em um processo coerente. Como conseqüência, desenvolvedores devem não apenas conhecer diversas abordagens ao, modelos e artefatos associados, mas também: (i) compreender como aspectos materializam-se em cada um deles e (ii) definir uma correspondência entre artefatos OA gerados em cada atividade, por cada abordagem adotada. Além disso, se a correspondência entre modelos e artefatos OA associados às diversas atividades não for bem definida, pode-se perder informações importantes ou mesmo introduzir erros de uma atividade para outra. Uma possível solução para estas questões é adoção de uma abordagem orientada a modelos, onde os vários modelos OA e a correspondência entre eles possam ser especificados com rigor.

Os modelos fornecem abstrações e visões para representar diferentes elementos de diversas camadas de abstração. Dessa forma, eles podem representar cada atividade do processo de desenvolvimento de software. Assim, para integrar essas diversas atividades, podemos derivar modelos a partir de outros modelos através das transformações e especificação de regras de transformações, o que resulta no mapeamento entre diferentes visões e interesses de um sistema.

A especificação de modelos e a transformação de tais modelos em outros modelos ou artefatos de software é a essência da abordagem de desenvolvimento de software orientado a modelos (*Model-driven Development* - MDD) (STAHL ET AL., 2006). Os modelos associados a diferentes atividades do processo de software são construídos com base em rigores semânticos especificados em alguma linguagem de descrição de modelos. Dessa forma, a integração entre tais atividades pode ser feita através da especificação de regras de transformação que definem como um modelo pode ser derivado a partir de outro. O uso de uma abordagem baseada em transformação entre modelos pode fornecer vários benefícios: (i) modularização e reuso das transformações, (ii) transformação de forma automatizada entre modelos, tornando o processo de integração menos suscetível a erros, (iii) possibilidade de propagar informações entre as diversas etapas de desenvolvimento, bem como navegar de uma etapa para outra com a facilidade de rastrear as informações provenientes de outras etapas de desenvolvimento, (iv) diminuição do *gap* entre os modelos envolvidos na transformação; (v) diminuição do retrabalho de modelagem e modificação; dentre outras.

Atualmente existem alguns trabalhos que integram DSOA e MDD (SANCHÉZ ET AL., 2006) (SIMMONDS ET AL., 2005), (ALVEZ ET AL., 2007), (WALDERHAUG ET AL., 2006) e (AMAYA; GONZÁLES; MURRILLO, 2006). No entanto, a maioria não visa agregar as fases de requisitos, arquitetura e projeto detalhado, não apresentam as especificações e rigor semântico na definição do processo de transformações MDD, nem propõe alternativas de ferramentas que auxiliem e automatizem esse processo.

O enfoque do presente trabalho é nas fases iniciais do desenvolvimento de software orientado a aspectos (*Early Aspects)* (FILMAN ET AL., 2005) visando integrar e comunicar as fases requisitos, arquitetura de software e projeto detalhado utilizando a representatividade de modelos e recursos para transformações MDD. Adicionalmente, a idéia é disponibilizar um ambiente integrado que automatize o mapeamento entre essas fases e os seus relacionamentos, tornando o processo menos

repetitivo, simplificado e menos passível de erros.

1.1. Motivação

A engenharia de requisitos (KONTOYA; SOMMERVILLE, 1998) (SOMMERVILLE, 1997), arquitetura de software (SHAW; GARLAN, 1996), e Projeto detalhado (SWEBOK, 2000) são fases do processo de desenvolvimento de software que têm sido recentemente aprimoradas com abstrações orientadas a aspectos (BASS ET AL, 2003). A integração e transformação entre elementos das fases de requisitos, arquitetura de software, e projeto detalhado são necessárias para gerenciar decisões tomadas em cada um destas etapas, e garantir a correspondência e a diminuição da distância entre modelos e artefatos produzidos durante estas fases. Além disso, essa transformação reduz o esforço na geração de uma arquitetura inicial derivada de um modelo de requisitos, manutenção das informações desde os requisitos até as etapas mais avançadas do ciclo de desenvolvimento de software, reduz a repetição do trabalho de modelagem nas diferentes fases, diminui o *gap* entre as informações dos modelos construídos nessas etapas, possibilita a rastreabilidade de elementos dos modelos, dentre eles as características transversais, facilita a localização de mudanças entre as etapas de desenvolvimento e diminui o trabalho repetitivo relacionado a mudanças nas especificações e modelos. Adicionalmente, evita a perda de informações e decisões importantes estabelecidas nas fases iniciais do desenvolvimento de software. Nessa direção, elaboramos diversos trabalhos para integração dessas atividades de desenvolvimento de software, descritos a seguir.

Em (SILVA ET AL., 2007), foi definido um mapeamento denominado *simbiôntico* entre requisitos e arquitetura, representados por um modelo de requisitos orientados a aspectos, AOV-graph (SILVA, 2006), e uma linguagem de descrição arquitetural orientada a aspectos, AspectualACME (BATISTA ET AL., 2006) e apresentamos as regras de mapeamento entre esses dois modelos orientados a aspectos. O mapeamento simbiôntico definido possibilita considerar que requisitos e arquitetura sejam artefatos associados a atividades que se beneficiam uma da outra, de forma que seja possível propagar mudanças entre estes artefatos e navegar de uma fase para outra.

Em (MEDEIROS ET AL., 2007a), foi acrescentado ao mapeamento simbiôntico o conceito de *sinergia*, que significa que, além dos elementos das atividades de

requisitos e projeto arquitetural beneficiarem-se um do outro para alcançar suas metas específicas, existe uma ação sinérgica entre eles, fazendo com que haja possibilidade de mapeamento bi-direcional e o efeito resultante dessa sinergia seja maior que a soma de seus efeitos individuais. Além disso, apresentamos as regras de mapeamento entre AspectualACME e AOV-graph e uma análise qualitativa dos resultados.

Em (MEDEIROS ET AL., 2007B), é dada continuidade à idéia de mapeamento e navegação entre as atividades do ciclo de desenvolvimento de software OA e a propagação de informações entre elas, acrescentando a definição da combinação harmônica entre os elementos definidos na atividade de projeto arquitetural de software e na atividade de projeto detalhado. De forma a permitir essa propagação de informações e navegação entre atividades, a idéia de combinação harmônica considera a arquitetura de software e o projeto detalhado, como artefatos que possuem elementos associados, dispostos e correspondentes harmonicamente aos elementos da outra atividade. Para efetivar essa combinação harmônica entre os elementos de projeto arquitetural e projeto detalhado, definimos regras de mapeamento entre AspectualACME e aSideML (CHAVEZ, 2004), uma linguagem para especificação e comunicação de projetos orientados a aspectos. As especificações AspectualACME usadas nesse trabalho são resultantes do mapeamento simbiôntico entre requisitos e arquitetura, definido em (SILVA ET AL., 2007). A linguagem aSideML oferece notação, regras e semântica para dar suporte à Modelagem Orientada a Aspectos (MOA), e trata aspectos e *crosscutting* como elementos de primeira classe. Dessa forma, *crosscutting* refere-se a uma mecanismo de modelagem que relaciona aspectos e componentes de forma transparente (CHAVEZ, 2004).

Em (MEDEIROS ET AL., 2007C) é apresentada a ferramenta MaRisa (*Mapping Requirements to Software Architecture)*, que realiza o mapeamento e transformação entre uma especificação de requisitos orientada a aspectos AOV-graph, para uma especificação arquitetural orientada a aspectos AspectualACME, e vice-versa. Ela foi desenvolvida em Java (JSE 6.0) e utiliza os recursos disponibilizados pela linguagem, tais como o DOM (*Document Object Model).*

Um ponto em aberto na série de trabalhos desenvolvidos é a falta da integração das estratégias e ferramentas de mapeamento entre modelos orientados a aspectos representativos das fases de requisitos, arquitetura de software e projeto detalhado com as tecnologias para definição, marcação, execução e validação de modelos propostas na linha do MDD. Essa falta de integração ocasionava menos representatividade entre as

fases em diferentes níveis de abstração, não existência de documentação a partir dos modelos, falta de verificação e rigor semântico dos modelos que são gerados, além da ausência de formalização na especificação das regras de mapeamento.

A inserção e integração das abordagens relacionadas à transformação entre modelos orientados a aspectos, no contexto do MDD (*Model Driven Development)*, provê suporte para: (i) criação de um ambiente integrado para automatização das transformações entre modelos de requisitos orientados a aspectos, AOV-graph, para uma especificação arquitetural orientada a aspectos, AspectualACME, e de AspectualACME para aSideML, através dos recursos e execução de transformações MDD; (ii) os resultados da execução do processo de transformações MDD entre tais modelos sejam modelos tanto de requisitos, como de arquitetura, melhor modularizados, independentes, encapsulados e reutilizáveis; (iii) seja facilitado o acesso às informações relativas às atividades e aos modelos do AOV-graph, AspectualACME e aSideML originados durante o processo de desenvolvimento, permitindo que requisitos possam ser adicionados e propagados facilmente para arquitetura, e de arquitetura para projeto detalhado; (iv) diminua-se o *gap* entre estas etapas de desenvolvimento e entre os modelos construídos nelas; (v) facilite-se a rastreabilidade de mudanças e características transversais tanto em modelos de requisitos quanto na arquitetura; (vi) diminua-se o retrabalho de modelagem e modificação entre os modelos AOV-graph, AspectualACME e aSideML e (v) favorecer o reuso, através da modularização e evolução dos modelos gerados.

1.2. Objetivos

Este trabalho tem como objetivo principal definir transformações MDD entre modelos orientados a aspectos das fases de requisitos, arquitetura e projeto detalhado, e criar um ambiente integrado para execução do processo dessas transformações. As fases de requisitos, arquitetura e projeto detalhado são representadas respectivamente por modelos AOV-graph, AspectualACME e aSideML.

Os modelos e transformações apresentados neste trabalho são inseridos no contexto de uma abordagem orientada a modelos, que visa fornecer um ambiente integrado para o desenvolvimento de software orientado a aspectos, desde a fase de requisitos, até projeto detalhado. O mapeamento entre requisitos e arquitetura, e vice-

versa, e arquitetura e projeto detalhado, são materializados por meio da especificação de transformações entre modelos de requisitos AOV-graph (SILVA, 2006) e modelos de arquitetura AspectualACME (BATISTA, 2006), e especificação de transformações entre modelos AspectualACME e aSideML (CHAVEZ, 2004). As transformações são implementadas no ambiente Eclipse (ECLIPSE, 2008). A linguagem KM3 (Kernel MetaMetaModel) (JOUAULT, 2006) é usada para descrição de metamodelos para AOV-graph, AspectualACME e aSideML. O componente TCS (*Textual Concret Syntax*) (JOUAULT, 2006) é usado para se obter a representação textual para modelos AspectualACME e aSideML. A linguagem de transformação ATL (*ATLAS Transformation Language*) (ATL, 2008) será usada para especificar as regras de transformação entre modelos.

Os objetivos específicos do trabalho incluem:

- Especificação de metamodelos em KM3 (para gerar ECORE) de AOV-graph, AspectualACME e aSideML.
- Especificação em ATL das transformações entre os modelos de AOV-graph, AspectualACME e, de AspectualACME para aSideML, sendo essas transformações baseadas nos respectivos metamodelos de cada linguagem.
- Especificação da BNF de aSideML para verificação do modelo textual utilizando o mecanismo TCS.
- Criação de extratores e injetores para as transformações de modelo para texto, e vice-versa.
- Implementação completa das transformações entre modelos, utilizando ATL e o ambiente MDA oferecido pela IDE Eclipse, formando o ambiente integrado MARISA-MDD.
- Validação das transformações através dos estudos de caso: Sistema Health Watcher (Soares et al., 2002) e Mobile Media (Figueiredo et al., 2008).
- Análise qualitativa dos modelos resultantes das transformações.

1.3. Estrutura do trabalho

O restante desse trabalho está organizado da seguinte forma. O Capítulo 2 trata dos conceitos básicos relacionados ao trabalho: desenvolvimento orientado a aspectos, engenharia de requisitos, arquitetura de software, projeto detalhado, bem como as linguagens de modelagem utilizadas em cada atividade, AOV-graph, AspectualACME e aSideML, incluindo os seus respectivos metamodelos, e MDD (*Model Driven Development)*, contemplando a apresentação das suas tecnologias e recursos. O Capítulo 3 apresenta a definição das regras de transformações baseadas em modelos: de Requisitos ao Projeto detalhado. O Capítulo 4 apresenta a descrição das regras de transformação em ATL e como são gerados os modelos a partir destas regras e dos metamodelos. O Capítulo 5 contém os trabalhos relacionados. O capítulo 6 contém as conclusões e trabalhos futuros, apêndices e anexos

2. Fundamentação Teórica

O objetivo deste capítulo é apresentar, resumidamente, os principais conceitos que estão envolvidos neste trabalho. Na Seção 2.1, apresentamos o HealthWatcher, sistema utilizado como estudo de caso ao longo da dissertação, Seção 2.2, damos uma visão geral do Desenvolvimento Orientado a Aspectos (DSOA). Na Seção 2.3 apresentamos os conceitos relacionados à Engenharia de Requisitos e a linguagem AOV-graph. A Seção 2.4 trata de Arquitetura de Software e da linguagem AspectualACME. Na Seção 2.5 são apresentados conceitos de Projeto Detalhado e a linguagem aSideML. Por fim, a Seção 2.6 apresenta o MDD (*Model Driven Development) e* as técnicas que serão utilizadas neste trabalho para transformação de modelos.

2.1. *HealthWatcher* (SOARES ET AL., 2002)

É um sistema de informações disponibilizado via WEB que oferece aos usuários serviços relacionados ao sistema público de saúde, como suporte ao registro de reclamações e guia de saúde para o usuário.

O sistema possui vários componentes, dentre eles: Negócio, relacionada as regras de negócio, dados, guarda informações relacionadas ao sistema, e Usabilidade, relacionada as interfaces do sistema.

Ao longo do nosso trabalho iremos focar os exemplos no componente Usabilidade.

2.2. Desenvolvimento de Software Orientado a Aspectos (DSOA)

A Programação orientada a Aspectos (POA) (KICZALES ET AL., 1997) emergiu com a finalidade de separar o código que implementa características transversais, do código que implementa a funcionalidade básica da aplicação. A POA propõe o conceito de *aspectos* (BAKKER ET AL., 2005) para representar as características transversais (*crosscutting concerns)* (KICZALES ET AL., 1997). De uma forma geral, características transversais são partes de um sistema que estão fortemente relacionadas, entrelaçadas ou sobrepostas, influenciando ou restringindo umas as outras, tornando o sistema complexo e difícil de analisar. Em termos de programação, encontram-se em partes do código da aplicação que implementam

funcionalidades específicas que não fazem parte das funcionalidades básicas que a aplicação deve atender. Como exemplo, é possível citar fragmentos de código espalhados por diversos componentes de um sistema, que tentam expressar propriedades como sincronização, interação entre componentes, distribuição e persistência.

Assim, no desenvolvimento de um sistema baseado em aspectos observa-se a distinção entre dois tipos de código: um *código base* que diz respeito ao propósito básico da aplicação e um código que está espalhado no código base causando o "entrelaçamento" de código e dificultando a compreensão e manutenção do software.

A idéia da POA tem sido refletida nas demais atividades do ciclo de desenvolvimento de software, definindo estratégias de Desenvolvimento de Software Orientado a Aspectos (DSOA). Os conceitos comumente definidos em DSOA são: aspectos, joinpoints, pointcuts, advices.

Aspectos são unidades modulares, designadas para implementar e encapsular características transversais por meio de instruções sobre onde, quando e como eles são invocados (FILMAN ET AL., 2005). Os *joinpoints* são locais bem definidos na estrutura ou fluxo de execução de programa onde comportamentos adicionais podem ser adicionados, i.e., são afetados por aspectos (FILMAN ET AL., 2005). Os *pointcuts* indicam os elementos afetados por uma determinada característica transversal, são mecanismos que encapsulam os *joinpoints.* Isso é uma importante característica da POA, por possibilitar um "mecanismo de quantificação", ou seja, um caminho para relacionar algo importante em muitos lugares de um programa com uma simples declaração (FILMAN ET AL., 2005). Por exemplo, um programador poderia designar todos os *joinpoints* em um programa onde, a segurança deveria ser invocada. Isso elimina a necessidade de referenciar explicitamente outro *jointpoint*, e reduz a probabilidade de que código de aspecto seja incorretamente invocado.

Advices definem o comportamento a ser adicionado no *joinpoint* (FILMAN ET AL., 2005). Os *advices* possuem duas partes, a primeira é o *pointcut,* que determina as regras de captura dos *joinpoints;* a segunda é o código que será executado quando ocorrer o ponto de junção definido pela primeira parte. Cada *advice* possui um tipo (*after, before e around)* que descreve quando o comportamento deve ser inserido nos *joinpoints.* (WINK; JUNIOR, 2006).

Intertype Declarations são mecanismos utilizados para adicionar novos tipos de dados ou de elementos ao sistema. Fornece a possibilidade de adição de novos elementos sem que seja preciso fazer alterações diretas na especificação ou programa.

Como exemplo, é possível citar a adição de novos métodos, atributos ou construtores a uma classe.

As abordagens que envolvem o desenvolvimento orientado a aspectos podem ser simétricas ou assimétricas. As abordagens simétricas não fazem distinções entre aspecto e componentes, e não fazem essa distinção em "modelo base". Já as abordagens assimétricas são caracterizadas pelos aspectos e componentes terem estruturas diferentes que implementam um "modelo base", a composição de componente-componente, composição aspecto-aspecto e classe-classe não são suportadas (HARRISON; OSHER; TARR, 2002).

2.3. Engenharia de Requisitos

O processo de descoberta, documentação e manutenção do conjunto de requisitos para um determinando sistema computacional é chamado de *Engenharia de requisitos* (SOMMERVILLE, 2003).

A Engenharia de Requisitos é uma atividade inicial do processo de desenvolvimento de software que auxilia na compreensão acerca da natureza dos problemas complexos que surgem no decorrer do desenvolvimento de sistemas e no atendimento das expectativas do usuário. Assim, é seu papel realizar a interação entre requisitantes [1], e desenvolvedores – entre "o que" deve ser feito e "como" deve ser feito.

Sommerville (SOMMERVILLE, 2003) separa diferentes níveis de descrição para o processo de engenharia de requisitos. Ele utiliza o termo *requisitos do usuário* para designar os requisitos abstratos de alto nível, e o termo *requisitos de sistema* para indicar a descrição detalhada do que o sistema deverá fazer. Além desses dois níveis de detalhes, uma descrição mais detalhada (uma especificação de projeto de software) pode ser produzida para associar a engenharia de requisitos e as atividades de projeto.

Dessa forma, para descrever e documentar requisitos do usuário e sistema existem as linguagens de descrição de requisitos. A seguir apresentaremos, brevemente, uma linguagem de descrição de requisitos que será usada nesse trabalho.

2.3.1. Linguagem de Descrição de Requisitos – V-Graph

Modelos de Metas surgiram com o objetivo de modelar requisitos. Os modelos de metas representam de forma explícita requisitos funcionais e não funcionais por meio

---

[1] O termo requisitante está sendo usado para representar pessoas que requisitam serviços ou impõem restrições, tais como usuários e cliente.

da decomposição de metas (GIORGINI ET AL., 2002) (YU ET AL., 2004). Esta decomposição é utilizada para indicar como satisfazer uma determinada meta, mostrando a razão pela qual as submetas são necessárias. Na literatura existem algumas variações de modelos de metas. De uma forma geral eles usam árvores *and/or* para representar modelos de metas e determinar um espaço de soluções para satisfazer a meta raiz (SILVA, 2006).

V-graph é um tipo de modelo de metas (YU ET AL., 2004). Originalmente, que utiliza os seguintes tipos de elementos: *softmetas*, metas e tarefas, e tipos de relacionamentos: contribuição e correlação. Os relacionamentos podem conter rótulos que podem ser *and, or, make, help, unknow, hurt, break* (YU ET AL., 2004). A Figura 1 ilustra os possíveis relacionamentos entre metas, softmetas e tarefas e a Figura 2 ilustra exemplos desses relacionamentos.

O modelo de metas V-Graph permite a descrição de nós intencionais (metas e *softmetas) e* nós operacionais (tarefas). Geralmente as *softmetas* são propostas como forma de modelar e analisar requisitos não funcionais (MYLOPOULOS ET AL., 1992), enquanto as metas representam macro-requisitos, regras ou objetivos do negócio, e as tarefas representam requisitos funcionais que operacionalizam metas e *softmetas* (SILVA, 2006).

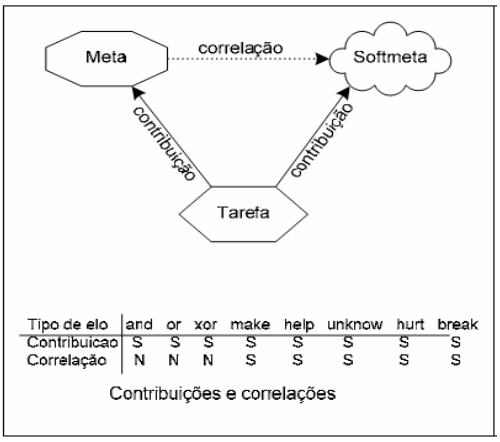


Figura 1: O modelo de metas V-graph: representação global definida em (Yu, 2004).

Os relacionamentos são:

- Contribuição – elos de contribuição podem ter os rótulos *AND*, *XOR*, *OR* *make*(++), help(+), *unknown(?), hurt*(-) e *break(-).* O rótulo *AND* indica que o subelemento é obrigatório, *OR* que ele é opcional e *XOR* que pode ocorrer

apenas um dos subelementos, m*ake* e *help* indicam contribuição positiva, *unknown* indica que há um relacionamento, porém desconhecido se positivo ou negativo, e *hurt* e *break* indicam contribuição negativa. A contribuição é utilizada no sentido de uma sub*softmeta* para uma *softmeta,* entre tarefas e metas, de uma tarefa para uma meta, ou de uma tarefa para uma *softmeta* devido a natureza *fuzzy* dos elementos *softmeta*.

- Correlação – elos de correlação também podem ter os rótulos *make(*++), *help* (+), *unknown(?), hurt*(-) e *break(-).* Entretanto, a correlação é utilizada para representar os relacionamentos de contribuição horizontal entre diferentes árvores ou subárvores, indicando maior acoplamento do que o elo de contribuição. A correlação é baseada no conceito de "conflito e harmonia" indicando a interferência positiva e negativa entre árvores de metas aparentemente desconexas. A correlação pode ocorrer de *softmeta* para *softmeta*, de meta para *softmeta*, e de meta para meta. A correlação de tarefa para tarefa ocorre somente quando existe uma correlação entre os elementos pais (*softmetas*) de ambas as tarefas.

Na Figura 2 estão representados os relacionamentos de contribuição e correlação. Na Figura 2 (a) está representado o relacionamento de contribuição entre as sub *soft*metas Confidencialidade, Integridade e Disponibilidade e a *soft*meta Segurança, indicando uma contribuição positiva através do elo *help.* Na Figura 2 (b) temos que a representação do relacionamento de correlação entre a meta Persistência e a *soft*meta Segurança, indicando interferência negativa, e o subelemento *softmeta* Confidencialidade, indicando uma interferência positiva.

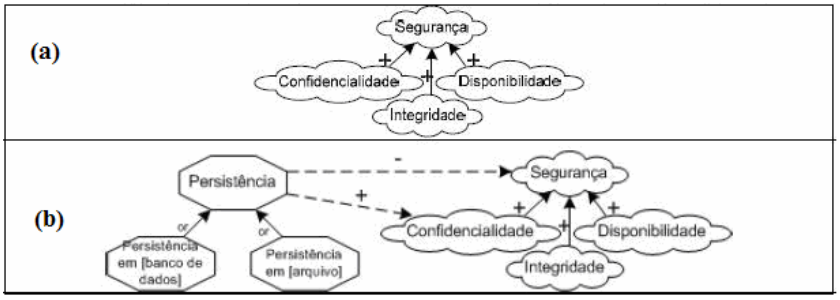


Figura 2: Exemplo de (a) contribuição e (b) correlação.

2.3.2. Linguagem de Requisitos Orientada a Aspectos – AOV-graph

A Engenharia de Requisitos Orientada a Aspectos provê técnicas e métodos para sistematicamente identificar, separar, representar e compor características transversais nas fases de desenvolvimento anteriores à implementação (RASHID ET AL., 2002). No processo de definição de requisitos ainda não se tem uma clara definição de "aspectos" (BAKKER ET AL., 2005) (NUSEIBEH, 2004). A maioria das abordagens adota o conceito de "candidato a aspecto", denotando as características transversais, em muitas destas, elas são somente requisitos não funcionais (ALENCAR, 2005), não abrangendo a completude que envolve a definição de aspectos.

Dessa forma, diversas linguagens de modelagem de requisitos orientados a aspectos têm sido criadas, dentre elas Theme/Doc (BANIASSAD; CLARKE, 2004) e RDL – Linguagem de Descrição de Requisitos (CHITCHYAN ET AL., 2006).

AOV-graph (SILVA, 2006) é uma extensão do modelo de metas V-graph que adiciona à linguagem V-graph um novo tipo de relacionamento denominado relacionamento transversal (*crosscutting relationship*). AOV-graph consiste de metas (*goals*), softmetas (*softgoals*) e tarefas (*tasks*). As metas e softmetas são nós intencionais e tarefas nós operacionais. Usualmente, softmetas representam RNFs, metas representam macro-requisitos, regras ou objetivos do negócio, e tarefas se representam RFs que operacionalizam metas e softmetas. Metas e softmetas distinguem pelos conceitos de satisfy e satisfice (MYLOPOULOS, 1992), respectivamente, i.e.: uma meta é satisfeita totalmente *(satisfy)* através do conjunto de submetas e tarefas nas quais ela se decompõe; e uma softmeta é suficientemente satisfeita *(satisfice)* pelas metas, softmetas e tarefas que contribuem ou estão correlacionadas positivamente a ela.

Na Figura 3, está ilustrado em UML o metamodelo AOV-graph que inclui os elementos de V-graph.

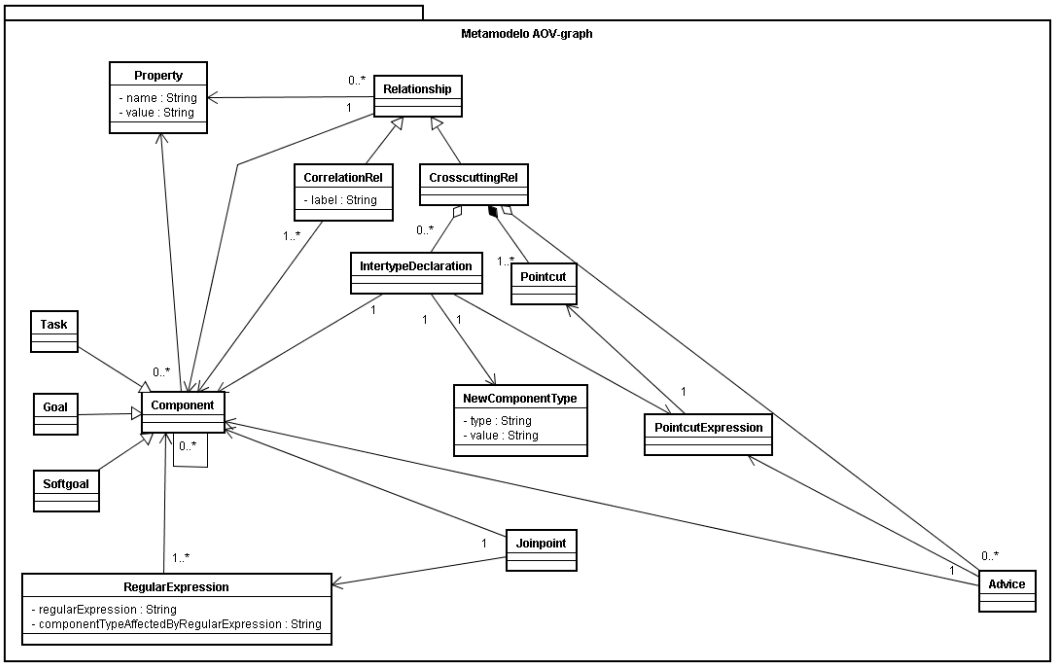


Figura 3: Metamodelo AOV-graph

O relacionamento transversal é descrito por: (i) *source* - indica o ponto de origem do relacionamento, que no diagrama da Figura 3 é representado por uma instância do elemento *Component*; (ii) *pointcut* – indica os pontos de destino do relacionamento, os pontos a serem afetados por uma característica transversal; (iii) *advice* - indica elementos pertencentes ao ponto de origem que estão transversais aos pontos de destino, i.e., a característica transversal em si. Os tipos *before*, *around* e *after* podem indicar, respectivamente, pré-condição, decomposição e pós-condição no AOV-graph; e (iv) *intertype declaration* – define novos tipos de elementos, também transversais, que surgem com a integração das características indicadas nos pontos de origem e destino do relacionamento transversal. Há dois tipos de *intertype declaration*: *element* – quando os novos tipos referem-se a instâncias de elementos de tipos existentes no modelo de requisitos, no caso de AOV-graph referem-se a metas, softmetas e tarefas; e *attribute* – quando os novos tipos referem-se a atributos inexistentes no modelo de requisitos.

Na Figura 4, exemplificamos o uso do relacionamento transversal. O ponto de origem do relacionamento transversal é *Usability [UI]* e os pontos afetados são *Select [kind of complain]*, S*elect [specialty], Select [unit], Request.* e Show.**Através do

advice, indicamos que *Use [drop down box], Show [specific form] e Show [information] friendly* afetam as tarefas *Select [kind of complaint]*, S*elect [specialty, Select [unit]* todas as tarefas iniciadas pela *string Request* e *Show*, respectivamente.

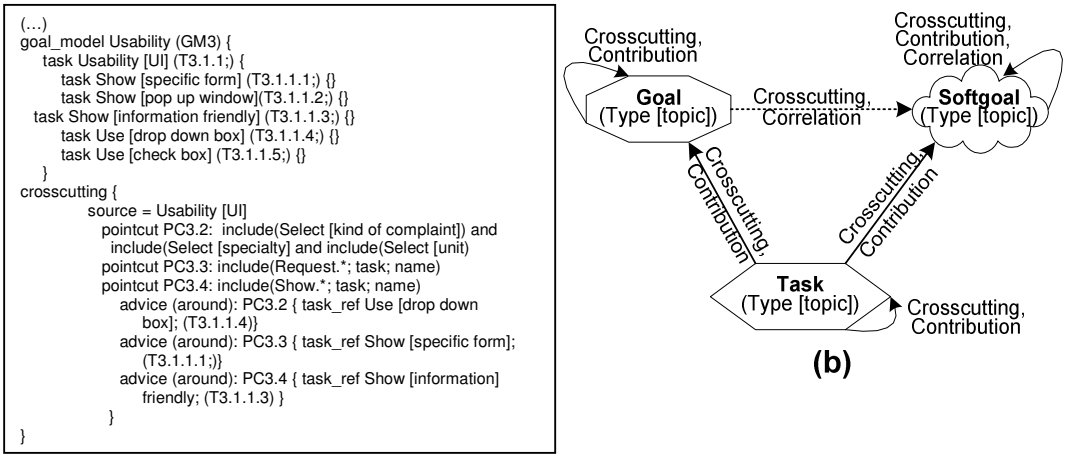


Figura 4: (a) Relacionamentos entre metas, softmetas e tarefas e (b) exemplo de relacionamento transversal em AOV-graph.

O relacionamento transversal em AOV-graph permite que sejam agrupadas tarefas, metas e *softmetas* que estariam espalhadas e entrelaçadas as demais. Todavia, são definidas em (SILVA, 2006) justificativas para a escolha em definir um relacionamento ao invés de um elemento de primeira ordem para modularizar características transversais, mostradas a seguir:

i. foram separadas as informações referentes a "o quê" é transversal, representadas pelos elementos do próprio modelo de requisitos, das referentes a "como" é transversal representadas no relacionamento transversal.

ii. a extensão realizada é menos intrusiva à linguagem de modelagem de requisitos escolhida.

iii. é facilitado o reuso de características que em alguns projetos têm essa propriedade de transversalidade e em outros não a têm. É considerado que características transversais, na verdade, "estão" transversais.

2.4. Arquitetura de Software

A arquitetura de Software (GARLAN; SHAW, 1993) é uma área de conhecimento que envolve a integração entre metodologias de desenvolvimento de software e modelos. Arquitetura de Software provê uma representação do sistema através de

componentes, conectores e configurações. Os componentes encapsulam um conjunto de funcionalidades e os conectores representam a forma como os componentes interagem entre si. Componentes e conectores contém uma interface associada. A interface de um componente é um conjunto de pontos de interação entre o componente e o mundo externo. Tais interfaces especificam os serviços (mensagens, operações e variáveis) que um componente fornece e também os serviços requisitados de outros componentes e são comumente chamados de *portas*. Conectores especificam regras que governam a interação entre componentes. A interface do conector especifica os pontos de interação entre o conector e os componentes ligados a ele. Configurações definem a estrutura arquitetural especificando a conexão de componentes com conectores (BATISTA ET AL., 2006).

Na Figura 5 está ilustrado o ponto onde as atividades de arquitetura são mais visíveis dentro do processo de desenvolvimento de sistemas. A diferença entre as fases de análise e de projeto e as atividades de arquitetura só são evidentes quando se trata de sistemas grandes e complexos. Neste caso, o entendimento claro do escopo e das possibilidades de evolução são mais difíceis de identificar dado o tamanho da incerteza que advém da falta de conhecimento do domínio do sistema (Varoto, 2002).

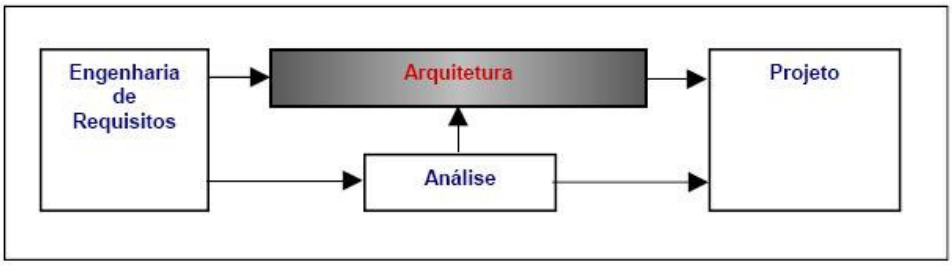


Figura 5: A Arquitetura de software no Processo de Desenvolvimento de Sistemas (VAROTO, 2002).

Assim, o esforço de avaliação e verificação da conformidade dos requisitos para com a arquitetura deve ser realizado durante todo o ciclo de desenvolvimento, ou seja, a cada fase que avança, o registro dos insucessos deve servir de subsídios para a melhoria ou evolução da arquitetura atual, mostrando que a definição e manutenção da arquitetura é um processo incremental e iterativo (VAROTO, 2002).

A seguir apresentaremos, brevemente, uma linguagem de descrição arquitetural que será usada nesse trabalho.

2.4.1. Linguagens de Descrição Arquitetural (ADLs) e ACME

Linguagens de Descrição Arquitetural (*Architectual Description Language – ADLs)* podem ser linguagem gráficas, textuais ou ambas. Elas expressam os componentes e relacionamentos arquiteturais (CLEMENTS ET AL., 2005). De acordo com Garland (GARLAND, 2002) as ADLs são linguagens formais utilizadas para representação de arquiteturas de sistemas, e para isso, elas oferecem abstrações e promovem a separação de conceitos (SoC) através da dissociação entre elementos arquiteturais que representam uma computação (componentes) e elementos que representam a conexão entre componentes (conectores).

ACME (GARLAN, 1997) é uma ADL que dá suporte à representação de: (i) estrutura da arquitetura, que é a organização do sistema e suas partes constituintes; (ii) propriedades, que são informações sobre o sistema ou suas partes que permitem uma completa representação abstrata de comportamentos, ambos funcionais e não funcionais; (iii) tipos e estilos, os quais definem classes e famílias de arquiteturas, e (iv) restrições, que são guias de como a arquitetura pode mudar.

A estrutura da arquitetura é descrita em ACME por meio de componentes, conectores, sistemas, configurações, portas, regras e representações. A seguir, algumas partes constituintes dessa ADL são conceituadas (BATISTA ET AL., 2006):

- Os componentes são, fundamentalmente, elementos computacionais que dão suporte a múltiplas interfaces, conhecidas como portas.
- Portas especificam serviços (mensagens, operações e variáveis) que um componente fornece e também serviços requisitados de outros componentes.
- Sistemas são abstrações que representam configurações de componentes e conectores. Um sistema inclui uma parte de componentes, uma de parte conectores, e uma parte de configurações.
- Configurações definem a estrutura arquitetural especificando a conexão de componentes e conectores.
- Representações são alternativas para decomposição de elementos (componentes, portas ou regras) para melhor descrição de detalhes. Assim, essa representação pode ser vista como mais um refinamento de descrição de um elemento.
- Propriedades são definidas por um nome, tipo e valor, que podem ser configurados para alguns elementos ACME como anotações. As propriedades

são um mecanismo para anotações do desenho e detalhamento de desenho de elementos, geralmente informações não estruturais.

A Figura 6 representa um exemplo da arquitetura do sistema *HealthWatcher* (HW) (SOARES ET AL., 2002), um sistema de informações disponibilizado via WEB que oferece aos usuários serviços relacionados ao sistema público de saúde, tais como o suporte a registro de reclamações e guia de saúde para o usuário. Nesse exemplo, podemos visualizar: (i) um componente *GUI- (Graphical User Interface)* que fornece uma interface WEB para o sistema; (ii) um componente *Business* que define regras de negócio, e (iii) um componente *Data* que guarda informações manipuladas pelo sistema.

```
System HealthWatcher = {
 Component  GUI ={
   Port GUI_RegisterUser;
   Port Gui RegisterComplaint;
   Port GUI_ListComplaint;
   Port UpdateComplaint;
};
 Component Business = {
  Port Business Complaint;
  Port saveInfo;
 };
 Component Data = {
  Port storeInfo;
  Port recoveryInfo;
};
 Connector C1, C2 = {
   Roles caller, callee;
}
  Attachments {
   GUI.UpdateComplaint to C1.caller;
   C1.callee to Business.BComplaint;
   Business.saveInfo to C2.caller;
   C2.callee to Data.storeData;
}}
```

Figura 6: Um exemplo de descrição em ACME definida em (BATISTA, 2006)

2.4.2. AspectualACME

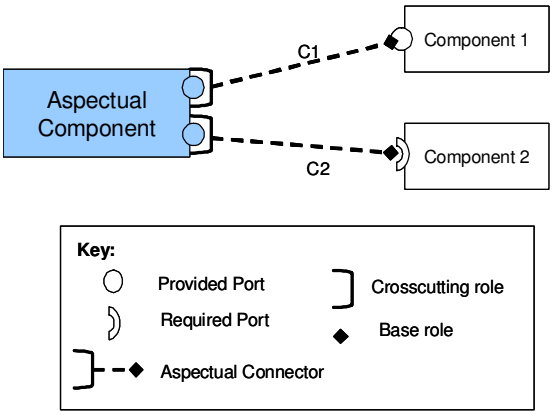
Visando oferecer suporte à modularização de caracteríticas transversais no nível arquitetural, várias linguagens de descrição arquitetural orientadas a aspectos (AO ADLs) têm sido propostas (BATISTA ET AL., 2006) (BENEDÍ ET AL., 2003) (PINTO ET AL., 2005). Em (BATISTA ET AL., 2006) é proposto o conceito de **Conector Aspectual** (*Aspectual Conector* – AC) como a única abstração necessária para expressar relacionamentos transversais no nível arquitetural. O conector aspectual baseia-se na tradicional distinção existente em ADLs entre computação e interação e explora essa mesma idéia para modelar interação entre componentes e conceitos transversais. Elementos computacionais são modelados como componentes, independentemente se implementam conceitos transversais ou não. A distinção entre componentes aspectuais e

componentes tradicionais está na forma que eles são compostos com outros elementos do sistema.

Na Figura 7 estão ilustradas a especificação (a) textual e (b) gráfica do *Conector Aspectual*.

```
Conector  aConnector = {
   Base Role aBaseRole;
   Crosscutting Role aCrosscuttingRole;
   Glue glueType;
}
```

(a) Notação Textual



(b) Notação Gráfica

Figura 7: Conector Aspectual: (a) Notação Textual; (b) Notação Gráfica

A Figura 7 (b) representa um exemplo de composição entre um componente (aspectual) com dois componentes. C1 e C2 são exemplos de conectores aspectuais. É importante ressaltar que não há uma abstração distinta para representar aspectos arquiteturais. Estes são representados por componentes. As cores diferentes na Figura 7 (b) são usadas apenas para diferenciar qual componente está fazendo o papel de componente aspectual na interação transversal.

A *base role, crosscutting role e glue clause* são especificações que compõem a interface do conector aspectual. A *base role* do *conector aspectual* é conectada à uma porta do componente afetado pela interação transversal (componente base) – e a *crosscutting role* é conectada a porta do componente aspectual. O par *base role – crosscutting role* não impõe restrição, comum em ADLs, de conectar portas de entrada (*provided ports)* e portas de saída (*required ports).* Na Figura 7, o conector aspectual C1 conecta a porta de entrada do componente aspectual com a porta de entrada do componente 1. A cláusula *glue* especifica detalhes desta conexão aspectual como, por exemplo, e em que momento a interação aspectual acontece – *after, before ou around* .

AspectualACME (BATISTA, 2006) estende a linguagem ACME incluindo o conceito de conector aspectual e incluindo a possibilidade de se usar mecanismo de quantificação, como wildcards, na descricação da configuração. O mecanismo de quantificação que simplifica, sintaticamente, a referência a conjunto de pontos de

junção em uma descrição arquitetural. Em ACME os principais elementos básicos são componentes, conectores e configurações (chamados de *attachments*). A configuração define o local onde os *joinpoints* estruturais são identificados em ACME O mecanismo de quantificação é usado na configuração através do uso de wildcards (*) que representam parte do nome de componentes ou portas.

Na Figura 8, está ilustrado em UML o metamodelo de AspectualACME.

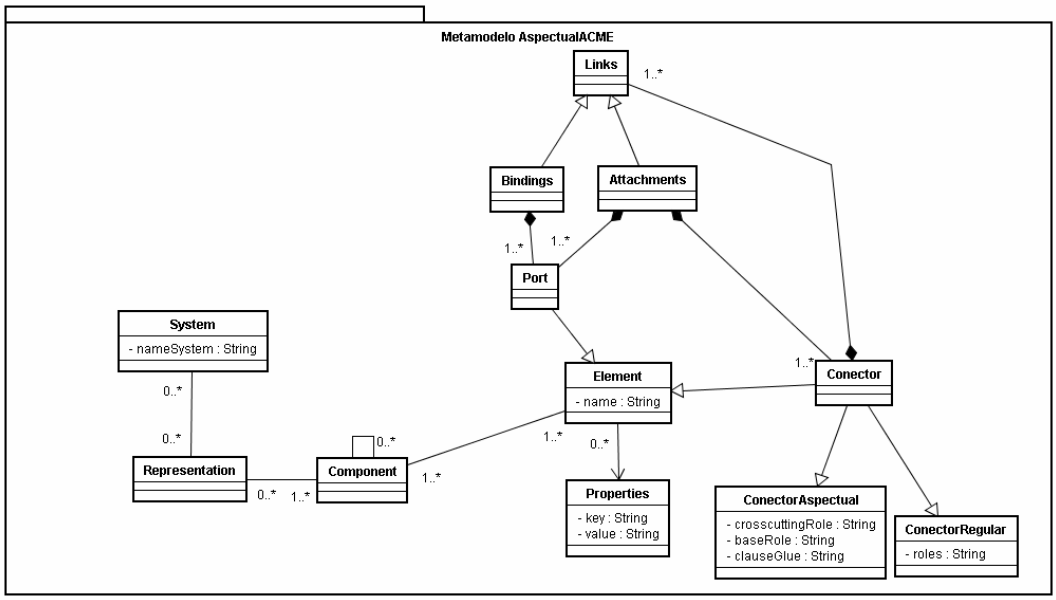


Figura 8: Metamodelo de AspectualACME

A

Figura 9 ilustra um exemplo em *AspectualACME* do *HealthWatcher* onde o componente *Usability_UI,* que é o componente aspectual, é conectado, via o conector aspectual *Usability_UI_Con1*, ao Componente *Specialties_of_a_health_unit* que contem a porta *Select_unit.*

```
System HealthWatcherSystem = {
(…)
Component Usability = {
  Properties { elementType = softgoal; contribution =[and,null]; correlations = [null,null]; topics =
[null];}
  Representation details= {
   System details = {
(…)
   Component Usability_UI = {
      Properties  = { elementType = softgoal; contribution    =[make,Usability]; correlations =
[null,null]; topics = [UI;}
       Port Show_specific_form =  {  (…) };
       Port Show_pop_up_window = { (…) };
        Port Show_information_friendly = { (…) };
       Port Use_drop_down_box = { (…) };
       Port Use_check_box = { (…) };
    Component Accessibility = {
    Properties  { elementType = softgoal; contribution    =[help,Usability_UI]; correlations =
[null,null]; topics = [null];}
       Port Access_functionality_with_mouse = { (…)};
       Port Access_functionality_with_keyboard = { (…)};
    };
  };
};
 Connector Usability_UI_Con1 = {
   baseRole sink;
   crosscuttingRole source;
   glue source around sink;
  Properties = {comesFrom: advice; crossRelSource = Usability_UI;}
      };
Attachments {
  Usability_UI.Use_drop_down_box to  Usability_UI_Con1.source;
   Usability_UI_Con1.sink to Specialties_of_a_health_unit.Select_unit;
     }
(...)
}
```

Figura 9: Exemplo em AspectualACME

2.5. Projeto Detalhado

Projeto Detalhado consiste em uma importante etapa do desenvolvimento de software: engenheiros de software produzem vários modelos que formam um tipo de rascunho da solução a ser implementada. Dessa forma, podemos analisar e avaliar esses modelos para determinar se eles satisfazem ou não os requisitos. Além disso, pode-se analisar e avaliar soluções alternativas. Finalmente, podemos usar os modelos resultantes para planejar atividades subseqüentes, e usá-los como entrada e ponto de início na construção e testes de software (SWEBOK, 2000).

O Projeto Orientado a Aspectos foca em técnicas e ferramentas para identificação, estruturação, representação e gerenciamento de características transversais, portanto, dos conceitos e propriedades relacionadas ao desenvolvimento de

software orientado a aspectos que auxiliam na compreensão, evolução e reutilização de modelos orientados a aspectos.

Linguagens de modelagem são utilizadas para especificar, estruturar e comunicar modelos. Seu vocabulário e regras têm foco voltado para representação conceitual e física de um sistema (BOOCH G; RUMBAUGH J.; JACOBSON, I., 1999). Uma linguagem de modelagem amplamente utilizada é a UML (*Unified Modelling Language)* (BOOCH G; RUMBAUGH J.; JACOBSON, I., 1999), que é de propósito geral e apresenta visões inter-relacionadas, complementadores a fim de representar modelos do sistema.

A seguir apresentaremos, brevemente, uma linguagem de modelagem orientada a aspectos que será usada nesse trabalho.

2.5.1. Linguagem de Modelagem Orientada a Aspectos - aSideML

Com a crescente preocupação em representar e comunicar projetos orientados a aspectos, algumas estratégias têm sido criadas, tais como: Theme/UML (CLARKE; WALKER, 2003), *Composition Patterns* (CLARKE; WALKER, 2001), AODM with UML (STEIN; HANENBERG; UNLAND, 2002).

Em (CHAVEZ, 2004) é apresentada aSideML, uma linguagem de modelagem construída para especificar e comunicar projetos orientados a aspectos. A linguagem aSideML tem como base o metamodelo aSide, que utiliza o metamodelo de UML (*Unified Modeling Language*) (UML, 2003) como base e provê extensões (novas metaclasses e metassociações) para descrição de aspectos, características transversais e outros elementos de DSOA.

A linguagem aSideML fornece semântica, notação e regras que permitem ao projetista construir modelos direcionados para os principais conceitos, mecanismos e propriedades de sistemas orientados a aspectos. Dessa forma, os aspectos e características transversais são explicitamente tratados como elementos de primeira classe. Esses modelos ajudam a lidar com a complexidade de sistemas orientados a aspectos, ao fornecer visões essenciais da estrutura e do comportamento que enfatizam o papel dos elementos transversais e seus relacionamentos com outros elementos.

As principais características da linguagem aSideML são: (i) suporte a interfaces transversais (*crosscutting interfaces*) de forma explícita, com o objetivo de organizar a descrição de *join points* que estão relacionados a algum local de um componente, sob

perspectiva do aspecto, e o comportamento transversal do aspectos; (ii) suporte à descrição de aspectos como elementos de modelagem parametrizados, promovendo seu reuso; (iii) suporte à descrição explícita de relacionamento entre aspectos e os elementos que ele afeta, chamado de *relacionamento de crosscutting*; (iv) suporte à descrição explícita de relacionamentos de dependência entre aspectos, dentre outros.

Dessa forma, aSideML oferece suporte a modelagem de elementos estruturais, comportamentais, e composicionais, relacionados a modelagem orientada a aspectos. Os principais elementos estruturais de modelagem são os *aspectos*, *elementos base* (de UML) que aspectos afetam e seus *relacionamentos*. ASideML define como principais elementos comportamentais de modelagem as instâncias de *aspectos*, *interações aspectuais* e *colaborações aspectuais.* Além disso, aSideML representa as características transversais em diversos compartimentos que auxiliam na modelagem e definição dos elementos comportamentais. A Figura 10 mostra uma interface transversal com os compartimentos *Additions,* onde são definidos os atributos estaticamente introduzidos nos elementos base*, Refinements* e *Redefinitions,* comportamentos a serem adicionados a classes base, e *Uses*. Os *Refinements* incluem operações exibidas com um adorno, indicando que a operação de *crosscutting* deve ser combinada antes (_op ou *before* op), depois (op_ ou *after* op) ou depois/antes (_op_ ou *around* op) do comportamento da operação base (op). Já as *Redefinitions* contemplam operações que possuem adorno depois ou antes. A operação base não é invocada dentro de seu corpo, pois seu comportamento é sobreposto (redefinido) ao comportamento da operação de *crosscutting*. *Uses* definem o serviço a ser usado dentro do aspecto.

Os elementos de modelagem composicionais são definidos para descrever elementos do processo de combinação (*weaving*). Além disso, aSideML fornece perspectivas, novos diagramas que enriquecem alguns diagramas da UML com o objetivo de apresentar os elementos transversais e seus relacionamentos.

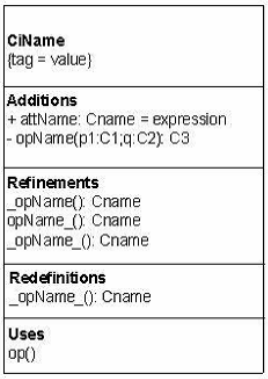


Figura 10: Interface Transversal com os compartimentos de adições, refinamentos, redefinições e usos.

Na Figura 11, está representada uma visão geral e resumida a respeito de aSideML, seus modelos, diagramas, perspectivas e elementos.

Modelo	Diagramas	Perspectivas	Elementos
Estrutural	Diagrama de aspectos		aspecto, interface transversal, característica transversal
	Diagrama de classe estendido	centrado em aspecto	aspecto, *crosscutting* interface transversal, *order*
		centrado em base	
Comportamental	Diagrama de seqüência estendido	ponto de combinação	ponto de combinação dinâmico
	Diagrama de colaboração aspectual		instância de aspecto, colaboração aspectual
	Diagrama de Seqüência		instância de aspecto, interação aspectual
Processo de Combinação	Diagrama de classes combinadas		classe combinada
	Diagrama de colaboração combinada		colaboração combinada
	Diagrama de seqüência combinada		interação combinada

Figura 11: Dimensões da modelagem orientada a aspectos com aSideML

Na Figura 12, Figura **13**, Figura 14 e Figura 15 estão representadas as partes que compõem o metamodelo de aSideML.

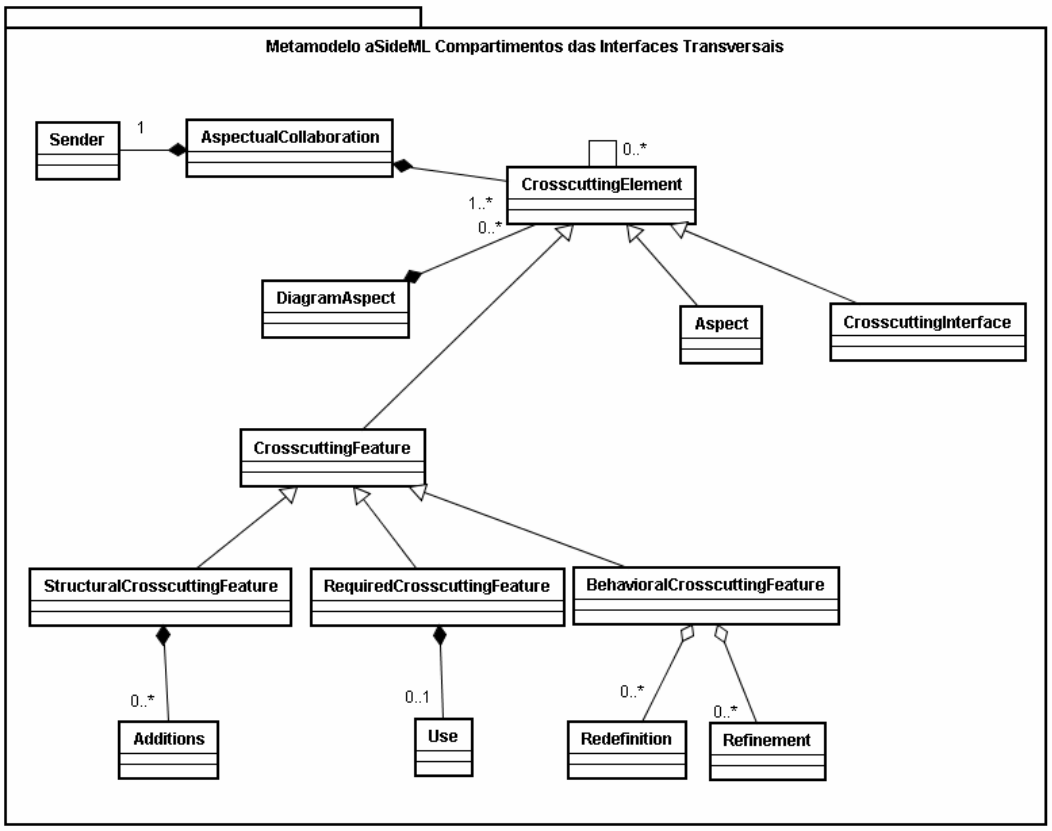


Figura 12: Representação dos compartimentos das interfaces transversais de aSideML

ASideML propõe perspectivas, que de uma forma geral possuem o objetivo de apresentar os elementos *crosscutting* e seus relacionamentos em diagramas.

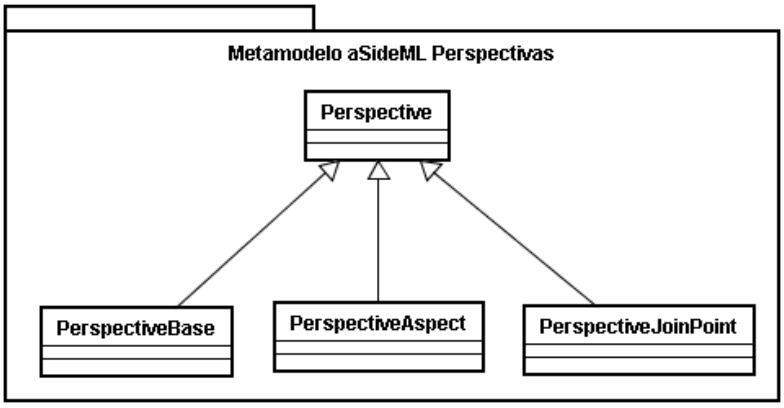


Figura 13: Representação das perspectivas de aSideML

A Figura 14 representa os relacionamentos e diagramas de aspecto e classes estendidos de aSideML. Como exemplo de relacionamento temos o relacionamento *Crosscutting* que é um relacionamento entre um elemento crosscutting (por exemplo,

um aspecto) e um elemento base. Ele especifica que o aspecto deve atravessar os limites do elemento base em pontos de combinação bem definidos e modificar incrementalmente a base nesses pontos.

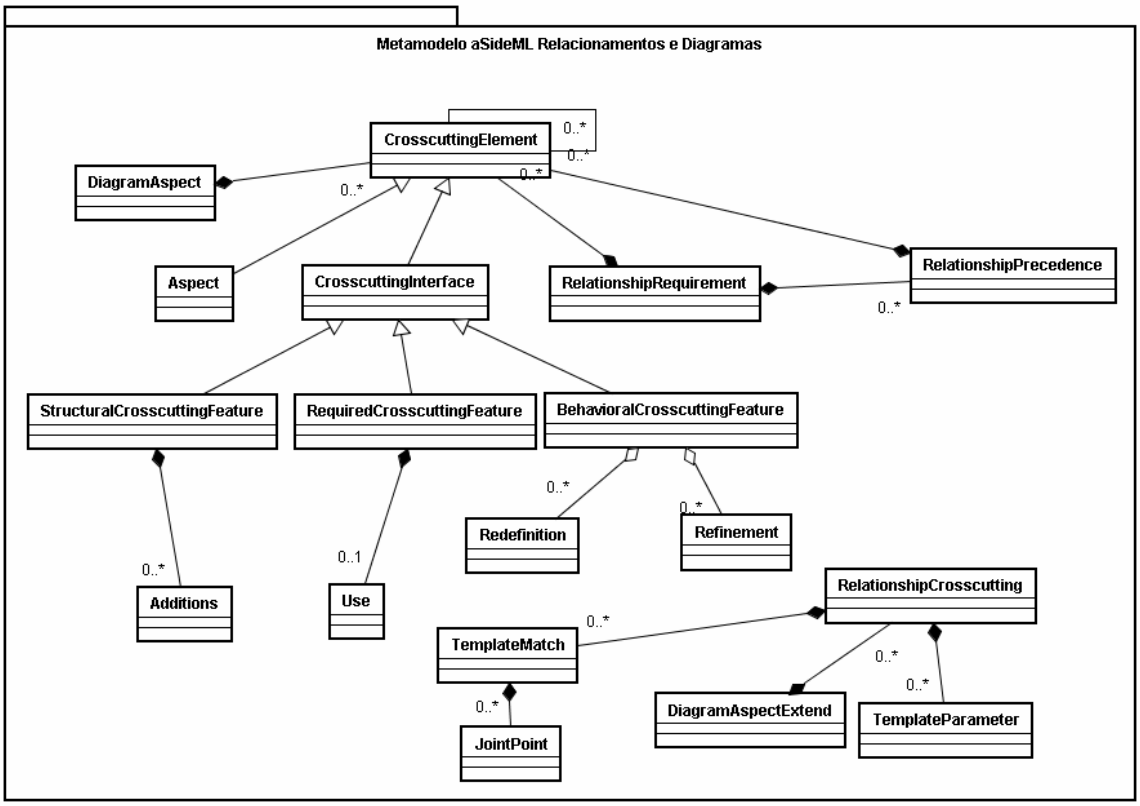


Figura 14: Representação dos relacionamentos e diagramas de aSideML

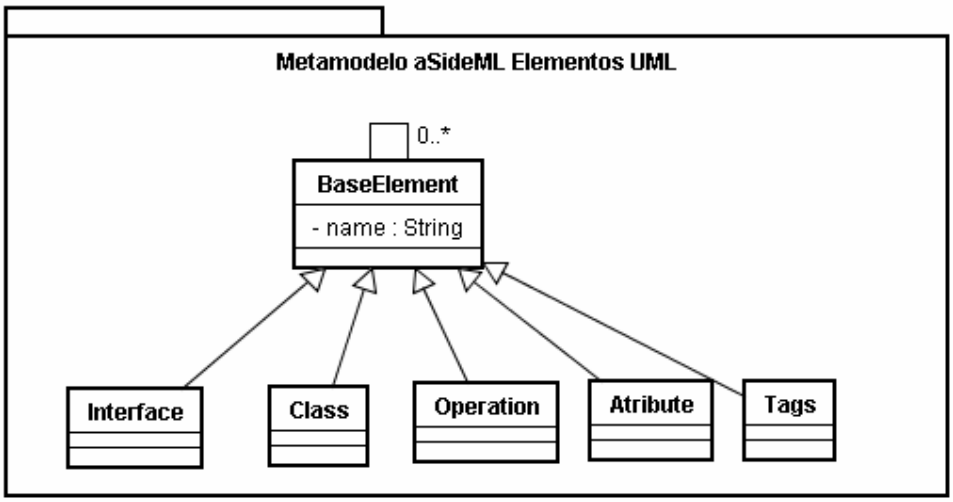


Figura 15: Representação dos elementos base da UML

Na Figura 15, temos a representação dos elementos de modelagem base da UML:

Os principais elementos base utilizado por aSideML são:

- Interface – Representa um serviço realizado. Não possuem implementação ou qualquer especificação interna.
- Classe – Uma classe representa uma categoria. Geralmente possui atributos e métodos.
- Operação – Também conhecida como método ou comportamentos, representa uma atividade que um objeto de uma classe pode executar.
- Atributo – Também conhecido como propriedade, representa característica de uma classe e contêm, normalmente, duas informações, o nome que identifica o atributo e o tipo de dado que o atributo armazena.
- Notas *(Tags)* – Sua função é apresentar um texto explicativo a respeito de um determinado componente do diagrama.

2.5.1.1. Modelagem Estrutural

Em aSideML, a modelagem estrutural fornece a visão estática de um sistema na presença dos aspectos. Os principais constituintes dos modelos estruturais são as classes, aspectos e seus relacionamentos.

Um aspecto é uma descrição de um conjunto de características que melhoram a estrutura e o comportamento de classes por meio de *crosscutting* de forma sistêmica. aSideML oferece uma notação gráfica para a declaração e o uso de aspectos. Os aspectos são declarados em diagramas de aspectos (ver Figura 16), e podem ser usados em outros diagramas, como por exemplo, o Diagramas de classes estendidos (CHAVEZ, 2004).

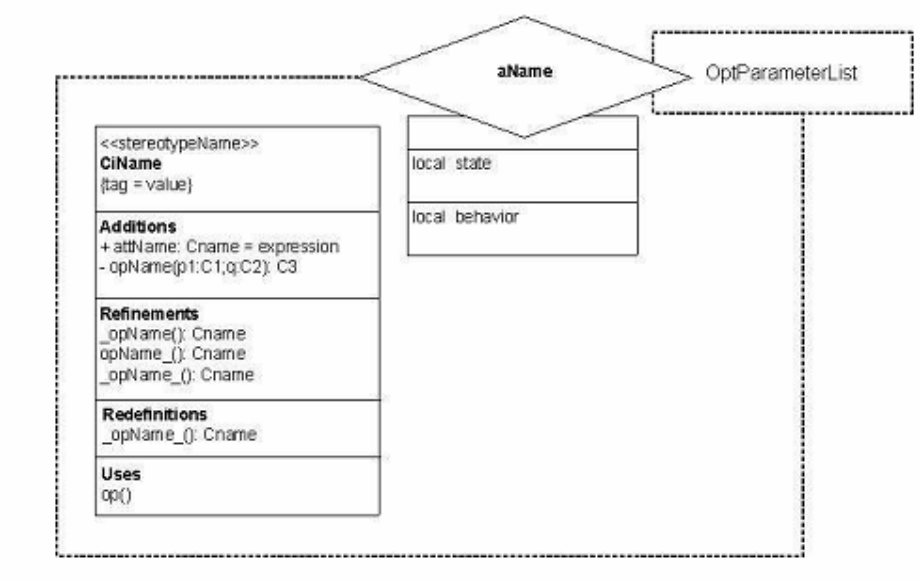


Figura 16: Diagrama de aspecto

Na Figura 16, temos o aspecto representado por um retângulo tracejado pequeno, sobreposto no canto superior direito do retângulo do aspecto. Esse retângulo é nomeado de caixa de parâmetros de templates e possui a lista de parâmetros formais, contendo uma lista para cada interface transversal de aspecto. O primeiro parâmetro da lista é o nome da interface transversal correspondente.

As interfaces transversais são conjuntos de características transversais com nome associado, que caracterizam o comportamento crosscutting de aspectos. Elas são declaradas dentro de aspectos, em diagramas de aspectos, como vimos na Figura 16 (CHAVEZ, 2004).

Uma característica transversal descreve uma propriedade nomeada (atributo ou operação) definida em um aspecto que pode afetar um ou mais elementos base em locais específicos por meio de crosscutting. Em aSideML, dentro das características transversais, existe a característica transversal comportamental que é uma especificação de um atributo que será estaticamente introduzido na interface de uma ou mais classes base, a característica transversal comportamental é uma especificação de uma fatia de comportamento que será adicionada a uma ou mais classes base, ou para refinar ou redefinir uma operação de uma ou mais classes base. Ainda, há a característica requisitada comportamental que é uma operação base que será usada dentro do espaço dos nomes dos aspectos (CHAVEZ, 2004).

O principal relacionamento representado pela modelagem estrutural é o relacionamento *crosscutting* que representa um relacionamento entre um elemento transversal e um elemento base.

Na Figura 17, temos a representação de um diagrama de classes estendido, contendo dois relacionamentos *crosscutting* que associam o aspecto Observation a classe *Button* (ligando *stateChange* a *Click*) e *ColorLabel* (ligando *update* a *colorCycle*).

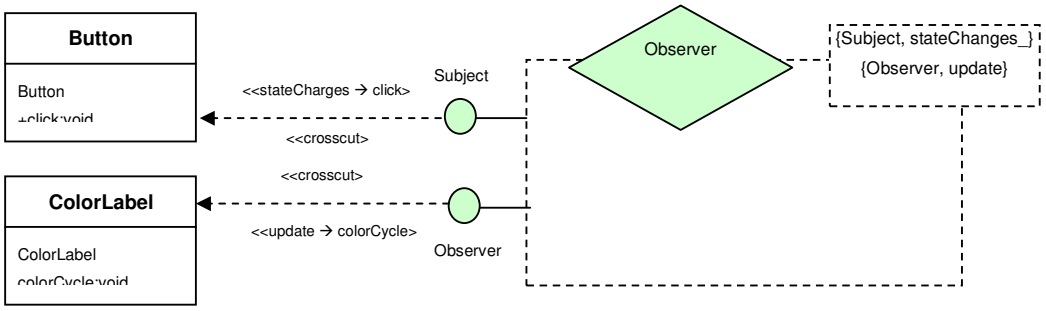


Figura 17: Representação de relacionamento *crosscutting* no Diagrama de Classe Estendido.

2.5.1.2. Modelagem Comportamental

Em aSideML, a modelagem comportamental oferece a visão de interacão de um sistema na presença de aspectos. Os principais constituintes dos modelos comportamentais são objetos, instâncias de aspectos e suas colaborações e interações (CHAVEZ, 2004).

Os modelos comportamentais caracterizam o comportamento de aspectos em termos da forma como interagem com objetos. Esses modelos são representados pelo diagrama de seqüência estendido, diagrama de colaboração aspectual e diagrama de seqüência (ver Figura 18). Além disso, os elementos desses modelos comportamentais são: pontos de combinação dinâmico, instância de aspecto e colaboração aspectual, que é uma descrição de uma organização geral de objetos e instâncias de aspectos que interagem dentro de um contexto a fim de implementar o comportamento *crosscutting* de uma característica transversal comportamental (CHAVEZ, 2004).

Dentre as interações existentes entre os elementos, temos a colaboração aspectual que possui uma parte estática e outra dinâmica. A estática descreve papéis que os objetos e instâncias de aspectos exercem. Já a dinâmica representa fluxo de

mensagens ao longo do tempo para mostrar o comportamento *crosscutting* de acordo com os elementos e papéis exercidos.

Na Figura 18 temos a representação da colaboração aspectual. Há a definição de pelo menos três papéis classificadores definidos em sua parte estática. Há dois papéis predefinidos para objetos: Sender e Receiver, denotando o chamador e o chamado da operação base sendo afetada pelo aspecto.

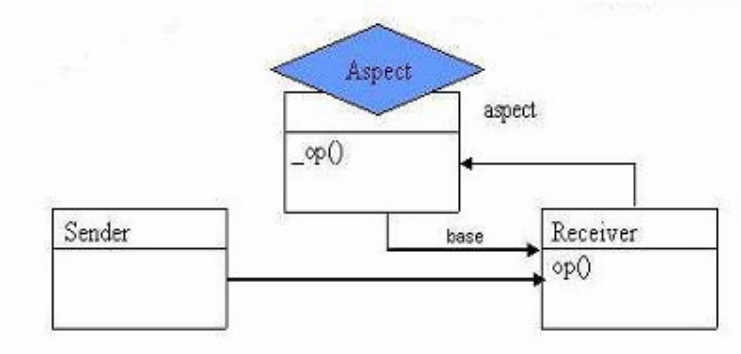


Figura 18: Colaboração aspectual (parte estática)

2.5.1.3. Modelagem Composicional

Na linguagem aSideML, os modelos de processo de combinação (ou composicionais) descrevem as visões estáticas e de interação de um sistema depois da combinação de modelos de objetos e modelos de aspectos (CHAVEZ, 2004).

Os principais constituintes visuais de modelos de processo de combinação são classes combinadas, colaborações combinadas e interações combinadas. Esses elementos são apresentados em diagramas de processo de combinação. Esses diagramas são gerados automaticamente com a ajuda de uma ferramenta de combinação; eles também podem ser considerados diagramas que oferecem uma perspectiva composicional em relação a diagramas de aspectos e diagramas de classes definidos separadamente (CHAVEZ, 2004).

Os elementos combinados modelam os resultados da combinação entre os elementos base e os elementos *crosscutting*. Os resultados de combinação dependem da estratégia de combinação adotada, que depende do modelo de implementação suportado pela ferramenta ou linguagem de programação orientada a aspectos (CHAVEZ, 2004).

2.6. MDD (*Model Driven Development*)

Model Driven Development (MDD) (STAHL ET. AL, 2006) é uma abordagem que engloba essencialmente a especificação de modelos e a transformação de tais modelos em outros modelos ou artefatos de software. Nessa proposta, modelos não são apenas veículos para descrever sistemas de software ou facilitar a comunicação entre as partes do sistema. Eles podem ser associados a diferentes fases do processo de desenvolvimento de software, como está representado na Figura 19. Adicionalmente, relacionamentos entre esses modelos fornecem dependências que auxiliam a entender as implicações de mudanças em algum ponto do processo de desenvolvimento de software (Beydeda, 2005).

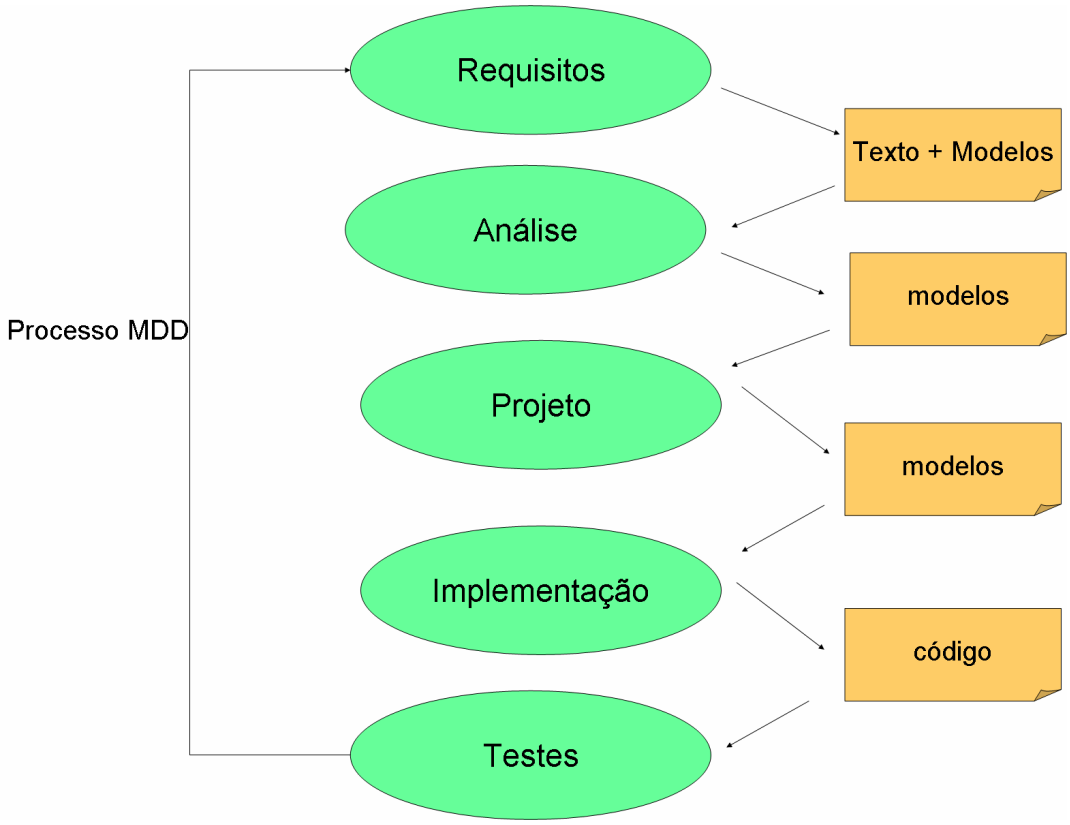


Figura 19: Representação do processo MDD

Uma transformação entre modelos especifica como um modelo de saída é construído baseado em elementos de um modelo de entrada. As linguagens de transformação entre modelos objetivam automatizar o processo de derivar um modelo de saída de um modelo de entrada. Dessa forma, pode-se definir regras para automatizar passos necessários para converter uma representação de um modelo em outro, permitir rastreabilidade entre modelos, e analisar características importantes desses modelos.

Dentro do MDD, temos a Arquitetura Orientada a Modelos (*Model Driven Architecture)* (OMG/MDA, 2006) é uma iniciativa da OMG que propõe a separação da especificação do funcionamento do sistema dos detalhes de como esse sistema utiliza as capacidades de determinada plataforma, através de três níveis de abstrações para modelagem: Modelo Independente de Computação (*Computational Independent Model* – CIM*)*, Modelo Independente de Plataforma (*Platform Independent Model* – PIM), Modelo Dependente de Plataforma (*Platform Specific Model – PSM).* Os modelos são mapeados de uma abstração para outra através de definições de transformações e atuação dos Engenhos de transformação para executar essas transformações. Durante as transformações são acrescentados novos elementos no modelo, diminuindo o nível de sua abstração até o nível de dependência da plataforma computacional aonde o sistema será implementado. Na Figura 20 podemos visualizar a relação entre esses níveis de abstrações.

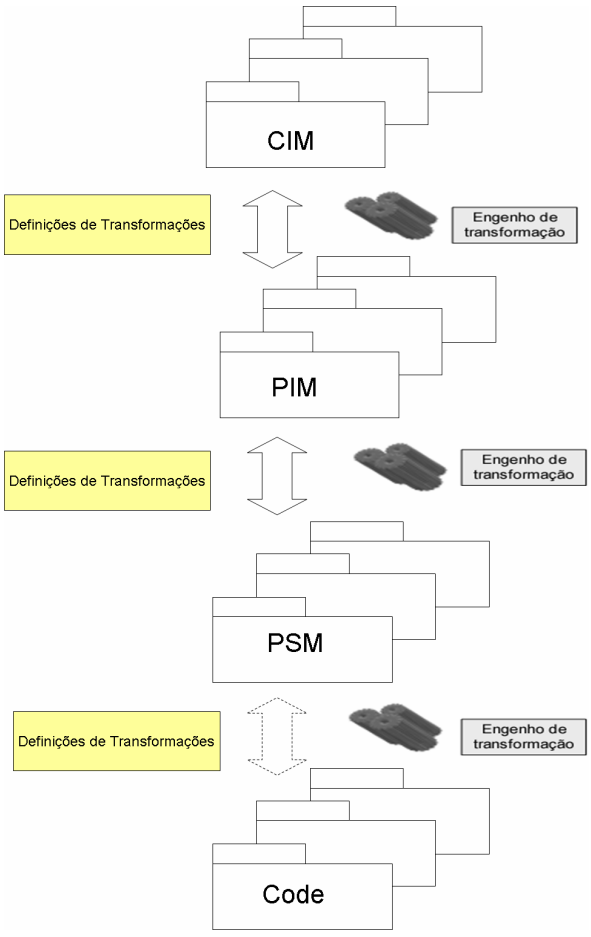


Figura 20: Relação entre os níveis de abstração do MDA

Além disso, ainda inserido no MDD, temos que um modelo precisa ser definido de acordo com uma semântica fornecida por um metamodelo. Da mesma forma, um metamodelo tem que estar em conformidade com um metametamodelo. Nessas três camadas de arquitetura (modelos, metamodelos, e metametamodelos), o metametamodelo está em conformidade com sua própria semântica (podem ser definidos usando seus próprios conceitos). Os metametamodelos existentes incluem o MOF (OMG/MOF, 2005), definido em padrão da OMG, e Ecore (BUDINSKY ET AL., 2004), que tem sido introduzido com o *Eclipse Modelling Framework* (BUDINSKY ET AL., 2004) e KM3 (JOUAULT; BÉZIVIN, 2006).

De uma forma geral, a transformação entre modelos é realizada em conformidade com metamodelos, que estão em conformidade com metametamodelos (ATLAS, 2006).

Portanto, como visão das abordagens das transformações entre modelos (ver Figura 21), temos a geração do modelo M_a em conformidade com o metamodelo MM_a, que é transformado em um modelo M_b em conformidade com o metamodelo MM_b. A transformação é definida por um modelo de transformação M_t, que está em conformidade com um metamodelo MM_t. Este último metamodelo, juntamente com os metamodelos MM_a e MM_b, tem conformidade com o um metametamodelo (tal como Ecore ou MOF).

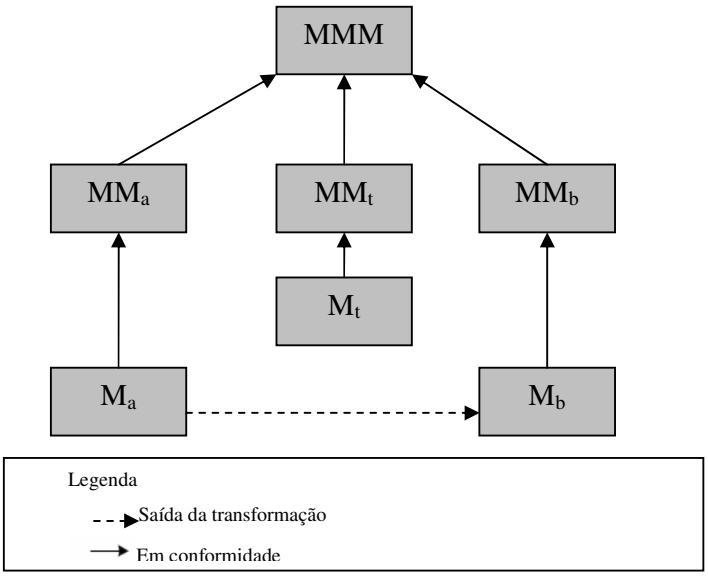


Figura 21: Visão das abordagens das transformações entre modelos.

Existem algumas linguagens para transformações entre modelos: QVT *(Query, View, Transformations)* (OMG/QVT, 2005) e ATL (Atlas Transformation Language) (ATLAS, 2008). ATL foi escolhida nesse trabalho por ser uma linguagem de transformação hibrída, pois aceita tanto construçãoes declarativas quando construções imperativas. Foi desenvolvida pela AMMA (*ATLAS Model Management Architecture*) e é aplicável em vários cenários de transformação entre modelos.

2.6.1. ATL

ATL é uma linguagem de transformação entre modelos especificada com uma sintaxe textual concreta e fornece aos desenvolvedores caminhos para especificar um número de modelos a serem produzidos baseados em algum modelo de entrada (ATL, 2006).

As transformações ATL são compostas de regras que definem como os elementos de um modelo de entrada são compostos e percorridos para criar os elementos que irão compor o modelo de saída (ATLAS, 2006). ATL é composta por várias unidades: *ATL Module, ATL Query, e ATL Library*. Na nossa abordagem utilizamos as unidades de *ATL Module* e *ATL Library.*

A unidade *ATL Module* contém especificações responsáveis por gerar modelos de saída a partir de modelos de entrada. Os modelos devem ser "tipados" com seus respectivos metamodelos. *ATL module* é composta por:

- Uma seção de cabeçalho que define atributos relacionados ao módulo de transformação (linhas 1 e 2 da Figura 23).
- Uma seção de importação que habilita a importação de *ATL Libraries* existentes.
- Uma parte de *Helpers*
- Uma parte de regras que definem caminhos para gerar modelos de saída de determinados modelos de entrada. (linhas 4 a 12 da Figura 23).

Outra unidade que compõe ATL é a unidade *ATL Query* que consiste em um modelo para transformação de valores referentes a tipos primitivos. O uso mais comum de *ATL Query* é a geração de saída textual a partir modelos de origens. Contudo, as *ATL queries* não são limitadas para a computação de valores de *string,* também podem retornar valores numéricos e *boolean* (ATL, 2006).

Outra unidade que compõe ATL, a *ATL Library*, que pode ser usada opcionalmente por outras unidades através da importação. A *ATL Library* não possui um módulo específico, e não é executada de forma independente. Dessa forma, é impossível declarar *Helpers* (equivalentes a métodos) nessa unidade, eles são definidos no contexto dos módulos de ATL.

Ainda, para a execução de aplicações práticas em ATL é necessário o suporte de ferramentas compilador/interpretador, ambiente de desenvolvimento, *debugger, profiler,* etc.

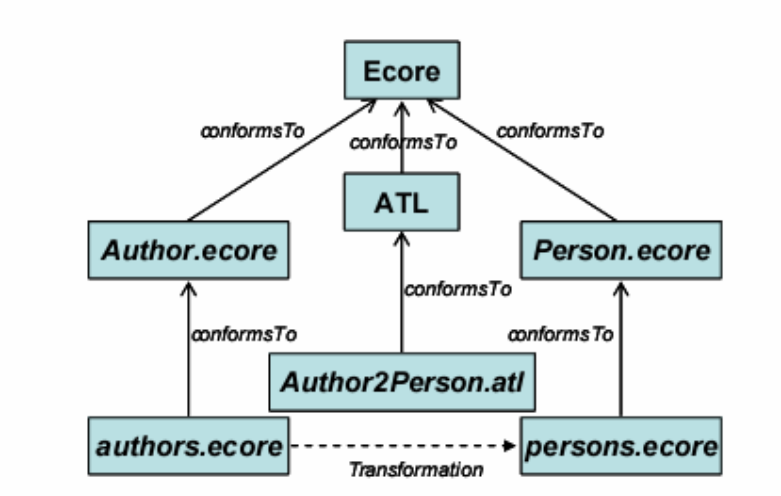


Figura 22: Visão da transformação de Autor para Pessoa (ATLAS, 2006).

A Figura 22 fornece uma exemplo do processo de transformação de *Autor* para *Pessoa*. Temos os modelos (*authors.ecore, persons.ecore),* os metamodelos (*Author.ecore, Person.ecore)* e a transformação ATL (*Author2Person.atl)*. Os arquivos utilizados na transformação são manuseados e utilizados no escopo das ferramentas de desenvolvimento de suporte a ATL e ao longo da transformação em ATL de *Author* para *Pessoa*.

Na Figura 23 está representado o código da transformação em ATL de *Author* para *Pessoa* (*Author2Person.atl)*. Nesse exemplo, temos a transformação de elementos de *Author* (do metamodelo *Author*), em elementos de Pessoa (do metamodelo *Pessoa*).. Essa regra ATL é composta de um padrão de entrada, no qual a entidade de origem *Author* é declarada (linha 2). Essa entidade é "tipada" por *Author!Author (*par *metamodel_name!entity_name*) (linha 6). Da mesma forma, ocorre com o padrão de saída , no qual a entidade de saída é "tipada" por *Person!Person* (linha 8). Esse padrão

de saída também define duas ligações, que especificam os elementos de Pessoa que serão inicializados pela regra. Nesse caso temos os elementos *name* e *surname* (linhas 9 e 10), que serão transformados de elementos de Pessoa para os elementos correspondentes em Author, *a.name e a.surname* (linhas 9 e 10).

```
1   module Author2Person;
2   create OUT : Person from IN : Author;
3
4    rule Author2Person {
5     from
6             a : Author!Author
7     to
8             p : Person!Person (
9                     name <- a.name
10                    surname <- a.surname
11            )
12  }
```

Figura 23: Exemplo de Transformação ATL de Autor para Pessoa.

2.6.2. KM3

Para descrição e criação de metamodelos é usado o formato KM3 (*Kernel MetaMetaModel*), que surgiu como resposta a freqüentes questões de usuários que utilizavam transformação em modelos utilizando a linguagem de transformação ATL. A OMG tem proposto o MOF (OMG/MOF, 2004) para definição de metamodelos, porém o suporte a essa linguagem não é uma ambiente é prático para o desenvolvimento, e está associada as definições de UML, dificultando a manutençãoda linguagem e uma semântica precisa Assim, KM3 é uma linguagem que fornece semântica para descrever metamodelos e para definir linguagens específicas de domínio (DSL) e usa notação textual semelhante a Java (SUN, 2007) para expressar os metamodelos. KM3 foi definida pelo grupo INRIA (INRIA, 2008), e está disponível para ser usado na plataforma Eclipse (Eclipse, 2007. A sintaxe abstrata do KM3 (o metamodelo KM3, ver Figura 24), é baseada em Ecore e eEMOF 2.0 (OMG/MOF, 2004). Além disso, os arquivos definidos como .km3 podem ser transformados em metamodelos e serializados em formato XMI (OMG/XMI, 2003).

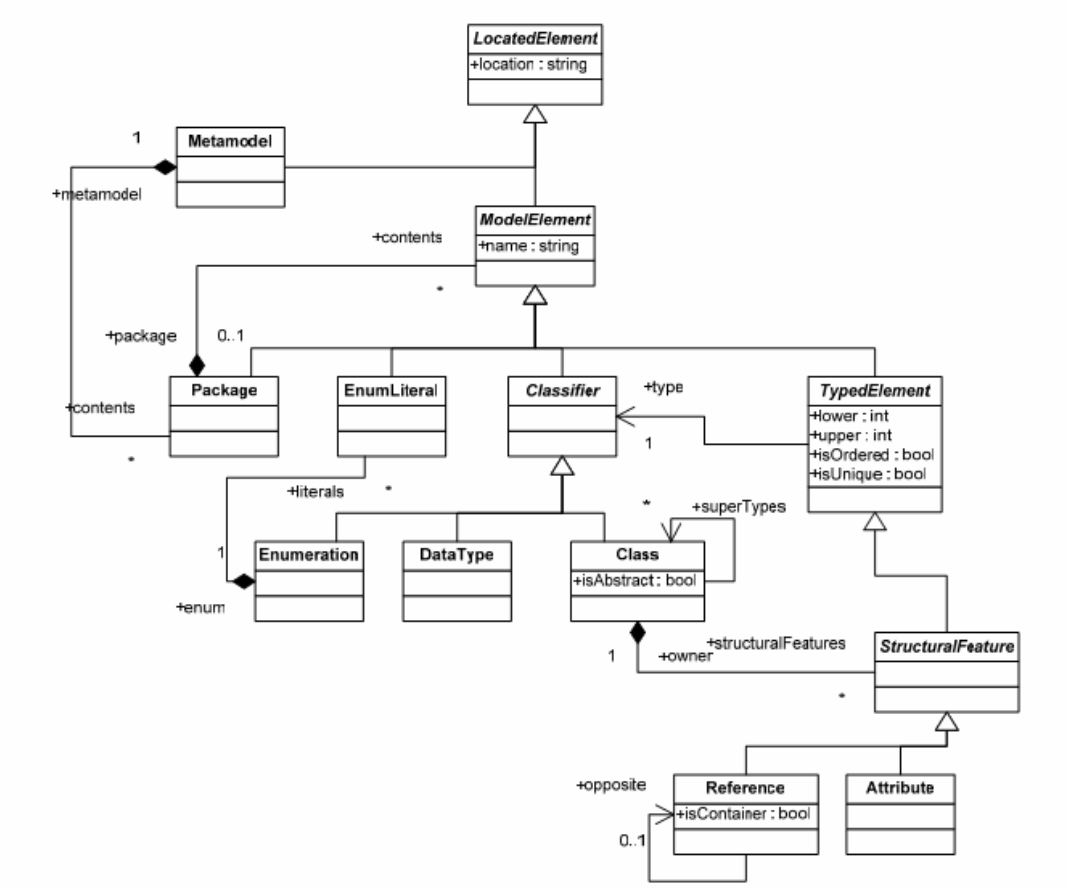


Figura 24: Metamodelo KM3

O código mostrado na Figura 25, descreve a especificação KM3 do metamodelo KM3 mostrado na Figura 24. Nessa figura, temos a descrição do metamodelo KM3 que é composto por elementos que formam o *Package* (pacote). Esse *Package* contém, por exemplo (destacado em cinza), a entidade *ModelElement* (TypedElement, Classifier, EnumLiteral e elemento de *Package*). Além disso, cada *ModelElement* é caracterizado por um nome , e elemento referenciando e do tipo *Package.*

```
package PrimitiveTypes {
        datatype Boolean;
        datatype Integer;
        datatype String;
}
package KM3 {
        abstract class LocatedElement {
                attribute location : String;
        }
        abstract class ModelElement extends LocatedElement {
                attribute name : String;
                reference "package" : Package oppositeOf contents;
        }
        class Classifier extends ModelElement {
```

```
	}
	class DataType extends Classifier {
	}
	class Enumeration extends Classifier {
		reference literals(*) ordered container : EnumLiteral oppositeOf enum;
	}
	class EnumLiteral extends ModelElement {
		reference enum : Enumeration oppositeOf literals;
	}
	class TemplateParameter extends Classifier {
	}
	class Class extends Classifier {
		reference parameters[*] ordered container : TemplateParameter;
		attribute isAbstract : Boolean;
		reference supertypes[*] : Class;
		reference structuralFeatures[*] ordered container : StructuralFeature oppositeOf owner;
		reference operations[*] ordered container : Operation oppositeOf owner;
	}
	class TypedElement extends ModelElement {
		attribute lower : Integer;
		attribute upper : Integer;
		attribute isOrdered : Boolean;
		attribute isUnique : Boolean;
		reference type : Classifier;
	}
	class StructuralFeature extends TypedElement {
		reference owner : Class oppositeOf structuralFeatures;
		reference subsetOf[*] : StructuralFeature oppositeOf derivedFrom;
		reference derivedFrom[*] : StructuralFeature oppositeOf subsetOf;
	}
	class Attribute extends StructuralFeature {
	}
	class Reference extends StructuralFeature {
		attribute isContainer : Boolean;
		reference opposite[0-1] : Reference;
	}
	class Operation extends TypedElement {
		reference owner : Class oppositeOf operations;
		reference parameters[*] ordered container : Parameter oppositeOf owner;
	}
	class Parameter extends TypedElement {
		reference owner : Operation oppositeOf parameters;
	}
	class Package extends ModelElement {
		reference contents[*] ordered container : ModelElement oppositeOf "package";
		reference metamodel : Metamodel oppositeOf contents;
	}

	class Metamodel extends LocatedElement {
		reference contents[*] ordered container : Package oppositeOf metamodel;
	}
}
```

Figura 25: Descrição de KM3 utilizando a própria linguagem

2.6.3. TCS

O TCS (*Textual Concret Syntax*) (JOUAULT, 2006) surgiu para a realização da transformação de modelo para texto, e vice-versa, assim como geração de especificações de forma bidirecional. O TCS é um componente Eclipse/GMT que fornece especificação de sintaxes textuais concretas para Linguagens Específicas de

Domínio (DSLs), e possui suporte de edição no Eclipse. As construções utilizando TCS mostram como se pode estabelecer a correspondência entre elementos dos metamodelos previamente definidos e sintaxes textuais de suas representações.

A Figura 26 ilustra a visão do uso de TCS, mostrando como um modelo M_L é extraído para SM_L. Vamos assumir que desejamos construir uma DSL (Linguagem específica de domínio) chamada L. Na parte nomeada MDE TS (ver Figura 26) fornecemos um metamodelo de L nomeado MM_L expresso utilizando KM3. A definição da sintaxe concreta é expressa em TCS e é denotada como CS_L. Ainda, temos os chamados *injector* e *extractor.* O *injector* recebe um modelo em L, expresso na sintaxe textual concreta de L, e gera um modelo em conformidade com MM_L na parte MDE TS. Um exemplo do modelo é denotado SM_L, e está em conformidade com a gramática de L denotada G_L. G_L é expressa em ANTLR. O extrator gera um representação textual dos modelos em MDE TS em conformidade com MM_L.

Partindo-se do metamodelo e descrição da sintaxe textual concreta de uma dada Linguagem L, o objetivo é obter três entidades de L: uma gramática G_L expressa em ANTLR, e o par de *injector* (injetor) e *extractor* (extrador). G_L é obtido por uma transformação ATL (*TCS2ANTLR.atl).* Essa transformação recebe MM_L e CS_L como entrada e gera as regras e anotações em GL. Essa gramática é usada para gerar o *injector.* O *injector* é um *parser* gerado de ferramentas fornecidas por tecnologia ANTLR. A geração é feita pelo *parser* ANTLR (*ANTL GEN).*

O *extractor* trabalha na representação interna de modelos expressos em L e na criação de sua representação textual. Dessa forma, é possível gerar um extractor da linguagem L. Entretanto, podemos ter outra abordagem na qual um *extractor* simples é implementado como um interpretador que trabalha para todas as linguagens. O *extractor* recebe um modelo M_L escrito em L, seu metamodelo MM_L, e sua descrição CS_L, em conformidade com sintaxe TCS, e gera representação textual SM_L de M_L.

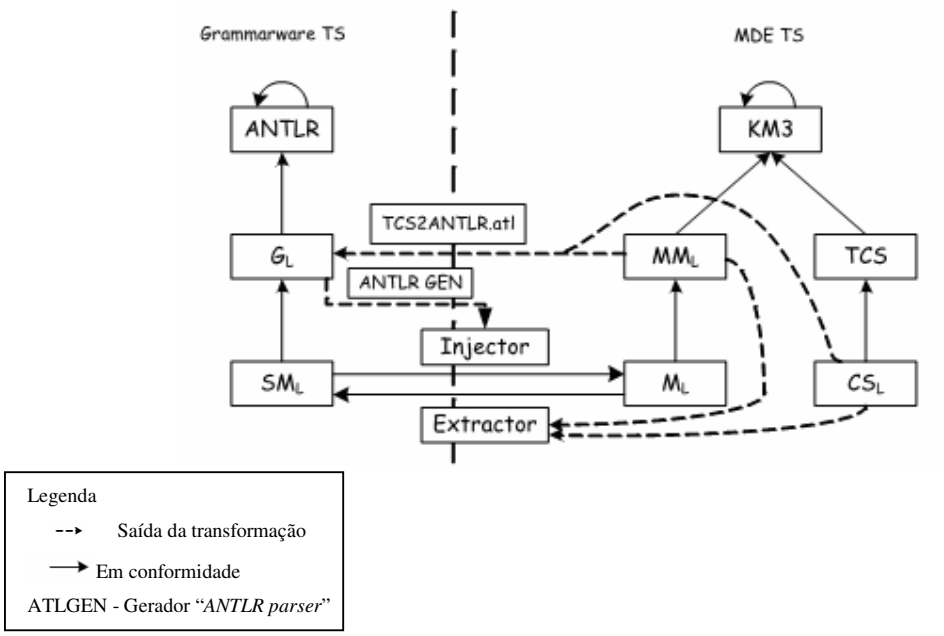


Figura 26: Visão do Uso de TCS (JOUAULT, 2006)

3. Mapeamento entre Modelos Orientados a Aspectos: dos Requisitos ao Projeto Detalhado

No processo de desenvolvimento de softwre as informações de requisitos, arquitetura e projeto detalhados são, em geral, isoladas, o que ocasiona uma distância entre os artefatos produzidos em cada atividade. As transformações entre modelos orientados a aspectos proporcionam a propagação de informações entre as fases do ciclo de desenvolvimento de software, e o tratamento das informações provenientes dessas fasess de uma forma não separada. Nessa dissertação partimos de requisitos, passando pela arquitetura de software, até o projeto detalhado. Na Seção 3.1. serão apresentadas as regras definidas nesse trabalho para a transformação entre AOV-graph e AspectualACME, e vice-versa. A Seção 3.2 mostra as regras de transformação entre AspectualACME e aSideML.

Apesar de AOV-graph, AspectualACME e aSideML terem sido desenvolvidas independentemente, elas compartilham algumas características. No caso de AOV-graph e AspectualACME: (i) qualquer elemento (requisito ou componente) pode entrecortar qualquer outro elemento, transversal ou não-transversal; (ii) ambas são estratégicas simétricas orientadas a aspectos e, consequentemente, elementos transversais ou não transversais são representados pela mesma abstração, a distinção entre elementos transversais é estabelecida baseada na maneira como eles interagem (ou compõem) com o restante do sistema; (iii) há um relacionamento especial ou conector que estende as linguagens de modelagem como forma de dar suporte a composição de conceitos que estão espalhados e entrelaçados no sistema. Entretanto, a transformação entre AOV-graph e AspectualACME não é trivial pois, apesar de haver intersecções sintáticas e semânticas entre seus elementos estruturais, há elementos e informações particulares de cada uma destas linguagens.

No que diz respeito à AspectualACME e aSideML: (i) focam na modelagem de características transversais, (ii) propõem modelos de propósito geral para representação de sistemas orientados a aspectos; (iii) possuem diferenças em relação a forma de modelar componentes e aspectos, uma vez que AspectualACME não tem representações diferentes para componentes e aspectos; (iv) na descrição AspectualACME resultante do mapeamento AOV-graph, as informações sobre *intertype declaration* estão representadas através de propriedades em AspectualACME e podem ser mapeadas para

aSideML. No entanto, na descrição AspectualACME não gerada a partir de AOV-graph, não há informação sobre *intertype declaration*, uma vez que originalmente AspectualACME não representa tal informação. Nesse caso, não há como gerar um modelo aSideML com *intertype declaration*; (v) AspectualACME não oferece suporte para informar se a característica transversal refina ou redefine alguma operação. Essa característica da linguagem impossibilita a separação dos refinamentos e redefinições no mapeamento para aSideML.

Além disso, essas transformações, que estão inseridas dentro do processo MDD, podem ser relacionadas com os níveis de abstração defindas na Arquitetura Orientada a Modelos (MDA). A Figura 27 mostra que definimos transformações entre os níveis CIM (AOV-graph), para PIM (AspectualACME), e de PIM (AspectualACME) para outro PIM (aSideML).

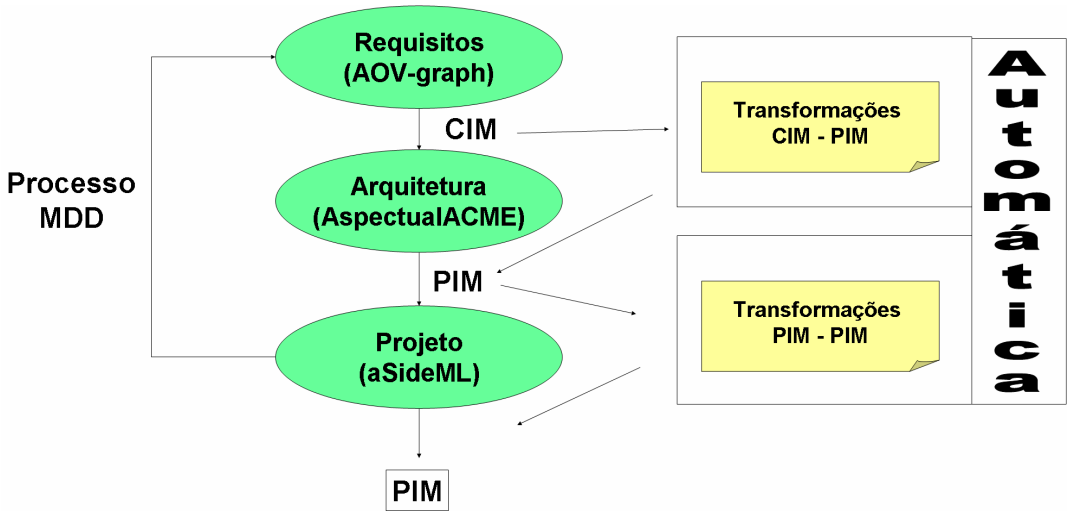


Figura 27: Representação das Transformações dos Requisitos ao Projeto Detalhado no contexto do MDA.

A seguir detalhamos o mapeamento entre AOV-graph e AspectualACME, e vice-versa, e entre AspectualACME e aSideML.

3.1. Mapeamento de AOV-graph para AspectualACME, e vice-versa.

A transformação entre elementos de AOV-graph e AspectualACME significa estabelecer uma associação entre essas duas linguagens de forma que seja possível propagar mudanças entre seus elementos rapidamente e navegar do modelo de requisitos para arquitetura. AOV-graph e AspectualACME focam na modelagem de características transversais e propõem modelos de propósito geral, baseados em algumas

extensões simétricas da orientação a aspectos. Em ambas as linguagens considera-se que o espalhamento e entrelaçamento de características transversais são representados pela maneira como essas características interagem umas com as outras. Tais informações sobre a interação estão nos relacionamentos transversais, no caso de AOV-graph, e nos conectores aspectuais e *attachments*, no caso de AspectualACME. Assim, cada relacionamento transversal é transformado para conectores aspectuais e *attachments*, e vice-versa.

Os conceitos do sistema são representados em AOV-graph por *softmetas, metas* e *tarefas,* e em AspectualACME por *componentes* e *portas*. A transformação entre estes elementos é possível porque há uma correspondência semântica entre *softmetas/metas/tarefas* e *componentes/portas*: todos eles podem ser vistos como serviços a serem providos ou requeridos pelo sistema ou partes dele. Assim, embora softmetas, metas e tarefas estejam em diferentes níveis de abstração, eles são transformados para serviços providos ou requeridos na arquitetura de software. Por outro lado, componentes representam serviços providos pelo sistema e portas representam serviços providos ou requeridos pelos componentes para alcançarem seus objetivos. Apresentamos os detalhes desta transformação na Seção 3.1.1.

3.1.1. Mapeamento de AOV-graph para AspectualACME

O processo de transformação de AOV-graph para AspectualACME ocorre em quatro etapas:

(i) Transformação da hierarquia de metas, softmetas e tarefas de AOV-graph para componentes, portas e representações (*representations*) de AspectualACME – Cada elemento (meta/*softmeta*/tarefa) de AOV-graph é transformado para um componente ou porta de AspectualACME. Essa decisão é baseada na posição em que cada meta/*softmeta*/tarefa está na hieraquia: elementos que são raízes da árvore de metas são transformados para componentes; subelementos que não são folhas (na árvore AOV-graph) são transformados para componentes em *representações,* e subelementos que são folhas são transformados para portas e são inseridos no componente pai correspondente. Para complementar, quando um componente e uma porta são gerados algumas propriedades (*properties)* também são inseridas para registrar a origem de cada um deles. Essas propriedades registram algumas informações sobre requisitos no

modelo arquitetural e possibilitam a transformação inversa – de AspectualACME para AOV-graph e facilitam o rastreamento.

(ii) Transformação de *pointcuts* e *intertype-declarations* de AOV-graph para portas e ligações (*bindings)* de AspectualACME – Alguns conceitos nos relacionamentos transversais também podem ser transformados para portas: (i) cada elemento referenciado (tarefa/meta/*soft*meta) no corpo de uma *intertype declaration;* (ii) cada elemento (tarefa/meta/*soft*meta) definido no corpo de um *advice* ; e (iii) cada elemento (tarefa/meta/*soft*meta) definido no corpo *pointcut.* Semanticamente, relacionamentos transversais agrupam diversos relacionamentos entre metas*,* tarefas *e soft*metas*,* ou seja, eles representam as interações que, na arquitetura, são representadas por conectores ligando portas, ou seja, especificando ligações entre serviços. Adicionalmente, se o elemento definido como origem *(source)* do relacionamento transversal referenciar o mesmo serviço de um componente com mesmo nome, ele é mapeado para uma porta de mesmo nome, e para dentro do corpo do elemento pai desse componente com o mesmo nome. Assim como é criada um porta *self,* dentro desse componente, e uma ligação da porta que referencia o mesmo serviço do componente para a porta *self.*

(iii) Transformação da propriedade *correlation* de AOV-graph para a propriedade *correlations* de AspectualACME – Em AOV-graph os elementos especificados como elementos alvo na *correlation,* são transformados e armazenados no atributo *correlations* dentro das propriedades do elemento especificado como origem na *correlation* em AOV-graph.

(iv) Transformação de relacionamentos transversais para conectores e configurações (*attachments)* – Conectores e configurações *também são* gerados a partir do relacionamento transversal: (i) cada *intertype declarations* e *advice* definidos no corpo do relacionamento transversal originam conectores aspectuais, e essa origem é armazenada no atributo *comesFrom* das propriedades desse conector, sendo que o tipo da cláusula *glue* é determinado pelo tipo do *advice* (podendo ser *around, after* ou *before*), ou pelo tipo de *intertype declaration* (sendo *around* se a intertype for do tipo *element* e valor definido pelo atributo criado, se for do tipo *attribute*); (ii) cada relação definida no *intertype declaration* → *pointcut* e no *advice* → *pointcut* é transformada

para uma associação de configuração de elementos, do *intertype declaration* para uma *crosscutting role,* e uma outra associação, de *base role* para elementos dos *pointcuts.* (iii) o elemento definido com origem (*source*) do relacionamento transversal é mapeado para o atributo *crossRelSource* das propriedades do conector aspectual de AspectualACME.

A Tabela 1 resume as regras de transformação entre AspectualACME e AOV-graph:

Tabela 1. Transformação AOV-graph para AspectualACME

Elemento de AOV-graph		Elemento de AspectualACME
Meta, softmeta, tarefa	→	Componente ou porta + Propriedade elementType
Correlation	→	Propriedade correlations
Topic	→	Propriedade topic
Advice	→	Conector Aspectual + Propriedades comesFrom
Intertype declaration	→	Conector Aspectual + Attachment + Propriedade comesFrom
Elementos do corpo do *Intertype Declaration*	→	Porta + Propriedade elementType
Advice type	→	glueType
Intertype declaration do tipo element	→	glueType = around
Intertype declaration do tipo attribute	→	glueType = valor do atributo
Source (crosscutting relationship)	→	Propriedade *crossRelSource*
Pointcut	→	Attachment, port self + binding

3.1.2. Mapeamento de AspectualACME para AOV-graph

Na transformação de AspectualACME para AOV-graph precisamos considerar duas situações: (i) a arquitetura em AspectualACME foi gerada com base em um modelo AOV-graph, desta forma há informações da modelagem de requisitos na especificação da arquitetura; (ii) a arquitetura foi gerada independentemente de um modelo de requisitos em AOV-graph, assim, não há informações sobre o modelo de metas. Isto pode ocorrer, por exemplo, na engenharia reversa ou quando outro modelo de requisitos deu apoio à modelagem da arquitetura.

Na primeira situação as informações registradas em propriedades (*properties*) guiam a transformação, dando apoio às decisões acerca do: (i) tipo de elemento, se é softmeta, meta ou tarefa; (ii) como os conceitos estão transversais, se via *advice* ou *intertype declaration*; (iii) correlações existentes no modelo de metas; (iv) rótulos das contribuições e correlações.

Para a segunda situação, visto que as informações citadas acima não estão disponíveis então algumas observações precisam ser feitas: (i) a hierarquia de componentes e portas deve ser transformada para uma hierarquia de metas, softmetas e tarefas, mas não há como determinar precisamente qual o tipo de cada elemento, assim, se a propriedade *elementType* não estiver disponível, portas são transformadas para tarefas e componentes são transformados para softmetas; (ii) o rótulo das contribuições entre metas, softmetas e tarefas não podem ser identificados precisamente, mas como determinado no próprio AOV-graph, o rótulo *and* é tido como *default*; (iii) o tipo de *glue* contido nos conectores aspectuais determina se aquele conector é transformado para: (a) um *advice* – advice dos tipos *before* e *after* podem ser diretamente identificados (b) uma *intertype declaration* do tipo *attribute* ou do tipo *element* – os *intertype declarations* do tipo *attribute* são geradas por *glue* que são diferentes de *around, before* e *after*. Porém advice do tipo *around* e *intertype declarations* do tipo *element* não podem ser distinguidos, pois possuem o *glue* do mesmo tipo. Assim, determinamos se o conector aspectual gera advice ao invés de *intertype declarations* através da especificação contida no atributo *comesFrom* das propriedades do conector aspectual*;* (iv) As correlações do AOV-graph não podem ser geradas, visto que elas representam associações abstratas entre *soft*metas e metas.

Além disso, assim como na transformação de AOV-graph para AspectualACME, onde as informações específicas deste modelo de requisitos são registradas em um elemento de AspectualACME (nas propriedades), é necessário que AOV-graph seja capaz de registrar informações arquiteturais especificadas em AspectualACME. Por exemplo, informações tais como aquelas relativas ao estilo arquitetural utilizado, ou o tipo de conexão existente entre os componentes. Assim, adicionamos um novo elemento à linguagem AOV-graph denominado *property*, que é idêntico ao elemento de mesmo nome em AspectualACME. Estas propriedades podem estar associadas ao modelo de metas (*goal_model*), softmeta, meta, tarefa, correlações e contribuições. Dado que tais informações arquiteturais são registradas em propriedades de AspectualACME então os valores de tais propriedades em AOV-graph são determinados pelos próprios valores registrados em AspectualACME (veja a última linha das Tabelas 1 e 2).

O processo que transforma especificações em AspectualACME para modelos em AOV-graph consiste das seguintes etapas:

(i) Transformação de conectores aspectuais e *attachments* para relacionamentos transversais – cada conector aspectual é transformado para uma relacionamento transversal com uma *intertype declaration* ou *advice* em AOV-graph, determinados pelo atributo *comesFrom* especificado nas propriedades dos conectores aspectuais. O tipo da cláusula *glue* é verificado e transformado para especificar o tipo deste *advice* e *intertype declaration,* enquanto seus *pointcuts* e corpo são determinados pelos *attachments.* Para cada *attachment* que possui o mesmo conector ligando portas e papéis: (i) portas relacionadas com o *crosscuttingRole* são transformadas para o corpo do *intertype declaration* ou *advice* (dependendo do tipo da cláusula *glue*); (ii) o componente que possui esta porta é transformado para a origem do relacionamento transversal no AOV-graph; (iii) portas ligadas ao *BaseRole* são transformadas para *pointcuts.* No final da transformação é necessário reunir todos os relacionamentos transversais que possuem a mesma origem, que está especificada pelo atributo *crossRelSource* das propriedades do conector aspectual, bem como todos os pontos que são atingidos pelo mesmo *intertype declaration* ou *advice*.

(ii) Transformação de componentes, portas, representações *(representations)* e propriedades para metas, tarefas ou softmetas de AOV-graph – (i) A partir das propriedades é possível identificar precisamente qual o tipo do elemento derivado de componentes ou portas, através da propriedade *elementType* e através da propriedade *topics* identificamos quais *strings* do nome da meta, *soft*meta ou tarefa deve ser posta entre colchetes. (ii) Quando informações sobre o AOV-graph não estão disponíveis – a hierarquia de componentes e portas gera uma hierarquia de *soft*metas e tarefas, sendo as tarefas derivadas de portas e as *soft*metas derivadas de componentes. Entretanto, é necessário analisar se cada um destes componentes e portas faz ou não parte do corpo de alguma *intertype declaration*. Se fizerem, então eles não são transformados para *soft*metas e tarefas, e eles existirão apenas no corpo da *intertype declaration*.

(iii) Transformação da propriedade *correlations* de AspectualACME para *correlation* de AOV-graph – Em AspectualACME os elementos alvo associados ao elemento de origem são especificados no atributo *correlations* das propriedades do elemento de origem especificado em AOV-graph. Dessa forma, as *correlations* de AspectualACME são transformadas para *correlation* em AOV-graph, sendo

especificado o elemento de origem, que é o elemento que possui a propriedade em AspectualACME, e os elementos alvo, que são os especificados dentro do atributo *correlations* das propriedades em AspectualACM do elemento de origem.

(iv) Transformação de conectores de AspectualACME para contribuições de AOV-graph – Os conectores não-aspectuais de AspectualACME representam relacionamentos entre portas que requerem serviços e portas que provêem serviços. Em AOV-graph tais relacionamentos são representados por contribuições. Assim, um serviço provido contribui para duas ou mais softmetas, metas e tarefas. Tendo em vista a maneira como AOV-graph foi implementado, a tarefa derivada da porta provedora é distinta da tarefa derivada da porta que requer o serviço, pois a primeira é transformada para uma tarefa de fato, enquanto a segunda é derivada para uma referência à primeira.

A Tabela 2 resume as regras de transformação entre AspectualACME e AOV-graph:

Tabela 2. Transformação AspectualACME para AOV-graph

Elemento de AspectualACME		Elemento de AOV-graph
Componente, porta	→	Meta ou *soft*meta ou tarefa + Propriedade
Conector Aspectual	→	Advice, intertype declaration + Propriedade
glueType	→	AdviceType, intertypeDeclarationType
Attachments	→	Pointcut, source, contribution (se não está associado a um conector aspectual)
Conector	→	Contribution + Propriedade
Propriedade topic	→	Topic
Propriedade elementType	→	Determina se o elemento é uma meta, softmeta ou tarefa
Propriedade correlations	→	Correlation
Propriedade comesFrom	→	Determina se os conceitos transversais devem ser descritos em advice ou intertype declaration
Propriedade crossRelSource	→	Auxilia na identificação da origem do relacionamento transversal

3.2. Mapeamento entre AspectualACME para aSideML.

A transformação entre AspectualACME e aSideML proporciona a associação dos elementos dessas linguagens de forma que é possível propagar as mudanças e navegar da fase de arquitetura para o projeto detalhado.

O processo de transformação entre AspectualACME e aSideML consiste em: (i) transformar os componentes bases e aspectuais de AspectualACME para classes e aspectos, respectivamente, em aSideML; (ii) transformar as portas dos componentes aspectuais do modelo arquitetural para interfaces transversais, e as portas dos componentes base para interfaces comuns, do modelo de projeto; (iii) transformar os

conectores aspectuais ou regulares para colaborações entre classes e aspectos. A seguir, detalhamos as regras de transformação.

(i) Transformação de Componentes para Classe ou Aspecto – Os componentes, na linguagem AspectualACME, encapsulam um conjunto de funcionalidades, representando elementos transversais ou não. Dessa forma, eles correspondem às classes ou aos aspectos da linguagem aSideML. A diferenciação entre classe e aspecto é extraída por meio de análise das configurações (*attachments)* da descrição arquitetural.

Se o componente possui uma (ou mais de uma) porta que está ligada a uma *crosscutting role* de algum conector aspectual, ele representa um aspecto, já que esse componente implementa uma característica transversal.

Se o componente não possui porta ligada a uma *crosscutting role* ele representa uma classe, em aSideML. Ou seja, se esse componente possui portas que não estejam ligadas a nenhum conector, ou ligadas a um conector comum ou a uma *base role* de um conector aspectual, ele não especifica nenhum comportamento transversal e é transformado em uma classe comum.

(ii) Transformação de Portas e Interface para Interface Transversal – As portas, em AspectualACME, definem serviços que um componente oferece ou requisita. Como esse é o mesmo conceito de interfaces, a definição de interfaces em aSideML é feita através da análise das portas dos componentes da descrição arquitetural.

As possibilidades de transformação de portas são: (1) *Transformação de Portas para Interface* – Se o componente da arquitetura não implementa características transversais, suas portas são transformadas em operações e dispostas em uma interface. O nome da interface é obtido a partir do nome do componente, acrescentado de um 'I' no início. Como a linguagem aSideML deriva da linguagem de modelagem UML, essas portas estão representadas puramente com UML; (2) *Transformação de Portas para Interface Transversal* – As portas dos componentes que implementam características transversais são transformadas em características transversais em aSideML. Todavia, como as interfaces transversais são apenas conjuntos de características transversais, é necessário inicialmente definir as características transversais para, posteriormente, termos a interface transversal.

As características transversais de aSideML são definidas a partir da análise das portas dos componentes aspectuais. As transformações e identificação dessas características são:

(a) *Característica Transversal Estrutural* – A característica transversal estrutural especifica um atributo que será estaticamente introduzido na interface de uma ou mais classes base. Ela é definida a partir das declarações intertipo (*Intertype declarations)* do componente que implementa características transversais. Todas as portas de um componente aspectual que tem ligação com uma *crosscutting role* de um conector aspectual são transformadas para característica transversal. A referência à declaração intertipos é proveniente do modelo de requisitos, e transformada para AspectualACME;

(b) *Característica Requisitada Comportamental* – Essa característica define operações usadas pelo aspecto. Para que as características requisitadas comportamentais possam ser transformadas, é necessário que o componente arquitetural explicite o tipo de serviço especificado. Como as portas da linguagem AspectualACME definem serviços requisitados ou oferecidos, porém nenhuma distinção é feita entre esses dois tipos de serviços, as propriedades (*properties*) das portas podem especificar se cada porta é de entrada (serviço oferecido) ou de saída (serviço requisitado). Uma nova propriedade foi criada, com o nome *type*, que pode assumir dois valores: *required*, para portas que definem serviços requisitados e *provided* para portas que definem serviços oferecidos. A propriedade *type* tem o valor *provided* como *default*. Como as características requisitadas comportamentais em aSideML especificam os serviços usados pelo aspecto, todas as portas que possuem a propriedade *type* com o valor *required* são transformadas nessa característica.

(c) *Característica Transversal Comportamental* – Esta é uma especificação de uma fatia de comportamento que será adicionada a uma ou mais classes base, seja para refinar ou redefinir uma operação de uma ou mais classes base. Porém, a linguagem AspectualACME não oferece suporte para informar se a característica transversal refina ou redefine alguma operação. Essa característica da linguagem impossibilita a separação dos refinamentos e redefinições na transformação. Dessa forma, todas as portas de um componente de AspectualACME que define a característica transversal como não proveniente de uma declaração intertipos, que não tem uma propriedade *type* com valor *required,* mas que está relacionada a uma *crosscutting role* de algum conector Aspectual é transformada para característica comportamental transversal de aSideML. Opcionalmente, a porta pode possuir a propriedade type com valor *provided*. Neste

trabalho, as redefinições são tratadas como refinamentos, ou seja, não existe a diferenciação entre esses dois compartimentos da interface transversal.

A definição dos adornos das características transversais comportamentais, em aSideML, é feita através da transformação das especificações feitas na cláusula *glue* do conector que se liga à porta do componente aspectual. Dependendo de como a cláusula *glue* define o momento da interação aspectual (*after, before* ou *around*), as características transversais terão os adornos op_, _op ou _op_, onde op é a operação do aspecto.

Nas características transversais há ainda as transformações: (i) *Operações* – as portas dos componentes que implementam características transversais que não são transformadas para nenhuma das características transversais citadas anteriormente são transformadas para operações do aspecto, ou seja, são dispostos no compartimento de comportamento local do aspecto; (ii) *Interface Transversal* – Com a transformação das características transversais é possível ter a definição da interface transversal de aSideML apenas relacionado-se as características transversais já definidas nos compartimentos da interface. No compartimento *additions*, da interface transversal, estão todas as características transversais estruturais, já no compartimento *uses* estão todas as características requisitadas comportamentais, e no compartimento *refinements* estão todas as características transversais comportamentais. Todavia, devido a não ocorrer diferenciação entre o refinamento ou a redefinição, o compartimento *redefinitions* não é utilizado na transformação. A definição do nome da interface é feita recuperando o nome do aspecto relacionado a ela, acrescentado a letra 'I', no início.

A linguagem aSideML especifica que cada aspecto pode implementar mais de uma interface transversal. A implementação de mais de uma interface tem o objetivo de modularizar o aspecto, separando cada funcionalidade em interfaces diferentes. Porém, na transformação de AspectualACME para aSideML não é possível fazer essa separação de funcionalidades através da observação das portas. Portanto, nesta transformação, cada aspecto está relacionado a apenas uma interface transversal.

(iii) Transformação de Conector Regular para Relacionamento entre classes – Os conectores regulares, ou seja, aqueles que conectam componentes que não especificam comportamento transversal, são transformados em colaboração ou outros elementos comportamentais entre classes em aSideML.

(iv) Transformação de *Attachment*, Conector Aspectual, Conector Regular para relacionamento de *Crosscutting* ou Colaboração Aspectual – Os conectores aspectuais da descrição arquitetural são transformados para o relacionamento *crosscutting* entre uma classe base e um aspecto, ou para colaborações aspectuais, que descrevem as interações entre aspectos e classes, dependendo da origem especificada no conector aspectual. Se o conector aspectual tem origem a partir de: (i) declaração intertipos, então é transformado para relacionamento *crosscutting* em aSideML; (ii) *advice*, então é transformado em colaboração aspectual. Os detalhes contemplados nessa transformação são definidos:

(i) *Transformação de Conector Aspectual para Relacionamento Crosscutting* – A definição dos chamados *templates matches* do relacionamento *crosscutting* em aSideML é feita através da análise dos *attachments* dos conectores aspectuais de AspectualACME. O *template match pode* possuir duas formas: <*formalName* → *actualName*> ou <*formalName* → *all*>. A segunda forma oferece suporte ao uso de wildcards(*). Apenas a primeira delas está sendo contemplada nesse trabalho. Cada par de *attachment* é transformada para um relacionamento de *crosscutting,* onde f*ormalName* representa a porta ligada a uma *crosscutting role*, já o *actualName* representa a porta ligada a *base role*.

(ii) *Transformação de Conector Aspectual para Colaboração Aspectual* – Cada par de *attachment* de um conector aspectual que é proveniente de um *advice* é transformado para uma colaboração aspectual. O componente aspectual relacionado ao *attachement* corresponde ao aspecto da colaboração aspectual. O componente tradicional relacionado ao *attachment*, ou seja, aquele que possui sua porta ligada a uma *baseRole,* é transformado para o objeto Sender, caso sua porta tenha uma propriedade *required*, ou para o objeto base, caso a porta tenha uma propriedade *provided.*

É importante destacar que o par de *attachment* só será transformado para uma colaboração aspectual se a porta ligada ao conector aspectual for transformada em um *Refinement* (ou *Redefinition,* se esse estivesse sendo usado na transformação), pois uma colaboração aspectual só tem sentido quando relacionada ao tipo de extensão *refine* ou *redefinition.* Os pares de *attachments* que não se encaixam nesse requisito, por exemplo, aqueles cuja porta do conector aspectual é transformado para um *addition,* não são transformados para colaboração aspectual, mas apenas vistos num contexto de modelagem composicional, como classes combinadas.

(v) Transformação de Propriedades para Tags – A fim de propagar as informações vindas desde a descrição dos requisitos, as propriedades dos elementos da descrição arquitetural em AspectualACME são transformadas, em aSideML, para *tags* (ou notas), elementos relacionados à extensibilidade de UML.

A Tabela 3 resume as regras de transformação entre AspectualACME e aSideML:

Tabela 3: Transformação de AspectualACME para aSideML

Elemento de AspectualAcme		Elemento de aSideML
Componente	→	Classe ou Aspecto
Portas de um componente que não implementa característica transversal	→	Operações
Conjunto de portas de um componente que não implementa característica transversal	→	Interface
Portas de um componente que implementa característica transversal	→	Característica transversal estrutural, característica transversal comportamental, característica requisitada comportamental ou operações
Cláusula *glue*	→	Adornos nas características transversais comportamentais
Conjunto de portas de um componente que implementa característica transversal	→	Característica transversal
Conector Regular	→	Colaboração entre classes
Conector Aspectual	→	Colaboração Aspectual (se o Conector Aspectual possui com origem um advice), ou Relacionamento *crosscutting* (se o Conector Aspectual possui como origem *intertype declaration*)
Attachments do Conector Regular	→	Elementos de Colaboração (ou outro elemento comportamental)
Attachments do Conector Aspectual	→	Elementos do relacionamento crosscutting (se o Conector Aspectual possui como origem um intertype declaration) ou elementos da Colaboração Aspectual (se o Conector Aspectual possui como origem um advice).
Propriedades	→	Tags

4. MARISA-MDD

MaRiSA-MDD é uma abordagem para transformação entre modelos que integra as atividades de requisitos, arquitetura e projeto detalhado orientado a aspectos. Essa abordagem inclui descrição de metamodelos e um conjunto de mapeamentos entre modelos. Além disso, inclui um ambiente integrado com a IDE Eclipse (ECLIPSE, 2008), e seus *plugins*, que dão suporte as tecnologias e transformações MDD.

A proposta dessa abordagem é caracterizada por um processo rigoroso e coerente, onde cada atividade possui modelos OA representativos (e metamodelos correspondentes) e um conjunto de transformações entre modelos.

Dessa forma, como mencionamos no Capítulo 1, no nosso trabalho utilizamos a linguagem KM3 (Kernel MetaMetaModel) (JOUAULT ET AL., 2006) para descrição de metamodelos para AOV-graph, AspectualACME e aSideML. Assim como, o componente TCS (*Textual Concret Syntax*) (JOUAULT, 2006) para se obter a representação textual para modelos AOV-graph, AspectualACME e aSideML. Usamos a linguagem de transformação ATL (*ATLAS Transformation Language*) (ATL, 2008) para especificação das regras de transformação entre modelos. Dessa forma, MaRiSA-MDD tem o objetivo de a partir de especificações de requisitos em AOV-graph, gerar arquiteturas em AspectualACME e, em seguida, gerar projetos detalhados em aSideML.

Na Seção 4.1 é apresentada uma visão geral da abordagem, ilustrando os passos do processo das transformações de AOV-graph, AspectualACME, e aSideML. A Seção 4.2 contém algumas regras de transformação especificadas em ATL entre AOV-graph e AspectualACME, e vice-versa, e AspectualACME para aSideML. A Seção 4.4 mostra a classificação das regras de transformação. A Seção 4.4 descreve estudos de caso e ilustra exemplos da transformação de um modelo textual de entrada AOV-graph para um modelo textual de saída AspectualACME, e vice-versa, assim como de uma representação textual AspectualACME para uma representação textual aSideML. A Seção 4.5 mostra comentários sobre a aplicação das regras de transformação.

4.1. Visão geral da abordagem

A Figura 28 ilustra o processo principal de transformação de MARISA-MDD entre os modelos OA AOV-graph para AspectualACME, e AspectualACME para aSideML. O processo está organizado em duas fases: Elaboração e Verificação de Modelo (Fase 1) e Mapeamento e Verificação de Modelo (Fase 2). Os metamodelos

(KM3) de AOV-graph, AspectualACME e aSideML, bem como as regras de transformações com ATL e a descrição da BNF das três linguagens são definidos em uma “Fase 0”, de preparação. Essa fase foi omitida da Figura 1 por questões de simplicidade.

A entrada do processo de transformação entre AOV-graph, AspectualACME e aSideML é uma representação textual de um modelo AOV-graph. Tal modelo pode ser elaborado pelo arquiteto ou obtido a partir de um modelo AOV-graph, por meio da transformação AspectualACME para AOV-graph realizada pela ferramenta MARISA-MDD. Se alimentado manualmente, o modelo AOV-graph é verificado, de modo que apenas modelos bem-formados sejam usados no processo de transformação. Um modelo AOV-graph bem formado é o resultado produzido da Fase 1.

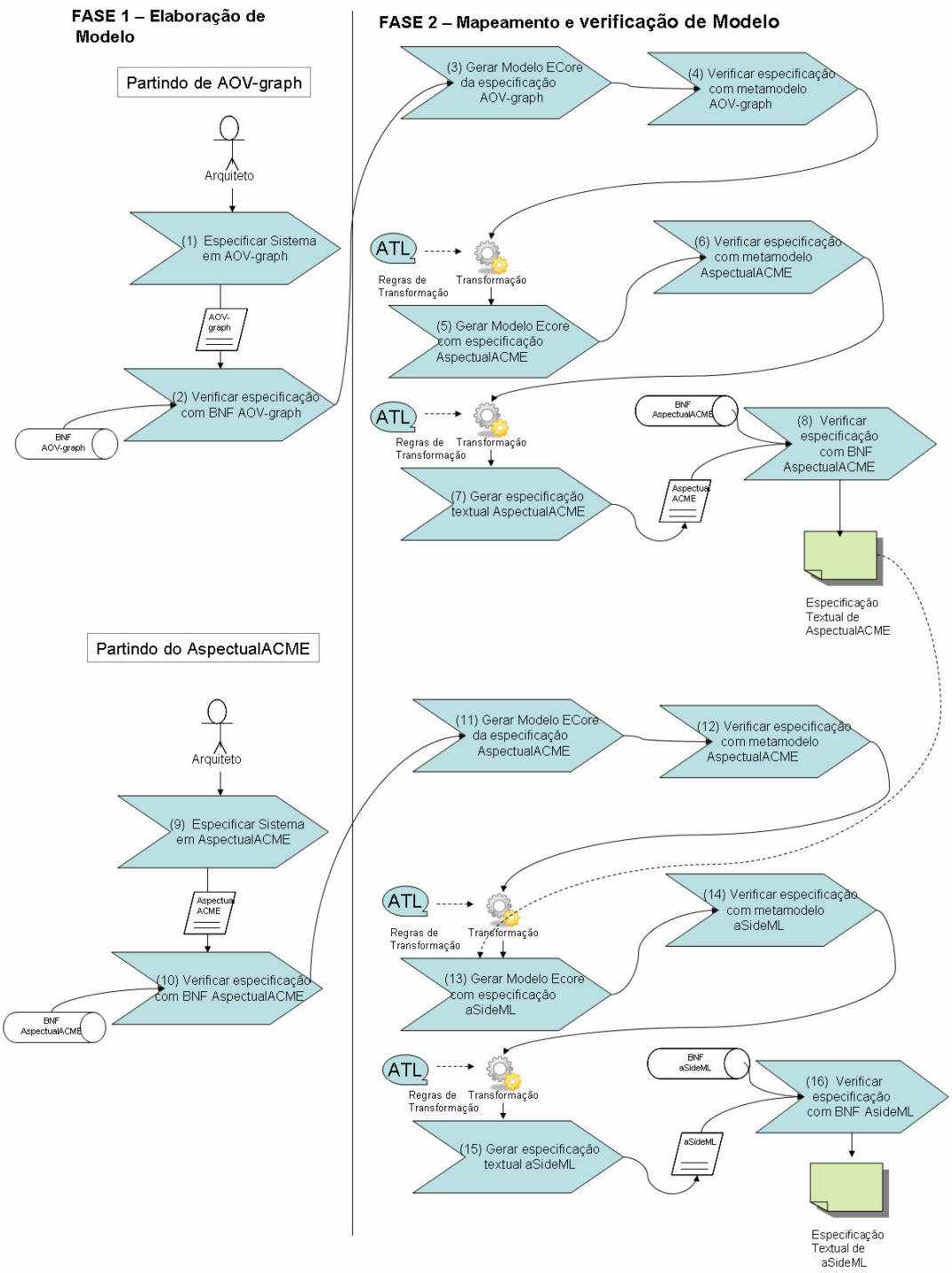


Figura 28: Processo mostrando os modelos de requisitos, arquitetura e projeto detalhado, atividades de definição e mapeamento entre modelos, e transformação.

Na Fase 2 é feita a transformação entre os modelos e a verificação de acordo com os metamodelos. O modelo textual de AOV-graph é transformado para o modelo ECORE do AOV-graph, e esse é verificado conforme o metamodelo AOV-graph

(definido em KM3). Assim, a área de atuação de ATL é a transformação entre o modelo ECORE do AOV-graph para o modelo ECORE AspectualACME, verificado conforme o metamodelo AspectualACME (definido em KM3). Em seguida acontece a transformação desse modelo ECORE do AspectualACME, para uma representação textual de AspectualACME, verificado conforme sua BNF. Os passos de verificação com metamodelos das linguagens (passos 4 e 6) são necessários pois o modelo Ecore do sistema pode ter sido gerado a partir de especificações mal-formadas. Dessa forma, como resultado da Fase 1 temos um modelo AspectualACME bem-formado.

Dando seguimento ao processo principal de MARISA-MDD, o resultado obtido da transformação de AOV-graph para AspectualACME, a representação textual de AspectualACME, pode ser entrada do processo de transformação entre AspectualACME e aSideML, nesse caso, indicamos pela linha pontilhada na Figura 28, que o modelo ECORE referente a essa representação textual de AspectualACME é carregado, e transformado diretamente para modelo ECORE do aSideML, não passando novamente pelos passos referentes a verificação da representação textual (passos 9 a 12), já que essa representação gerada já passou por esses passos no processo de transformação AOV-graph para AspectualACME. No caso da representação textual ser uma representação elaborada pelo arquiteto, o processo segue normalmente na ordem dos passos enumerados da Fase 2.

Na Fase 2, o modelo textual de AspectualACME é transformado para o modelo ECORE do AspectualACME e esse é verificado conforme o metamodelo AspectualACME (definido em KM3). Assim, ATL transforma o modelo ECORE do AspectualACME para o modelo aSideML, verificado conforme o metamodelo aSideML (definido em KM3). Em seguida acontece a transformação desse modelo ECORE do aSideML, para uma representação textual de aSideML, verificada conforme sua BNF.

Adicionalmente, MARISA-MDD possui o processo de transformação inverso de transformação entre AOV-graph e AspectualACME. O processo também está organizado em duas fases: Elaboração e Verificação de Modelo (Fase 1) e Mapeamento e Verificação de Modelo (Fase 2). A entrada desse processo de transformação é uma representação textual de um modelo AspectualACME. Essa representação pode ser construída pelo arquiteto ou obtido a partir de um modelo AOV-graph, por meio do processo de transformação entre AOV-graph e AspectualACME da ferramenta MARISA-MDD. Nesse caso, o metamodelo e modelo dessa representação textual gerada por essa transformação são carregados, e é realizada uma transformação direta

para o modelo ECORE do AOV-graph, não necessitado dos passos referentes à verificação da representação textual. Caso seja alimentado de forma manual, o modelo AspectualACME é verificado se está de acordo com sua BNF, dessa forma temos modelos bem-formados usados nesse processo de transformação. De modo que, um modelo AspectualACME é a saída da Fase 1.

Na Fase 2, o modelo textual de AspectualACME é transformado para o modelo ECORE do AspectualACME, e esse é verificado conforme o metamodelo AspectualACME (definido em KM3). Assim, ATL atua na transformação entre o modelo ECORE do AspectualACME para o modelo AOV-graph, verificado conforme o metamodelo AOV-graph (definido em KM3). Em seguida acontece a transformação desse modelo ECORE do AOV-graph, para uma representação textual de AOV-graph, e se essa especificação está de acordo com a BNF. Além disso, a representação textual de AOV-graph resultante dessa transformação, pode ser entrada do processo de transformação entre AOV-graph para AspectualACME e aSideML, representado na Figura 28.

4.2. Regras de Transformação

Na Figura 29 estão representadas as áreas de atuação das tecnologias MDD envolvidas no processo total da transformação de AOV-graph, passando por AspectualACME, até aSideML executado no ambiente MaRiSA-MDD. A parte em azul identifica a área de atuação do mecanismo TCS: (i) transformar representações textuais de linguagens específicas de domínio em modelos do tipo ECORE, essa ação é realizada pelo *injector*; (ii) transformar modelos do tipo ECORE em representações textuais de linguagens específicas de domínio, essa ação é realizada pelo *extrator* e (iii) verificar sintaxes das representações textuais, conforme BNF. A parte cinza identifica a área de atuação do KM3 que é responsável por descrever os metamodelos. A parte em amarelo destaca a área de atuação da linguagem ATL, realizar transformações entre modelos do tipo ECORE, e verificar se a sintaxe dos modelos estão de acordo com o que foi definido nos metamodelos.

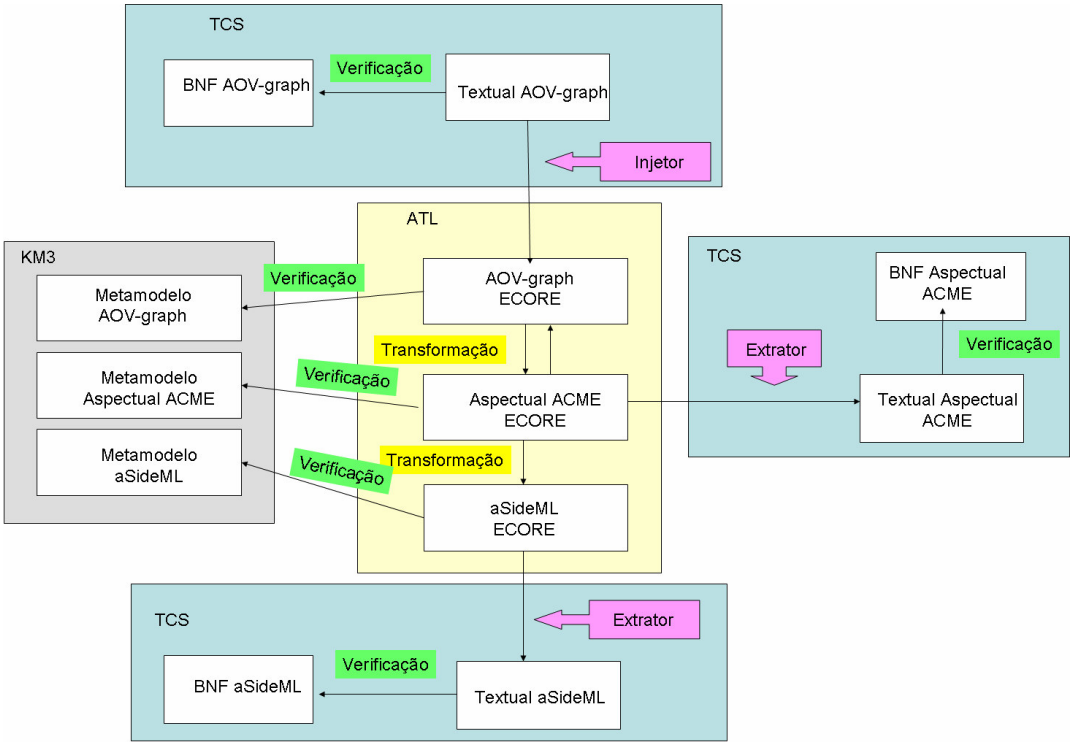


Figura 29: Representação do Processo de Tranformação de AOV-graph para aSideML

Para exemplificar os resultados do processo ilustrado na Figura 28, e 29. as Figuras 30 a 36 representam as regras de transformação que foram definidas para mapear modelos AOV-graph em modelos AspectualACME, e vice-versa, e de modelos AspectualACME para aSideML.

Na Figura 30 temos a regra *TaskAOVgraph2ComponentAspectualACME*, responsável por realizar o mapeamento de tarefas de AOV-graph para Componentes em AspectualACME. Se o componente é do tipo Tarefa em AOV-graph *(comp.oclIsTypeOf(AOVgraph!Task)*, e possui filhos (*comp.getAllChildren().notEmpty()),* então ele é transformado em componente de AspectualACME. Os atributos desse componente serão preenchidos pelas informações vindas dos atributos Tarefa de AOV-graph para Componentes em AspectualACME (linhas 6 a 42).

```
--@begin rule TaskAOVgraph2ComponentAspectualACME
--@comments Essa regra realiza verificação se o elemento Task tem filhos (se o método getAllChildren não é vazio), se ele tem filhos ele é um Component em AspectualACME
1 rule TaskAOVgraph2ComponentAspectualACME {
2       from
3               comp: AOVgraph!Component (comp.oclIsTypeOf(AOVgraph!Task) and comp.getAllChildren().notEmpty())
4       to
```

```
5                       element : AspectualACME!Component(
6                               name <- comp.name,
7                               properties <- Sequence {valorProperties,valorPropertiesTwo,valorPropertiesThree,valorPropertiesFour},
8                         representation <- valorRepresentations,
9                               elements <- comp.getAllChildren()
10                              ),
11                              valorProperties: AspectualACME!Properties (
12                                      key <-  'elementType',
13                                      value <- 'task'
14                              ),
15                              valorPropertiesTwo : AspectualACME!Properties (
16                                      key <-  'contribution_label',
17                                      value <- comp.contribution_label.toString()
18                              ),
19                              valorPropertiesThree : AspectualACME!Properties (
20                                      key <-  'topics',
21                                      value <- comp.topic
22                              ),
23                              valorPropertiesFour : AspectualACME!Properties (
24                                      key <-  'id',
25                                      value <- comp.id
26                              ),
27                              valorRepresentations: AspectualACME!Representation (
28                                                              system <- valSystem
29                              ),
30                                      valSystem: AspectualACME!System (
31                                              nameSystem <- if element.getText()->size()<1
32                                                      then
33                                                              'details'
34                                                      else
35                                                              ''
36                                                       endif,
37                                                      elements <- if element.getText()->size()<1
38                                                         then
39                                                         comp.getAllChildren()
40                                                                 else  OrderedSet{}
41                                                                         endif
42                                      )
43 }
```

Figura 30: Transformação ATL de Tarefas

Na Figura 31 é definida a regra *CrosscuttingRelAOVgraph2AspectualConnectorAspectualACME*, responsável pelo mapeamento do relacionamento transversal de AOV-graph para elementos do C*onector Aspectual* de AspectualACME. Se o relacionamento é do tipo transversal *(crosscutting.oclIsTypeOf(AOVgraph!CrosscuttingRel)*), então ele é transformado para Conector Aspectual em AspectualACME. Dessa forma, os seus atributos (linhas 5 a 14) vão ser preenchidos com informações provenientes do relacionamento transversal de AOV-graph. O atributo *links* de AspectualACME recebe o resultado de outras regras definidas para determinar se os elementos dos *Attachments* vão ser mapeados para *intertype declaration ou advices* em AOV-graph (linhas 6 a 10).

```
--@begin rule CrosscuttingRelAOVgraph2AspectualConnectorAspectualACME
--@comments Essa regra ATL transforma o crosscutting relationship de AOV-graph para Conector em AspectualACME
1rule CrosscuttingRelAOVgraph2AspectualConnectorAspectualACME{
2           from
3             crosscutting: AOVgraph!Relationship (crosscutting.oclIsTypeOf(AOVgraph!CrosscuttingRel))
4            to
5             conectorAspectual : AspectualACME!ConectorAspectual(
```

```
6               links <- Sequence {crosscutting.advice ->
7                       collect(ad|thisModule.AdviceAOVgraph2AttachmentsAspectualACME(ad)),crosscutting.intertypeDeclaration ->
8                       collect(intertype|thisModule.IntertypeDeclarationAOVgraph2AttachmentsAspectualACME(intertype))},
9               name <- crosscutting.source.name,
10              properties <- valorProperties
11          ),
12              valorProperties : AspectualACME!Properties(
13                                  key <- 'source',
14                                  value <- conectorAspectualACME.name
15          )
16 }
```

Figura 31: Transformação ATL de Relacionamento Transversal

A Figura 32 define a regra *ComponentsAspectualACME2TasksAOVgraph* que transforma os elementos Componentes de AspectualACME para tarefas em AOV-graph. É verificado se o componente possui o atributo *elemenType* de propriedades igual a Tarefa *(elems.returnElementTypePropertiesAACME()='task')* ,informação vinda do modelo AOV-graph da transformação de AOV-graph par AspectualACME. Dessa forma, os atributos de Componentes de AspectualACME vão preencher os atributos correspondentes a Tarefa em AOV-graph (linhas 6 a 12). Além disso, os atributos *component* são preenchidos pelo resultado de outras regras definidas para essa transformação (ver Figura 33).

```
1 rule ComponentsAspectualACME2TasksAOVgraph {
2           from
3                       elems: AspectualACME!Component (elems.returnElementTypePropertiesAACME()='task')
4           to
5
6                       compTask: AOVgraph!Task (
7                          name <- elems.name,
8                          component <- elems.representation ->
9                                   collect(representation|thisModule.ElementAspectualACMERepresentationTasksAOVgraph (representation)),
10                         component <- elems.getAllChildrenAACME(),
11                         component <- elems.elements ->collect(elems|thisModule.PortsAspectualACME2TasksAOVgraph(elems)))
12 }
```

Figura 32: Transformação de Componentes

A Figura 33 define a regra *ElementAspectualACMERepresentationTasksAOVgraph* e *ElementsAspectualACMESystem2TasksAOVgraph.* A primeira regra forma os elementos tarefas de AOV-graph que são componentes dentro de representações em AspectualACME (linhas 5 a 7), e preenche um dos atributos *component* da regra de Transformação de Tarefas. A segunda regra, preenche o atributo *component* da regra *ElementAspectualACMERepresentationTasksAOVgraph* de representação (linha 6 a 7).

```
1 lazy  rule ElementAspectualACMERepresentationTasksAOVgraph {
2           from
3                       representation : AspectualACME!Representation
4           to
```

```
5                       comp: AOVgraph!Task (
6                               component <- representation.system ->
7                                       collect(system|thisModule.ElementsAspectualACMESystem2TasksAOVgraph (system))
8                               )
9}

10 lazy rule ElementsAspectualACMESystem2TasksAOVgraph {
11                      from
12                              system : AspectualACME!System
13                      to
14                              comp: AOVgraph!Task (
15                                      component <- system.getAllChildrenAACMESystem()
16                              )
17 }
```

Figura 33: Transformação elementos da Representação e Sistema

A Figura 34 define a regra principal *SystemAspectualACME2ModelaSideML* que forma o modelo aSideML a partir do modelo AspectualACME. Essa regra recebe em cada um de seus elementos (linha 6 a 11) a coleção de resultados provenientes das outras regras. A seguir serão explicadas e mostradas três dessas regras que compõem os atributos *componentsAspectuals* (linha 6), *aspectualCollaboration* (linha 10) e *relCrosscutting* (linha 11) da regra principal *SystemAspectualACME2ModelaSideML*.

```
1 rule SystemAspectualACME2ModelaSideML {
2     from
3         system: AspectualACME!System
4     to
5         model: aSideML!Model (
6          componentsAspectuals <- system.compAspectuais -> collect(system|thisModule.ComponentACME2AspectaSideML(system)),
7          classesAspectuals <- system.compClass -> collect(system|thisModule.ComponentsACME2ClassaSideML(system)),
8          interfaceBase <- system.interBase -> collect(system|thisModule.PortsACME2OperationIBaseaSideML(system)),
9          interfaceCrosscutting <- system.interCrosscutting ->collect(system|thisModule.PortsACME2OperationICrosscuttingaSideML(system)),
10         aspectualCollaboration <- system.aspCollaboration ->
                   collect(system|thisModule.ConectorAspectual2AspectualCollaborationaSideML(system)),
11          relCrosscutting <- system.relCross -> collect (system|thisModule.ConectorAspectual2relCrosscuttingaSideML(system))      )
12 }
```

Figura 34: Transformação ATL de Sistemas em AspectualACME

A Figura 35 define a regra de transformação *ComponentsACME2AspectaSideML* que mapeia componentes de AspectualACME para aspecto em aSideML, e a coleção de seus resultados compõem o elemento *componentsAspectuals* da regra principal. Se o componente possui uma (ou mais de uma) porta que está ligada a uma *crosscutting role* de algum conector aspectual *(elems.verificarPortaCross (elems) = true)*, ele representa um aspecto, já que esse componente implementa uma característica transversal. Dessa forma, os atributos do aspecto serão preenchidos pelas informações provenientes dos Componentes de AspectualACME para Aspecto em aSideML (linhas 6 a 11).

```
1 lazy rule ComponentsACME2AspectaSideML {
2          from
3            elems:AspectualACME!Component(elems.verificarPortaCross(elems)=true)
4           -- verifica se o elemento está ligado a porta cross)
5          to
6                    aspect : aSideML!Aspect(
7                              nameCrosscuttingElement <- elems.name,
8                                    crosscuttingElement <- valorCross
9                    ),
10                             valorCross : aSideML!CrosscuttingElement (
11                                   nameCrosscuttingElement <- elems.name ) }
```

Figura 35: Transformação ATL de Componentes

A Figura 36 define duas regras de transformação *ConectorAspectual2AspectualCollaborationaSideML* e *ConectorAspectual2relCrosscuttingaSideML* que decidem sobre o mapeamento dos conectores aspectuais. Na primeira regra, se o conector associa um advice a um elemento base (*conectorAspectual.verificarAdvice (conectorAspectual) = true*), ele é mapeado para uma colaboração aspectual. Assim, os elementos originados dos conectores aspectuais de AspectualACME são mapeados para os elementos que formam a colaboração aspectual de aSideML (linhas 7 a 11), e a coleção dos resultados dessa regra formam o elemento *aspectualCollaboration* da regra principal. A segunda regra é empregada quando o conector associa um *intertype declaration* a um elemento base (*conectorAspectual.verificarIntertype (conectorAspectual) = true*). Nessa situação, os elementos do conector aspectual são mapeados para os elementos do relacionamento crosscutting em aSideML (linhas 20 a 38), e a coleção dos resultados dessa regra compõem o elemento *relCrosscutting* da regra principal. Como exemplo, temos o elemento *templateMatch* que contém as informações relacionadas ao relacionamento crosscutting, uma dessas informações é o *templateParameter* (linha 28 a 30), que é formado pelo elemento base e o elemento aspectual envolvidos no relacionamento crosscutting e o joinpoint (linhas 35 a 38).

```
1 lazy rule ConectorAspectual2AspectualCollaborationaSideML {
2          -- origem=advice => mapeamento do Conector Aspectual para Colaboração Aspectual
3          from
4                        conectorAspectual:  AspectualACME!ConectorAspectual  (conectorAspectual.verificarAdvice  (conectorAspectual)  =
true)
5          to
6             --crosscuttingElemenbt incluem os elementos aspectuais afetados
7                        crosscuttingElement : aSideML!CrosscuttingElement(
8                                    name <- conectorAspectual.name                          ),
9            --baseElement operações base afetadas
10                       baseElement : aSideML!BaseElement (
11                                   name <- conectorAspectual.ports.name )
12 }
13
14 lazy rule ConectorAspectual2relCrosscuttingaSideML {
15          -- origem=intertypeDeclaraction=> mapeamento do Conector Aspectual para IntertypeDeclaration
16          from
```

```
17                  conectorAspectual:  AspectualACME!ConectorAspectual  (conectorAspectual.verificarIntertype  (conectorAspectual)  =
true)
18          to
19              --crosscuttingElement incluem os elementos aspectuais afetados
20                  crosscuttingElement : aSideML!CrosscuttingElement(
21                              name <- conectorAspectual.name                          ),
22              --baseElement operações base afetadas
23                  baseElement : aSideML!BaseElement (
24                              name <- conectorAspectual.ports.name),
25          templateMatch : aSideML!TemplateMatch(
26                          templateParameter <- temP,
27                          joinPoint <- jPoint          ),
28                  temP: aSideML!TemplateParameter (
29                                  cElement <- crossElement,
30                                  bElement <- baseElem),
31                  crossElement : aSideML!CrosscuttingElement(
32                                  name <- conectorAspectual.name                          ),
33                  baseElem : aSideML!BaseElement (
34                              name <- conectorAspectual.ports.name),
35                  jPoint: aSideML!JoinPoint (
36                          baseElement <- baseEl          ),
37                  baseEl: aSideML!BaseElement(
38                              name <- conectorAspectual.ports.name)  }
```

Figura 36: Transformação ATL do Conector Aspectual

4.3. Classificação das Transformações

No que se refere à Classificação de Transformações entre modelos no contexto do MDD, no trabalho de (CZARNECKI; HELSEN, 2006) é realizada uma categorização de abordagens que realizam transformações entre modelos para capturar características comuns às transformações MDD. Identificamos que nossa estratégia está relacionada com as seguintes categorias: **(i) Organização das Regras**, essa categoria refere-se a relações entre regras de transformação. Existem três relações que são consideradas: (a) mecanismos de modularidade, mostram as regras organizadas em módulos. Além disso, um módulo pode importar outros módulos; (b) mecanismos de reuso, oferecem caminhos para definir regras baseadas em uma mais regras, como exemplo, temos a herança entre regras; e (c) estrutura organizacional, as regras devem ser organizadas de acordo com a estrutura da linguagem de origem, de modo que as regras estão em conformidade a estrutura da linguagem de origem; ou linguagem de destino, onde temos uma regra para cada tipo de elemento e as regras estão de acordo com o metamodelo de destino; ou elas podem ser independentes da organização da estrutura; **(ii) Programação de Regras,** essa categoria inclui mecanismos pela ordem na qual as regras são aplicadas. Esses mecanismos variam de acordo com a forma, ou seja, o caminho em que a ordem é expressa. A forma pode ser implícita ou explícita. Na forma implícita temos a relação das regras com outras regras. Na forma explícita, temos o uso de construtores para controlar a ordem de aplicação das regras. Essa forma pode

ser externa ou interna. A interna controla o fluxo da estrutura dentro das regras e invocação explícitas. Na' externa a lógica de programação é usada de forma separada das regras de transformação. E se a interação das regras é recursiva, *looping*, ou *fixpoint***; (iii) Direcionalidade,** as transformações podem ser unidirecionais ou multidirecionais. As transformações unidirecionais só podem ser executadas em uma direção, na qual um modelo de destino é executado (ou atualizado) baseado em um modelo de origem. As transformações multidirecionais podem ser executadas em várias direções. Além disso, podem ser alcançadas usando as regras multidirecionais, ou através da definição de diversas regras unidirecionais complementares, uma para cada direção, e **(iv) Rastreio**, categoria relacionada às informações e ligações de rastreabilidade na transformação entre modelos. As ligações para a rastreabilidade podem ser controladas de forma manual ou automática. No caso da automática, por exemplo, informações das regras podem ser controladas. Há a escolha da localização onde as ligações de rastreabilidade serão armazenadas (em modelo, se o de origem,ou destino; ou separada).

Inserindo nossa abordagem nesse contexto, temos que, a transformação modelo para modelo (*model-to-model),* que fazemos pode ser classificada como: (i) em relação a Organização das Regras, conforme ilustra a Figura 37, a nossa estratégia é: (a) utiliza mecanismos modularizados, ou seja, as regras estão organizadas em módulos; (b) orientada a destino *(target)*, pois há uma regra para cada tipo de elemento e as regras estão de acordo com o metamodelo de destino; (c) utiliza mecanismos de herança para reuso de regras; (ii) em relação a Programação de Regras, de acordo com a representação da Figura 38, a nossa proposta possui: (a) forma explícita, ou seja, temos o uso de construtores para controle da ordem de aplicação das regras, e forma implícita, onde temos a relação das regras com outras regras, as regras mostradas anteriormente na Seção 4.2 são exemplos dessa forma, e (b) a interação das regras é realizada pelo uso da recursão; (iii) em relação a Direcionalidade, de acordo com a Figura 39, a nossa estratégia é unidirecional, apesar de realizar transformação entre AOV-graph e AspectualACME, e realizar o sentido inverso, ela utiliza regras em um sentido só para realizar ambas as transformações; (iv) em relação ao Rastreio, conforme a Figura 40, nossa abordagem é: (a) ligações de rastreabilidade automáticas, de modo que as regras são controladas de forma automática e (b) e as ligações serão armazenadas nos modelos de origem e destino.

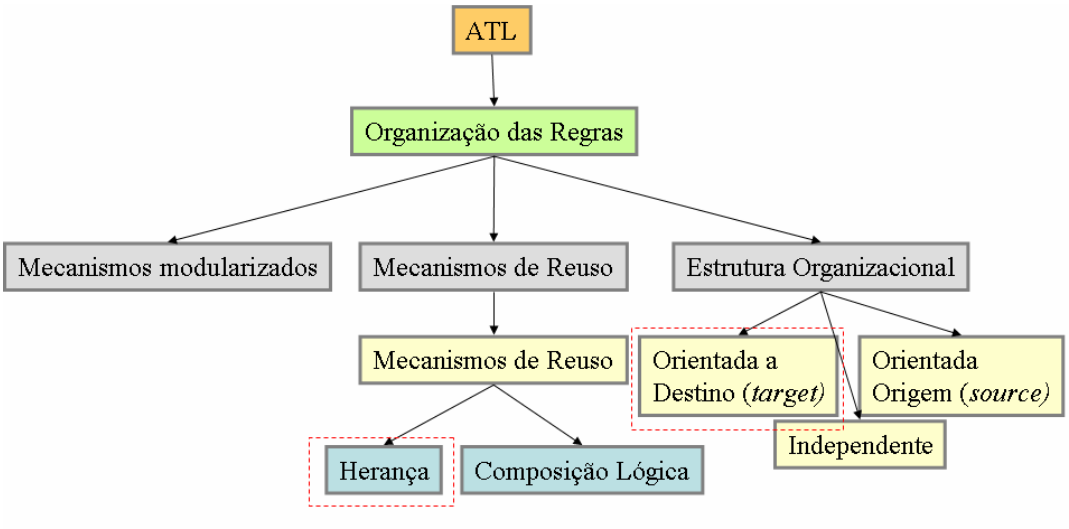


Figura 37: Classificação de ATL de acordo com a Categoria Organização das Regras

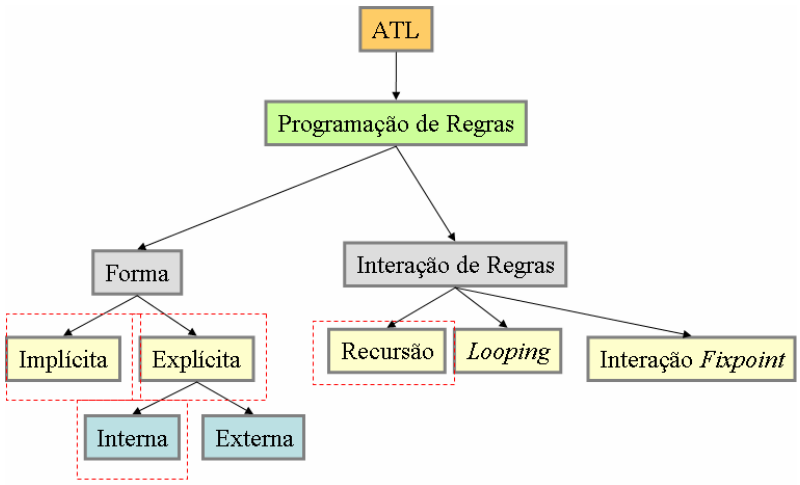


Figura 38: Classificação de ATL de acordo com a Categoria Programação das Regras

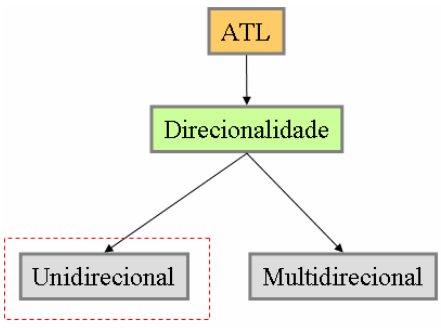


Figura 39: Classificação de ATL de acordo com a Categoria de Direcionalidade

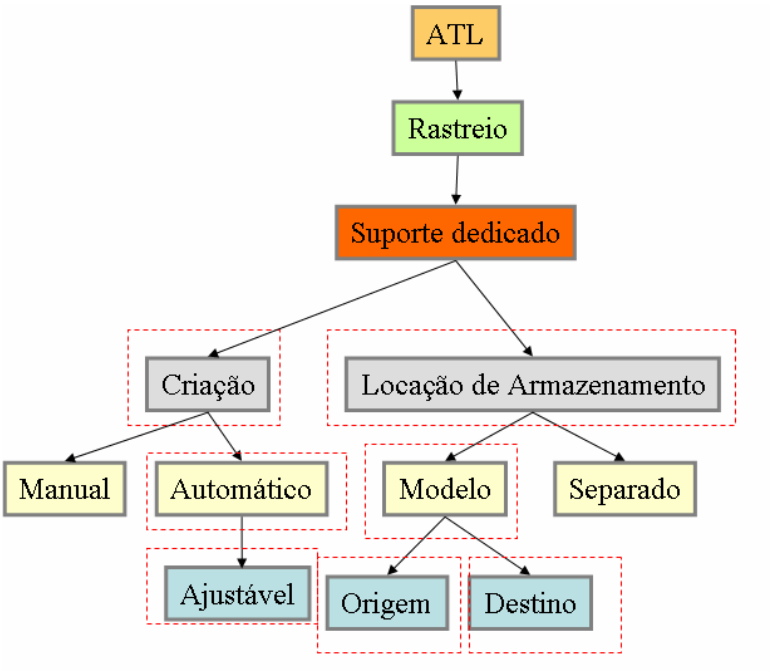


Figura 40: Classificação de ATL de acordo com a Categoria de Rastreio

4.4. Estudos de Caso

Para validação das transformações e exemplos, aplicamos além do HeatlhWatcher, que está expresso ao longo do nosso trabalho, utilizando nessa seção o *Mobile Media* (FIGUEIREDO ET AL., 2008).

4.4.1. Móbile Media

O Mobile Media é um sistema para manipulação de fotos, música, vídeos em dispositivos móveis.

A Figura 41 ilustra outro exemplo da descrição textual em AOV-graph (a) e a descrição textual AspectualACME (b) resultante do processo de transformação baseada em modelos. Na Figura 41 (a) é descrito o modelo de metas *MobileMedia_ProductLine.* A hierarquia de tarefas é mapeada para o componente *External_Software_Communicator* de AspectualACME e para as portas *Communication_with_MusicPlayer*, *Communication_with_PhotoViewer*, *Communication_with_SMSSoftware, Communication_with_VideoPlayer,* e *Communication_with_EmailSoftware.*. Cada advice do relacionamento transversal dá origem a um conector aspectual de AspectualACME. Assim, o relacionamento

transversal de origem *External_Software_Communicator,* e o tipo da cláusula *glue* é determinado pelo tipo do advice (no caso, around), cada relação advice→ pointcut é mapeada para um associação de configuração de elementos. Os elementos do *pointcut* em AOV-graph são mapeados para as conectadas a *base role* emAspectualACME, e os elementos do *advice* são mapeados para as *portas* conectadas a *crosscutting role.* Dessa forma, as duas primeiras configurações do exemplo em AspectualACME associando *External_software_communicator. Communication_with_MusicPlayer to External_software_communicator_Conec.source e External_software_communicator_Conec.sink to Media_manager.Play_media.*

```
(…)
goal_model MobileMedia_ProductLine (GM1) {
  task External software communicator (T1.29; ) {
  task Communication with SMSsoftware (T1.30; ) {}
  task Communication with Emailsoftware (T1.31; ) {}
  task Communication with PhotoViewer (T1.32; ) {}
  task Communication with MusicPlayer (T1.33; ) {}
  task Communication with VideoPlayer (T1.34; ) {}
   crosscutting {
     source = (External software communicator; T1.29)
      pointcut (SMS; PC1.30.1): include(.*Sms; task;
name)
      pointcut (Email; PC1.30.2): include(.*email; task;
name)
      pointcut (PhotoViewer; PC1.30.3): include(Display
[media]; T1.4.10) and include(Display [photo] of incoming
caller; T1.4.17)
      pointcut (MusicPlayer; PC1.30.4): include(Play
[media]; T1.4.15)
       advice (around): PC1.30.4 {
        task_ref Communication with MusicPlayer (T1.33; )
             task_ref Communication with VideoPlayer
(T1.34;)}
            advice (around): PC1.30.3 {
              task_ref Communication with PhotoViewer
(T1.32; )}
            advice (around): PC1.30.1 {
              task_ref Communication with SMSsoftware
(T1.30; )}
            advice (around): PC1.30.2 {
              task_ref Communication with Emailsoftware
(T1.31; )} } }
```

(a) AOV-graph

```
System MobileMedia = {
(…)
Component External_software_communicator = {
  Properties {elementType = task; contribution = [null,and];
correlations = [null,null]; topics = null};
       Port self = { (…)};};
       Port Communication_with_MusicPlayer = {
          Properties {elementType = task; contribution
=[External_software_communicator,and]; correlations =
[null,null]; topics = null};};
        Port Communication_with_PhotoViewer = { (…)};
        Port Communication_with_SMSsoftware = { (…)};
        Port Communication_with_VideoPlayer = { (…)};
        Port Communication_with_Emailsoftware = { (…)};
};
(…)
  Connector External_software_communicator_Conec1 = {
    baseRole sink;
    crosscuttingRole source;
    glue source around sink;
    Properties = {comesFrom = advice ; crossRelSource =
External_software_communicator};
};
Attachments {
   External_software_communicator.
Communication_with_MusicPlayer to
External_software_communicator_Conec1.source;
            External_software_communicator_Conec1.sink
to Media_manager.Play_media;};
   }
}
```

(b) AspectualACME

Figura 41: Exemplo de transformação entre AOV-graph e AspectualACME do Sistema MobileMedia

A Figura 42 ilustra outro exemplo de descrição textual em AspectualACME (a) e a descrição textual AOV-graph (b) . Essa descrição textual AspectualACME também foi gerada pela transformação AOV-graph para Aspectual realizada por MaRiSA-MDD. A Figura 42 (a) é descrito o componente *External_software_communicator.* As portas *Communication_with_MusicPlayer*, *Communication_with_PhotoViewer*, *Communication_with_SMSSoftware, Communication_with_VideoPlayer,* e

Communication_with_EmailSoftware desse componente são mapeadas para tarefas em AOV-graph. O conector aspectual *External_software_communicator_Conec1* e seus *attachments* geram o relacionamento transversal cuja a origem é *External_software_communicator* com advice em AOV-graph, o advice novamente foi determinado, devido a especificação em propriedades (*comesFrom:Advice)* do conector aspectual em AspectualACME. A porta relacionada à *crosscutting role* (porta *Communication_with_MusicPlayer)* é mapeada para o corpo do advice (dependendo do tipo da cláusula glue), e a porta ligada a *base role* (porta *Play_media)* é mapeada para *pointcuts.*

```
System MobileMedia = {
(…)
Component External_software_communicator = {
  Properties {elementType = task; contribution
=[null,and]; correlations = [null,null]; topics = null};
       Port self = { (…)};};
       Port Communication_with_MusicPlayer = {
          Properties {elementType = task; contribution
=[External_software_communicator,and]; correlations =
[null,null]; topics = null};};
        Port Communication_with_PhotoViewer = { (…)};
        Port Communication_with_SMSsoftware = { (…)};
        Port Communication_with_VideoPlayer = { (…)};
        Port Communication_with_Emailsoftware = { (…)};
};
(…)
  Connector External_software_communicator_Conec1 =
{
    baseRole sink;
    crosscuttingRole source;
    glue source around sink;
    Properties = {comesFrom = advice ; crossRelSource
=  External_software_communicator};
};
Attachments {
   External_software_communicator.
Communication_with_MusicPlayer to
External_software_communicator_Conec1.source;
            External_software_communicator_Conec1.sink
to Media_manager.Play_media;};
   }
}
```

(a) AspectualACME

```
(…)
goal_model MobileMedia_ProductLine (GM1) {
  task External software communicator (T1.29; ) {
  task Communication with SMSsoftware (T1.30; ) {}
  task Communication with Emailsoftware (T1.31; ) {}
  task Communication with PhotoViewer (T1.32; ) {}
  task Communication with MusicPlayer (T1.33; ) {}
  task Communication with VideoPlayer (T1.34; ) {}
   crosscutting {
     source = (External software communicator; T1.29)
      pointcut (SMS; PC1.30.1): include(.*Sms; task;
name)
      pointcut (Email; PC1.30.2): include(.*email; task;
name)
      pointcut (PhotoViewer; PC1.30.3): include(Display
[media]; T1.4.10) and include(Display [(photo]) of
incoming caller; T1.4.17)
      pointcut (MusicPlayer; PC1.30.4): include(Play
[media]; T1.4.15)
       advice (around): PC1.30.4 {
        task_ref Communication with MusicPlayer (T1.33; )
            task_ref Communication with VideoPlayer
(T1.34;)}
             advice (around): PC1.30.3 {
              task_ref Communication with PhotoViewer
(T1.32; )}
            advice (around): PC1.30.1 {
              task_ref Communication with SMSsoftware
(T1.30; )}
            advice (around): PC1.30.2 {
              task_ref Communication with Emailsoftware
(T1.31; )} } }
```

(b) AOV-graph

Figura 42: Exemplo de transformação entre AspectualACME e AOV-graph do Mobile Media

A Figura 43 ilustra outro exemplo de descrição textual em AspectualACME (a) e a descrição textual ASideML (b) resultante do processo de transformação baseada em modelos. A descrição textual em AspectualACME foi gerada pela transformação AOV-graph para AspectualACME realizada por MaRiSA-MDD. A Figura 43 (a) ilustra o componente *External_Software_Communicator* que é conectado via um conector aspectual ao componente *Media_manager* via o conector

External_Software_Communicator_Conec1. O componente *Media_manager* é mapeado para uma classe pois não tem relação com nenhuma *crosscutting role*, mas apenas com a base role *sink*. As tags das classes são mapeadas a partir das propriedades do componente do qual elas foram derivadas. A porta *Communication_with_MusicPlayer* do componente *External_Software_Communicator* está conectada a uma *crosscutting role* do conector *External_Software_Communicator_Conec1*, então esse componente é mapeado para um aspecto.

No caso do aspecto *External_Software_Communicator,* um parâmetro de *template* é criado, o tp1. Esse parâmetro possui a interface *IExternal_Software_Communicator,* e operação *Communication_with_MusicPlayer,* que é a única porta do componente I*External_Software_Communicator* que se liga a alguma *crosscutting role* de um conector aspectual. A operação do parâmetro já é exibida com seu adorno, extraído da cláusula *glue* de *External_Software_Communicator_Conec1*, ao qual a porta está conectada.

No componente *External_Software_Communicator*, cinco de suas portas são mapeadas para operações do aspecto (*Communication_with_MusicPlayer*, *Communication_with_PhotoViewer*, *Communication_with_SMSSoftware, Communication_with_VideoPlayer,* e *Communication_with_EmailSoftware).* A porta *Communication_with_MusicPlayer* relaciona-se com o conector aspectual *External_Software_Communicator_Conec1*. Porém, como o conector não possui a propriedade *comesFrom: intertype_declaration* nem a porta possui a propriedade *type: required*, ela é mapeada para uma característica transversal comportamental e disposta no compartimento *Refinement* da interface. Para cada característica transversal comportamental da interface, um sub-nível em relação ao nível *Refinement* é criado, nesse caso, o *refinement1.*

O conector *External_Software_Communicator_Conec1* possui a propriedade *comesFrom:advice* e, portanto, é mapeado para uma colaboração aspectual. Para cada conector, um compartimento *RepresentationCollaboration* é criado e, dentro dele, para cada par de *attachment*, um sub-nível é criado descrevendo a colaboração aspectual em si. Ainda no compartimento *RepresentationCollaboration*, as tags que foram mapeadas do conector são exibidas. No caso de *External_Software_Communicator_Conec1*, apenas um par de *attachment* é definido. No compartimento *AspectualCollaboration External_Software_Communicator_Conec1* a colaboração aspectual mapeada é exibida. *AspectOperation* é a característica mapeada a partir da porta que se liga a *crosscutting*

role do conector, no caso *Communication_with_MusicPlayer*. O elemento base é que foi mapeado a partir do componente cuja porta liga-se à base role do conector, que no exemplo é *Media_manager*. A operação que é mapeada a partir da porta que se relaciona com a base role, *Play_media,* não possui a propriedade *type* e, como seu valor default é *provided,* ela é mapeada para a operação base da colaboração aspectual e não para o objeto chamador da operação base (*Sender*). O tipo de crosscutting (*Ornament*) da colaboração aspectual é derivado da cláusula *glue* do conector *External_Software_Communicator_Conec1*, e o tipo de extensão é *Refinement* porque a *feature Communication_with_MusicPlayer* é do tipo *Refinement*.

```
System MobileMedia = {
(…)
Component External_software_communicator = {
  Properties {elementType = task; contribution
=[null,and]; correlations = [null,null]; topics = null};
       Port self = { (…)};};
       Port Communication_with_MusicPlayer = {
          Properties {elementType = task; contribution
=[External_software_communicator,and]; correlations =
[null,null]; topics = null};};
        Port Communication_with_PhotoViewer = { (…)};
        Port Communication_with_SMSsoftware = { (…)};
        Port Communication_with_VideoPlayer = { (…)};
        Port Communication_with_Emailsoftware = { (…)};
 };
Connector External_software_communicator_Conec1 ={
 baseRole sink;
 crosscuttingRole source;
 glue source around sink;
    Properties = { comesFrom = advice ; crossRelSource
= External_software_communicator};
};
Attachments {
  External_software_communicator.
  Communication_with_MusicPlayer to
External_software_communicator_Conec1.source;
   External_software_communicator_Conec1.sink to
Media_manager.Play_media;};
}
```

```
Model aSideML MobileMedia = {
(...)
   Aspect  External_software_communicator  {
      Interface: IExternal_software_communicator;
      Tags: elementType = task; contribution =[null,and];
correlations = [null,null]; topics = null;
      Operations: op1;
      TemplateParameter : tp1;
   };
   Operation op1 {
       Name: Communication_with_MusicPlayer;
       Tags: elementType = task; contribution
=[External_software_communicator,and]; correlations =
[null,null]; topics = null;
       Name: Communication_with_PhotoViewer;
       Tags: (…);
      Name: Communication_with_VideoPlayer;
      Tags: (…);
      Name: Communication_with_Emailsoftware;
      Tags: (…);
 TemplateParameter tp1 {
     Parameter: parameter1;
 };
  Parameter1 parameter1 {
    Interface: IExternal_software_communicator;
    Operations: Communication_with_MusicPlayer;;
  };
CrossccuttingInterface  IExternal_software_communicator {
    Addition: addition1;
    Refinements: refinement1;
    Redefinitions: null;
    Uses: null;
  };
Refinement refinement1 = {
    Interface: I External_software_communicator;
    Operation: _ Communication_with_MusicPlayer _;
    Tags: elementType = task; contribution
=[(External_software_communicator,and]); correlations =
[(null,null]; topics = null]);};
Addition addition1 = {
   Interface: I External_software_communicator;
   Operation: Detect_communication_exception;}
(…)
   };
AspectualCollaboration
External_software_communicator_Conec1 {
      Aspect: External_software_communicator;
      AspectOperation: Communication_with_MusicPlayer;
      BaseElement: Media_manager;
      BaseOperation: Play_media;
      Sender: ;
      Ornament: around;
      Type: Refinement;
```

	};
	RepresentationCollaboration
	External_software_communicator {
	AspectualCollaboration:
	External_software_communicator_Conec1;
	Tags: comesFrom = advice, crossRelSource =
	External_software_communicator;
	};
	};

(a) AspectualACME	**(b) aSideML**

Figura 43: Exemplo de transformação de AspectualACME para aSideML do Mobile Media

4.5. Análise dos modelos obtidos de MARISA-MDD

Nesta seção, relatamos a experiência envolvida na transformação entre modelo de requisitos (AOV-graph), e modelo de arquitetura (AspectualACME), e vice-versa, e entre modelo de arquitetura (AspectualACME), e modelo de projeto detalhado (aSideML), fazendo uma análise qualitativa que considera os seguintes parâmetros: completude, rastreabilidade e sinergia.

4.5.1. Avaliação do modelo de arquitetura gerada a partir de AOV-graph

(1) Completude: há elementos em AspectualACME que não tenham sido modelados? Os principais elementos estruturais de AspectualACME, que são *components, ports, connectors, attachments, bindings, representations* e *properties* são utilizados no mapeamento a partir dos elementos de AOV-graph. Além disso, semanticamente, é possível obter com resultado uma estruturação inicial da arquitetura que é essencial e valiosa por prover as informações levantadas nas atividades de requisitos. Todavia, as informações arquiteturais tais como plataforma a ser utilizada, estilo arquitetural mais adequado, protocolos de conexão, dentre outras, são informações que não estão presentes na arquitetura, pois os requisitos não as descrevem.

(2) Rastreabilidade: a arquitetura descrita em AspectualACME contem elementos que representem todos os requisitos modelados e suas características?; *Rationale* – é possível identificar quais elementos nos requisitos deram origem aos elementos arquiteturais? A rastreabilidade é possível, à medida que, os elementos estruturais de AOV-graph são mapeados para algum elemento estrutural de AspectualACME. Dessa forma, há continuidade das informações definidas em AOV-graph, pois elas continuam existindo em AspectualACME. É possível, por exemplo, identificar quais componentes estão correlacionados a uma certa softmeta a partir da arquitetura gerada, bem como

saber se esta correlação é positiva ou negativa ou saber se determinados componentes e portas foram gerados de softmetas, metas ou tarefas.

4.5.2. Avaliação do modelo de requisitos gerado a partir de AspectualACME

Chamemos de Situação1: Descrição AspectualACME resultante do mapeamento AOV-graph, quando há informações do AOVgraph nas propriedades (*properties*) de AspectualACME; e Situação 2: Descrição AspectualACME não foi gerada a partir de AOV-graph, quando as informações de propriedades (*properties)* de AspectualACME não estão disponíveis.

(1) Completude – há elementos em AOV-graph que não tenham sido modelados? Na Situação1 todos os elementos estruturais de AOV-graph são gerados. Na Situação2 correlações não podem ser geradas, pois não temos em AspectualACME , nem *intertype declarations*, pois não temos as informações relacionadas a esses elementos que são específicas de requisitos.

(2) Rastreabilidade: o modelo de requisitos em AOV-graph possui elementos que representem todos os elementos arquiteturais modelados e suas características? É possível identificar quais elementos na arquitetura deram origem aos elementos no modelo AOV-graph? Um novo atributo, denominado *property,* foi adicionado aos elementos de AOV-graph (softmeta, meta, tarefa, correlação, contribuição, crosscutting). Com isto, na Situação1 e Situação2 todas as informações específicas da arquitetura são mapeadas para propriedades em AOV-graph, tal como fizemos de AOV-graph para AspectualACME. Dentre tais informações é possível saber quais os componentes ou portas geram uma determinada meta, ou se uma certa contribuição foi gerada por um conector não-aspectual, por exemplo, essas informações são provenientes dos requisitos, da transformação AOV-graph para AspectualACME, e armazenadas nos atributos, *elementType, contributions, correlations, topics*, pertencentes as Propriedades de AspectualACME.

4.5.3. Avaliação da sinergia entre modelo de requisitos gerado a partir de AspectualACME, e vice-versa

Quais os benefícios da integração para os engenheiros de requisitos e arquitetos de software? Com a transformação entre modelos de AOV-graph para AspectualACME, conseguimos obter, de forma automática, versões iniciais da arquitetura e do modelo de requisitos, bem como propagar mudanças realizadas em ambos os modelos. Além disso, os arquitetos de software podem mais facilmente recorrer às informações de requisitos,

assim como os engenheiros de requisitos também podem recuperar mais agilmente as informações arquiteturais. Ressaltamos que embora informações extras estejam sendo geradas como elementos da própria linguagem de requisitos e de arquitetura elas devem ser geradas e mantidas por uma ferramenta e não pelo engenheiro de requisitos e arquiteto. Assim, esta transformação gera mais do que modelos de requisitos e arquiteturas, ele gera relacionamentos de associações entre os artefatos gerados, possibilitando uma ação com atuação coordenada (sinérgica) nestes modelos.

4.5.4. Avaliação do modelo de projeto detalhado gerado a partir de AspectualACME

(1) Completude: há elementos em AspectualACME que não tenham sido modelados? AspectualACME e aSideML têm características distintas relacionadas a forma de modelar componentes e aspectos, uma vez que AspectualACME não tem representações diferentes para componentes e aspectos. No entanto, temos duas situações: (a) Descrição AspectualACME resultante do mapeamento AOV-graph. Nesse caso, informações sobre *intertype declaration* estão representadas através de propriedades em AspectualACME e são mapeadas para aSideML; ou (b) Descrição AspectualACME não foi gerada a partir de AOV-graph. Nesse caso, não há informação sobre *intertype declaration*, uma vez que originalmente AspectualACME não representa tal informação. Nesse caso, não há como gerar um modelo aSideML com *intertype declaration*.

(2) Rastreabilidade: é possível identificar quais elementos em AspectualACME deram origem aos elementos no modelo aSideML? A rastreabilidade é possível pois: (a) todos os elementos de AspectualACME tem correspondência em aSideML, (b) para propagar as informações provenientes da descrição dos requisitos, as propriedades dos elementos da descrição arquitetural são transformadas para tags em aSideML.

(3) Sinergia: Quais os benefícios da integração para os arquitetos e projetistas de software? Com a integração entre as linguagens AspectualACME e aSideML conseguimos prover, de forma automática, versões iniciais da projeto detalhado, mantendo as informações arquiteturais e até as informações provenientes dos requisitos (quando a versão AspectualACME é gerada a partir de AOV-graph). Além disso, consegue-se propagar mudanças realizadas no modelo original. Com o mapeamento automático pode-se gerar, mais agilmente, modelo do projeto detalhado a partir das informações arquiteturais. Como informações extras são colocadas em tags, o

mapeamento produz como resultado mais do que um modelo de projeto detalhado; gera associações entre estes artefatos que possibilitam uma ação coordenada (sinérgica) entre estes modelos.

5. Trabalhos Relacionados

Essa seção apresenta um resumo das principais pesquisas realizadas na área de transformações no contexto de DSOA, que adotam explicitamente uma abordagem orientada a modelos, assim como trabalhos relacionados a transformações entre fases iniciais do processo de desenvolvimento de software, que não adotam explicitamente essa abordagem.

5.1. Relação de Requisitos e Arquitetura e framework de rastreabilidade

Este trabalho (CHITCHYAN ET AL., 2005) comenta sobre a distância existente na relação de requisitos orientados a aspectos e arquitetura orientada a aspectos, utilizando as abordagens AORE (*Aspect-Oriented Requirements Engineering),* como representante da atividade de requisitos que envolve identificação, representação, e tratamento de todos os tipos de requisitos (funcional, não funcional, etc), e AOAD (*Aspect-Oriented Architecture Design),* que representa a fase arquitetural, e oferece técnicas relacionadas ao desenvolvimento de software onde todos os requisitos são derivados de AORE.

Adicionalmente, para diminuir tal distância, o trabalho propõe um framework de rastreabilidade para trabalhar em conjunto com ferramentas que dêem suporte ao desenvolvimento orientado a modelos.

Esta abordagem considera que os aspectos podem ser mapeados de requisitos para arquitetura com os seguintes passos:

(i) Componente Arquitetural: um módulo arquitetural localizado, ou um elemento de um módulo.

(ii) Aspecto Arquitetural: um módulo arquitetural que tem influência sobre um número de outros módulos.

(iii) Decisão Arquitetural: uma decisão localizada para uma arquitetura em particular.

(iv) Decisão Arquitetural Aspectual: uma decisão influenciada por características transversais.

(v) Decisão Não Arquitetural: uma decisão relacionada à revisão de processos de negócio.

Para mostrar a ligação existente entre AORE e AOAD e entre os artefatos produzidos nas fases de Engenharia de Requisitos e Arquitetura, foram utilizados respectivamente, a *Requirements Description Language* (RDL), em termos de estratégia para expressar requisitos, e DAOP-ADL, uma linguagem de descrição arquitetural. Ambas são descritas utilizando a linguagem XML (Figura 47).

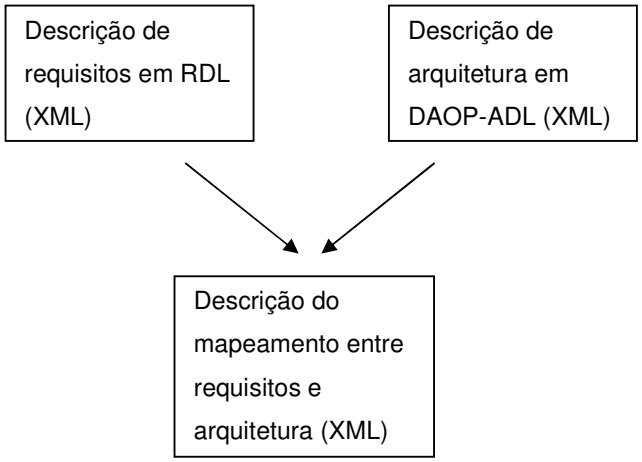


Figura 44: Representação do Framework

A proposta do Framework inclui a construção e extensão de diferentes representaçãoes (XML) para:

- Descrição de requisitos, arquitetura, projeto, manutenção de artefatos e implementação (RDL, DAOP-ADL)
- Mapeamentos e decisões
- Avaliação de modelos

Na tabela abaixo, está um resumo dessa proposta de mapeamento de requisitos para arquitetura.

Tabela 4: Resumo do mapeamento de requisitos para arquitetura

Requisitos	Arquitetura
1. Característica	1. Elemento Arquitetural (EA)
1.1. Característica Aspectual	1.1. EA Aspectual
1.2. Característica Não-Aspectual	1.2. EA Não Aspectual
2. Pontos de Visão	2. Decisão Arquitetural (DA)

3. Requisitos	2.1. DA Aspectual 2.2. DA Não Aspectual
Peso: O "peso" relacionado e alocado por um requisito pelo *stakeholder*	Nível de Satisfação 1. É satisfeito (nível arquitetural) 2. É parcialmente satisfeito 3. Não é satisfeito
Influência: a parte de outros elementos influenciadas (positivamente ou negativamente) pelo elemento em questão.	Conseqüência da Influência 1. É propagado para 2. add novo requisito 3. add nova decisão 4. depende de 5. influências 6. implicações
Requisitos emergentes: Requisitos identificado na fase de arquitetura , incluindo *links* para decisões que motivam essa considerações/inclusões	*Links* para elementos emergentes *Links* para decisões em outros estágios de desenvolvimento que serão propagados para arquitetura,e para as modificações nos requisitos que são motivadas por decisões do arquitetura de software
Elementos de Composição: Operações, ações	Elementos de Composição: Pontos de junção (*Joinpoints)* ou decisões Interfaces EA ou decisões

5.2. Um Framework Dirigido a Modelos Orientados a Aspectos

Este trabalho apresenta o AOMDF *(Aspect-Oriented Model Driven Framework)* (SIMMONDS, ET AL., 2005), que aplica a filosofia MDD no desenvolvimento de aplicações orientada a aspectos (AOSD), focando na fase de projeto detalhado. Esse framework propõe a separação vertical e horizontal de características e ilustra como técnicas baseadas em aspectos facilitam a separação de características e a transformações entre modelos da fase de projeto, representados por diagramas de seqüência. No nível abstrato PIM esses diagramas de seqüência são independentes de plataforma. Dessa forma, através do mapeamento para o nível PSM, os diagramas de seqüência gerados vem identificados com detalhes de plataforma a ser utilizada, como por exemplo, CORBA, que é o exemplo usado no trabalho.

O AOMDF dá suporte à separação de características horizontais fornecendo mecanismos para encapsular características transversais usando aspectos. Na modelagem da proposta do AOMDF, características transversais são modeladas como aspectos e compostos com o modelo de projeto que descreve as funcionalidades para formar aplicações completas.

A separação vertical de características é apoiada por técnicas MDA para transformar modelos de um nível de abstração para outro, de PIM para PSM. Os modelos são transformados usando mapeamentos definidos separadamente para um modelo inicial e outro para aspectos, utilizando como base QVT.

O foco inicial do AOMDF é a transformação de modelos orientados a aspectos de forma mais abstrata para forma mais detalhada. As principais atividades são divididas em quatro categorias: nível de origem (*source level),* mapeamentos, nível alvo (*target level*) e modelos de composição (*model composition*).

O nível de origem inclui atividades para aquisição ou descoberta de aspectos iniciais e modelos iniciais. Os modelos de aspectos são adquiridos de um repositório de aspecto, e é realizada a verificação para identificar se o aspecto está sendo avaliado ou desenvolvido pelo arquiteto do sistema. Adicionalmente, o modelo inicial é desenvolvido pelo arquiteto, utilizando UML (Diagrama de Seqüência), que realiza as decisões e o que será tratado como aspecto. Essas decisões são baseadas na distinção de requisitos funcionais e requisitos extra-funcionais (Serviços de Qualidade). Assim como técnicas de AOSD são usadas para separar características que são endereçadas como requisitos a partir de funcionalidades de negócio iniciais.

O mapeamento inclui as atividades para desenvolvimento ou aquisição de aspectos correspondentes e modelos iniciais. Essas transformações são definidas por meio da separação do mapeamento para cada aspecto e o modelo inicial (ver Figura 45).

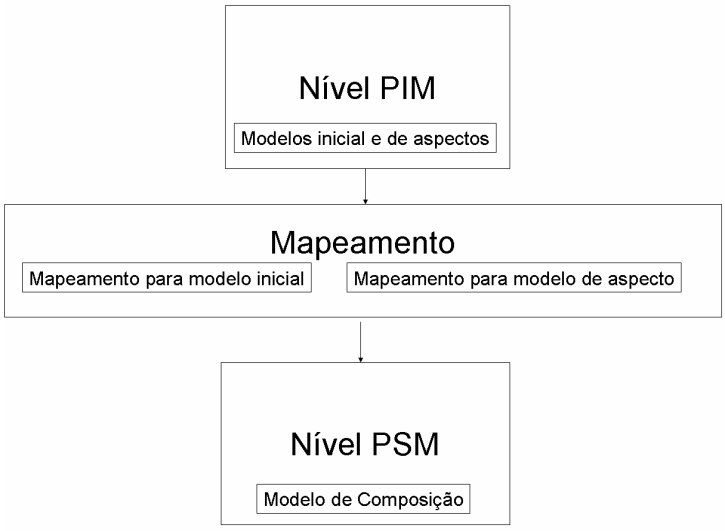


Figura 45: Representação da abordagem AOMDF

O nível alvo inclui atividades para agrupar o mapeamento para o nível de origem, do modelo inicial e modelos de aspectos. Esse nível detalha modelos do projeto detalhado (realizados em UML, e no caso de aspecto, utiliza-se o estereotipo <<aspect>>), que são obtidos por agrupamento das transformações origem e alvo que são especificadas no mapeamento.

O modelo de composição é formado por atividades para instanciar e compor o aspecto e os modelos iniciais usando diretivas de ligações e composições. Os modelos de aspectos precisam ser instanciados antes de serem compostos. Além disso, a instanciação é realizada pela ligação dos elementos do modelo de aspectos para elementos específicos do modelo da aplicação. Dessa forma, a instanciação é feita, o modelo de composição é executado usando a composição de diretivas e um procedimento emparelhado de nomes bases.

Usando a terminologia do MDA, os dois níveis de abstração significativos para modelos são PIM e PSM.

Alguns benefícios propostos pelo AOMDF são:

(i) O framework mostra desenvolvimentos para contextualizar, descrever, e comunicar características transversais com unidades conceituais em vários níveis de abstração.

(ii) A separação horizontal de características como modelos de aspectos e um modelo inicial facilita o mapeamento de especificação de forma separada.

(iii) Mudanças para características transversais podem ser feitas em um lugar, e o modelo de aspectos afetado pela composição do modelo de aspecto modificado com um modelo inicial.

(iv) Os aspectos são freqüentemente aplicados independentemente. O modelo de aspecto e os mapeamentos.

5.3. Transformações de Requisitos para Arquitetura Orientadas a Aspectos

Neste trabalho (SANCHÉZ ET AL., 2006) é apresentada uma abordagem inicial que une MDD e AOSD e uma (semi) automatização do processo para produzir arquiteturas orientadas a aspectos a partir de especificações de requisitos. Um modelo UML é construído para representar a especificação textual dos requisitos expressos em AORE (*Aspect-Oriented Requirements Engineering),* um modelo utilizado para separar características transversais no nível de requisitos. Essas características transversais representam requisitos não-funcionais de alto nível de abstração. Para representar o modelo AORE, foi desenvolvido um *UML Profile*. Partes são modelados como cenários, e cada um pode ter diferentes características não funcionais. Dessa forma, a saída desse processo é um modelo de cenários de requisitos orientados a aspectos. Adicionalmente, transformações MDD usando QVT *(Query, View e Transformations)* (OMG/QVT, 2005) foram predefinidas para gerar um modelo orientado a aspectos de arquitetura, satisfazendo os requisitos funcionais e não-funcionais decididos, e encapsulados as características transversais. Esse modelo arquitetural é expresso usando UML 2.0 Profile para CAM (Pinto, 2005).

Na Figura 46 está representada a abordagem descrita de derivar um modelo de arquitetura OA a partir de um modelo de requisitos OA.

A primeira tarefa consiste em coletar os requisitos através de entrevistas e representar as características do sistema usando a abordagem AORE. A saída desse processo é um especificação textual de requisitos. A abordagem é independente do AORE, porém uma regra é colocada na saída desse processo. Os requisitos funcionais devem ser modelados em alto nível para facilitar a transformação entre modelos, utilizando o UML Profile.

Os requisitos OA são modelados com o UML Prolife como cenários. A saída desse processo é modelo de cenários de requisitos OA. Assim, esse modelo é transformado em um modelo arquitetural OA usando transformações MDD. O modelo arquitetural OA é construído de forma incremental transformando cada cenário individualmente. Para transformar cada cenário, cada requisito funcional é transformado primeiro, e então os requisitos não funcionais são injetados entre a transformação dos requisitos funcionais. Todas as informações do modelo de requisitos são utilizadas no nível arquitetural.

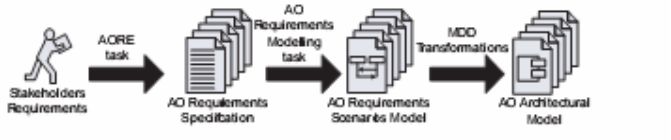


Figura 46: Representação da transformação automática de um modelo de requisitos orientados a aspectos para um modelo de arquitetura orientada a aspectos

O modelo de cenário OA e o modelo arquitetural são baseados em UML Profiles 2.0. Os modelos UML usam representação XMI, utilizada para serializar modelos. A visualização dos modelos é possível através da importação/exportação do XMI através de alguma ferramenta.

5.4. CrossMDA: Arcabouço para integração de interesses transversais no desenvolvimento orientado a modelos

Este trabalho (ALVES ET AL., 2007) apresenta o CrossMDA, um arcabouço que incorpora um processo de transformação para integração de interesses transversais em sistemas orientados a modelos. Esse processo de integração é realizado combinando capacidades de separação existentes nas abordagens MDA e a Programação Orientada a Aspectos (POA). Além disso esse arcabouço utiliza o conceito de separação horizontal de interesses da POA para criar modelos de negócio e aspectos independentes, integrando-os através de transformações MDA. CrossMDA provê um processo de desenvolvimento e um conjunto de serviços e ferramental de apoio para dar suporte ao processo.

O CrossMDA tem como objetivo:

(i) elevar o nível de abstração na modelagem orientada a aspectos através do uso de modelos PIM de interesses transversais independentes do modelo de negócio;

(ii) reusar artefatos de interesses transversais no nível de modelos PIM;

(iii) automatizar o mapeamento de relacionamento de interesses transversais com elementos do modelo de negócio através do processo de transformação MDA;

(iv) facilitar o reuso de artefatos de transformação MDA relacionados a interesses transversais ;

(v) favorecer o reuso de modelos PIM de negócios.

O Processo do CrossMDA está apresentado no Diagrama de Processos da Figura 47, e é composto por atividades organizadas em: Fase 1 – seleção de fontes, Fase 2 – mapeamento e Fase 3 – composição de modelo.

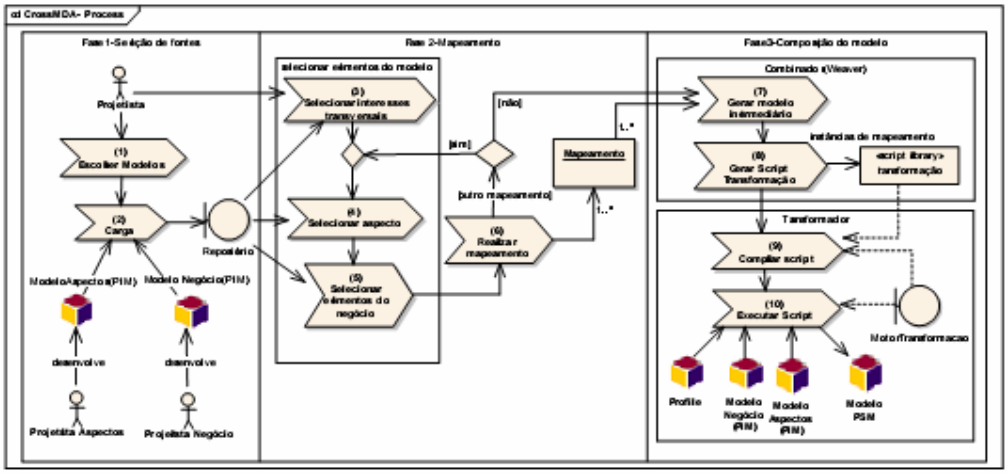


Figura 47: Processo CrossMDA

A fase 1 engloba as atividades (1) e (2) (ver Figura 47). A atividade (1) é responsável por realizar as escolha dos modelos PIM fontes a serem utilizados durante o processo de transformação, e a atividade (2) é responsável pela carga e persistência dos modelos no repositório metadados. Os modelos fontes podem ser modelos de aspectos, que consiste na representação abstrata, composto por classes e esteriótipos <<aspect>>, e organizados em pacotes, e modelo de negócio, composto por entidades, classes de serviços, relacionamentos, restrições, diagramas de classes e interação, além dos demais elementos que representam toda a modelagem do processo de negócio.

A fase 2 é responsável pelo mapeamento dos tipos de relacionamentos entre aspectos e os elementos do modelo de negócio. Essa fase inicia com a atividade (3) que permite ao projetista selecionar os pacotes de interesses transversais que são relevantes

ao domínio da aplicação. Em seguida, é iniciado o processo repetitivo de definição de relacionamento que engloba as atividades (4), (5) e (6). A atividade 4 provê a seleção dos elementos aspectuais; a atividade (5) é responsável pela seleção dos elementos de negócio; a atividade (6) realiza o mapeamento final do relacionamento armazenando os elementos selecionados na atividades (4) e (5) juntamente com o tipo designador do ponto de junção (*join point)* e os tipos de adendo (advice) selecionados no modelo do mapeamento.

A fase 3 é a responsável pela composição de um novo modelo, incluindo todos os elementos de serviço existentes e os novos elementos que representam as instâncias de aspectos mapeados em um nível já dependente de plataforma computacional (PSM). Essa fase é composta por quatro atividades que representam a combinação de modelos (*weaving)* e a transformação. A fase é iniciada com as atividades (7) e (8) do combinador (*weaver).* A atividade (7) é responsável por gerar um modelo intermediário a partir dos relacionamentos mapeados da fase 2. O modelo intermediário é uma representação que contém a hierarquia da composição de uma instância de uma classe aspecto e a sua dependência com o elemento de negócio ao qual se relaciona. Em seguida, a atividade (8) é iniciada, cuja a responsabilidade é transformar o modelo intermediário em uma especificação formal através da geração de um programa de transformação baseada na especificação MOF QVT da OMG. Além disso, as atividades (9) e (10) representam as funções do transformador de modelos, e consistem respectivamente em compilar e executar o programa de transformação gerado pelo combinador de modelos.

O CrossMDA provê serviços de:

(i) persistência de modelos, responsável por implementar as operações básicas para permitir a carga e persistência de modelos e as operações para navegar, recuperar e instanciar novos elementos de modelos existentes. E a realização dessa tarefa é realizada por um serviço de respositório para persistência de metadados.

(ii) mapeamento de modelos, provê mecanismos para gerenciar o mapeamento dos relacionamentos entre os aspectos e os elementos de negócio que é uma atividade chave do processo do CrossMDA. Os mapeamentos suportados pelo CrossMDA são: (i) pontos de atuação, que seguem o padrão de especificação dos pontos de junção da abordagem POA e da linguagem AspectJ, e (ii) intertipos;

(iii) combinador, consiste em integrar o modelo de aspectos ao modelo de negócio gerando instâncias dos aspectos selecionados e as associações destas com os elementos de negócio.

(iv) transformador de modelos, essa atividade é iniciada quando um programa de transformação, gerado pelo combinador, necessita ser compilado e executado. O CrossMDA fornece um serviço para compilar e executar o programa de transformação e dessa forma gerar o novo modelo, através através do motor de transformação de ATL, que inclui uma máquina virtual (ATLvm) e um compilador.

5.5. Framework Dirigido a Modelos Orientados a Sujeito

Este trabalho (AMAYA; GONZÁLES; MURILLO, 2006) apresenta uma abordagem para integração de MDA e AOSD. Cada nível abstrato do MDA é constituído de modelos, e cada modelo corresponde a um aspecto. Os aspectos (modelos) são tratados e transformados em um processo interativo incremental integrando Modelagem Orientada a Sujeito e MDA. Além disso, os aspectos são modelados no nível CIM e PSM, através do elemento esteriótipo *viewCIM e viewPIM,* respectivamente, suportados pela representação da UML2.0.

Assume-se, nesse trabalho, que as características transversais foram identificadas nos estágios inicias (fora do escopo do trabalho). Cada característica deve ser modelada isoladamente por grupos. Esses modelos devem ser transformados independentemente do processo de desenvolvimento. Os benefícios oferecidos estão em modelar os aspectos em um caminho descentralizado, paralelo e consistente. Os grupos modelam e transformam cada aspecto de um CIM para o PSM, de forma separada, utilizando a Modelagem Orientada a Sujeitos. Dessa forma, cada requisito do sistema (representado por caso de uso) , deve ser projetado e implementado entre entidades independentes ligadas a sujeitos.

O primeiro passo dessa abordagem é a especificação de relacionamentos de composição em XML, e através da ferramenta *Xlinkit,* ocorre o gerenciamento e consistência entre os modelos de cada nível de abstração do MDA (CIM, PIM ou PSM).

As funcionalidades dessa ferramenta são adaptadas para: (i) verificar aspectos (modelos) de um mesmo nível de abstração com relacionamentos de composição (verificação horizontal), através do desenvolvimento de regras que validam e

identificam conflitos nesses tipos de relacionamentos entre os modelos; (ii) verificar consistência entre um modelo e uma transformação um modelo abstrato para um mais concreto e, (iii) usar *LinkBase* como documento para navegar entre os relacionamentos de composição entre aspectos, como origem para suportar rastreabilidade automática entre diferentes modelos de abstração e controlar o impacto de mudanças.

O segundo passo é criar regras para verificar e estabelecer diferentes especificações na composição dos arquivos XML. O terceiro passo é avaliar *viewPIMs* em formato XML. Após a execução desses três passos, *Xlinkit* é executado para processar os modelos e os relacionamentos de composição, especificados em regras. O *LinkBase* é gerado em formato XML.

Adicionalmente, a abordagem apresenta a integração da Modelagem Orientada a Sujeito e MDA, habilitando a adição ou modificação de comportamentos e estruturas em um modelo já implementado, por meio de um processo de desenvolvimento incremental.

O uso de XML estabelece relação entre os modelos e relacionamentos de composição entre elementos de diferentes modelos, à medida que, como os modelos são desenvolvidos em diferentes localizações, torna-se quase impossível estabelecer relacionamentos de composição com a UML. Assim como, XML pode ser facilmente adaptado em uma ferramenta para verificar consistências entre os modelos que são modelados separadamente.

A Figura 48 representa resumidamente essa abordagem. Dentro de <*systemModel*> temos os requisitos do sistema (representados por casos de uso), e em <*viewCIM*> outros requisitos relacionado ao sistema. Em <*viewPIM*> temos os sujeitos que são transformados em entidades isoladas no nível PSM (chamado na abordagem de <*viewPIM*>*,* e o código é gerado a partir desse nível de abstração, além dos aspectos modelados. O XML é utilizado para relacionar os modelos e seus elementos definidos em <*systemModel*> *e* <*viewCIM*>, e entre <*systemModel*> *e* <*viewPIM*>

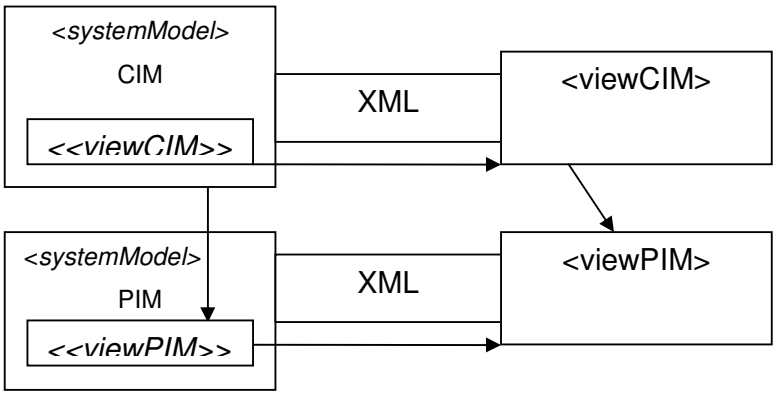


Figura 48: Representação da abordagem do Framework Dirigido a Modelos Orientados a Sujeito

5.6. Comparação

Para efeitos comparativos das abordagens dos trabalhos listados acima com a nossa, categorizamos as principais informações envolvidas nesses trabalhos na tabela abaixo.

Tabela 5: Comparativo entre abordagens

Abordagem	Uso de metamodelos	Ferramenta para Automatização do Processo de transformação	Transformação entre as fases iniciais do processo de desenvolvimento de Software	Utilização puramente da UML	Direcionadas a MDD	Direcionadas a DSOA
Relação de Requisitos com Arquitetura (CHITCHYAN ET AL., 2005)	Não	Não	Sim	Não	Não	Sim
AOMDF (SIMMONDS, ET AL., 2005)	Sim	Sim	Trata apenas o Projeto Detalhado	Não	Sim	Sim
Transformações MDD de requisitos para arquitetura de software orientada a aspectos (SANCHÉZ ET AL., 2006)	Sim	Sim	Sim, requisitos e arquitetura	Sim	Sim	Sim
CrossMDA (ALVES ET AL., 2007)	Sim	Sim	Não	Sim	Sim	Sim
Framework dirigido a modelos orientados a sujeito	Sim	Sim	Não	Não	Sim	Sim

(AMAYA; GONZÁLES; MURILLO, 2006)						
Transformações entre Modelos Orientados a Aspectos: dos Requisitos ao Projeto Detalhado	Sim	Sim	Sim	Não	Sim	Sim

A Relação entre Requisitos e Arquitetura (CHITCHYAN ET AL., 2005) endereça a distância existente na relação de requisitos e arquitetura OA, utilizando para a fase de requisitos o AORE (*Aspect-Oriented Requirements Engineering)* AOAD (*Aspect-Oriented Architecture Design)* para a fase arquitetural. O presente trabalho vai além pois envolve a transformação entre modelos de nível requisitos (AOV-graph), nível arquitetural (AspectualACME) e modelos de nível de projeto (aSideML), porém inserindo essas transformações no contexto do MDD. Além disso, disponibiliza um ambiente para execução dessas transformações, que são especificadas em ATL.

O AOMDF *(Aspect-Oriented Model Driven Framework)* (SIMMONDS, ET AL., 2005), aplica a filosofia MDD, no desenvolvimento de aplicações orientada a aspectos (AOSD), focando na fase de projeto detalhado, diferentemente do nosso trabalho que enfoca na integração e transformações entre três fases: requisitos, arquitetura e projeto detalhado. As transformações envolvidas no AOMDF são realizadas utilizando a linguagem de transformação QVT, o nosso trabalho utiliza ATL. A linguagem QVT possui algumas desvantagens, uma delas é a instabilidade ocasionada pela não finalização de algumas partes da sua especificação. A realização da transformação com QVT possui um nível de complexidade maior, que faz diferença ao ser utilizada. Porém, muitas indústrias estão envolvidas com o seu desenvolvimento, assim como existem ferramentas para dar suporte à execução dessa linguagem. A linguagem ATL também possui o suporte de ferramentas para sua execução, provê suporte à especialização e composição, atomicidade, pré-condições e pós-condições, assim como estabelece o uso de bibliotecas que tornam componentes da linguagem ATL reusáveis e adaptáveis em diferentes ambientes, e pode ser aplicada em casos mais genéricos e abstratos.

As Transformações MDD de Requisitos para Arquitetura de Software Orientada a Aspectos (SANCHÉZ ET AL., 2006), apresenta uma estratégia inicial que une MDD e AOSD e uma (semi) automatização do processo para produzir arquiteturas orientadas

a aspectos, a partir de especificações de requisitos, representadas, respectivamente, por AOAD (*Aspect-Oriented Architecture Design,* representada pela *UML 2.0 Profile for CAM* (Pinto, 2005))*)* e AORE *Aspect-Oriented Requirements Engineering,* representada pela *UML Profile)*, para realização dessas transformações utiliza a linguagem QVT, que possui um nível de complexidade maior nas suas transformações. Nossa proposta engloba as fases de requisitos e arquitetura, inclui uma continuidade e integração com a atividade de projeto detalhado, e para automatização desse processo utiliza ATL, que possui maiores recursos para reutilização em outros ambientes e abordagens. Além disso, não utiliza a UML, apesar de usar aSideML, que é baseada em UML, também usa uma ADL para descrição arquitetural. Uma das vantagens da ADL é a disponibilidade de ferramentas que inclui facilidades de verificação. Ambas as abordagens propõem um processo automatizado baseado em MDD, e possuem automatização desse processo com suporte de ferramentas.

O CrossMDA (ALVES, 2007) é um arcabouço para dar suporte à integração entre modelagem com aspectos e MDA. Em CrossMDA, aspectos são modelados explicitamente apenas no nível de projeto, enquanto que Marisa-MDD trabalha com modelos de arquitetura também. CrossMDA adota um modelo de aspectos simples, porém baseado em "profiles" de UML, para prover uma representação abstrata e independente de plataforma (PIM) e usa outra notação, AODM (STEIN; HANENBERG; UNLAND, 2002), para descrição de modelos PSM. Nossa abordagem, parte de modelos de requisitos em AOV-graph, transforma em AspectualACME, e gera modelos de projeto em aSideML. Ou seja, nossa estratégia integra as 3 fases e garante que informações de uma fase são representadas na fase seguinte. CrossMDA e MaRiSA-MDD utilizam templates ATL para dar suporte às transformações. Porém, CrossMDA dá apoio à transformação apenas entre modelos no nível de projeto. Tanto CrossMDA como MaRiSA-MDD definem um processo de transformação entre modelos.

O Framework Dirigido a Modelos Orientados a Sujeito (AMAYA; GONZÁLES; MURRILLO, 2006) apresenta uma proposta para integração de MDA e AOSD. Nessa estratégia cada nível MDA é constituído de modelos, e cada modelo corresponde a um aspecto. Os aspectos (modelos) são tratados e transformados em um processo interativo incremental integrando Modelagem Orientada a Sujeito e MDA. O enfoque é na transformação de CIM para PSM. Nossa abordagem propõe a transformação entre as fases de Requisitos (AOV-graph – nível de abstração CIM),

Arquitetura (AspectualACME – nível de abstração PIM) e Projeto Detalhado (aSideML – nível de abstração PIM), onde temos maior refinamento e propagação de informações entre essa fases e níveis de abstração. O Framework Dirigido a Modelos Orientados a Sujeito e MARISA-MDD oferece automatização dos processos de transformação. Todavia, nosso trabalho oferece um ambiente de desenvolvimento e execução de transformações especificadas na linguagem ATL, provendo rigor semântico na definição de transformações, verificação se os modelos gerados estão bem formados, enquanto que o Framework provê transformações utilizando XMLs, e uma ferramenta Xlinkit que realiza a verificação da consistência da especificação XML.

6. Conclusão

Abordagens orientadas a aspectos associadas a diferentes atividades do processo de desenvolvimento de software são, em geral, independentes. Além disso, os modelos associados a cada uma das atividades não estão naturalmente alinhados ou inseridos em um processo coerente. Dessa forma, os desenvolvedores ficam com a sobrecarga de: conhecer diversas abordagens OA, modelos e artefatos associados; compreender como aspectos são representados em cada uma das atividades; e definir a correspondência entre artefatos OA gerados em cada atividade. Além disso, se a correspondência entre modelos e artefatos OA associados às diversas atividades não for bem definida, pode-se perder informações importantes ou mesmo introduzir erros de uma atividade para outra.

Para endereçar esse problema, nesse trabalho propusemos a integração entre desenvolvimento orientado a aspectos (DSOA) e desenvolvimento baseado em modelos (MDD) onde define-se os modelos OA de cada atividade do processo de desenvolvimento, bem como a correspondência entre eles. Centramos o trabalho nas atividades de requisitos, arquitetura e projeto detalhado usando as linguagens AOV-graph, AspectualACME e aSideML, respectivamente. Definimos um processo baseado em modelos onde os mapeamentos entre os modelos de cada linguagem foram especificados e implementados de forma que pode-se automaticamente propagar modelos de uma atividade para outra atividade.

Os modelos associados às atividades de requisitos, arquitetura e projeto detalhado foram construídos com base em rigores semânticos especificados por linguagens para descrição de modelos. Dessa forma, utilizamos a linguagem KM3 (*Kernel MetaMetaModel*) para descrição de metamodelos para AOV-graph, AspectualACME e aSideML. Assim como, o componente TCS (*Textual Concret Syntax*) para obtermos a representação textual de modelos AOV-graph para modelos AspectualACME, e de modelos AspectualACME para modelos aSideML. Além disso, a linguagem de transformação ATL (*ATLAS Transformation Language*) foi usada para especificarmos as regras de transformação entre modelos. Essas transformações foram implementadas na IDE Eclipse.

Com os modelos gerados automaticamente é possível obter uma estruturação inicial da arquitetura e projeto detalhado, que são valiosas por trazerem as informações levantadas desde a fase de requisitos. Contudo, devido a algumas limitações existentes,

como por exemplo, em arquitetura as informações sobre estilo arquitetural mais adequado, protocolos de conexão, dentre outras, são informações que não estão presentes no modelo de arquitetura gerado, pois os requisitos não as descrevem. Já no projeto detalhado, apenas uma interface transversal pode ser implementada pelo aspecto, devido a não ser possível diferenciar as funcionalidades através das portas especificadas no modelo de arquitetura.

Dessa forma, temos a geração de modelos incompletos devido ao *gap* semântico existente entre as linguagens. Todavia, esses modelos podem ser refinados pelo arquiteto ou projetista de software com informações adicionais. Grande parte do trabalho do arquiteto e do projetista é realizado automaticamente pela ferramenta, portanto, a função deles será ajustar os modelos gerados incluindo as informações extras. Com o ambiente MARISA-MDD garante-se que todas as informações presentes nos requisitos são representadas nos modelos das fases seguintes, mesmo que esses modelos não estejam completos.

Além disso, o mapeamento realizado considera a geração de apenas um arquivo como resultado das transformações. Não contempla a formação de diferentes opções de arquitetura e projeto detalhado.

Assim como, os nomes utilizados em requisitos, que são repassados para nomes dos elementos em arquitetura, e de arquitetura, para nomes dos elementos em projeto detalhado podem sem modificados pelo Arquiteto afim de se manter um padrão para nomes, uma opção para geração desses nomes mais padronizados seria a utilização dos *topics* de AOV-graph, que armazenam palavras chaves para nome dos requisitos, e conseqüentemente podem auxiliar na definição de nomes de classes na fase de implementação.

A ferramenta MARISA-MDD, as transformações e os exemplos completos estão disponíveis em: http://www.ppgsc.ufrn.br/~analuisa/marisa/marisamdd/atlTransformations/.

6.1. Contribuições

As contribuições desse trabalho incluem:

- a integração das atividades de requisitos, arquitetura e projeto detalhado orientado a aspectos através de transformações baseadas em modelos

- a definição do meta-modelos de AOV-graph, AspectualACME e aSideML em KM3
- a especificação de regras de transformação entre AOV-graph e AspectualACME e entre AspectualACME e aSideML, procurando expressar, em todas as atividades, as informações contidas na atividade predecessora. Para isso, muitas vezes, houve a necessidade de incluir elementos adicionais no modelo. Por exemplo, no modelo AspectualACME foram utilizadas as Propriedades e especificados nessas propriedades os atributos *comesFrom, source, elementType, topics, contributions, correlations* para armazenar informações provenientes de AOV-graph. E no modelo aSideML foram utilizadas as *Tags* para armazenar as informações de arquitetura provenientes dos requisitos.
- a implementação das regras de transformação em ATL permitindo que ocorra um processo automatizado de transformações entre os modelos das 3 atividades supracitadas. O processo automatizado permite que sejam mantidas as informações
- a disponibilização de um ambiente integrado, MARISA-MDD, que permite a criação de modelos com base em metamodelos bem definidos e regras de transformação entre seus elementos. Tal ambiente facilita o acesso às informações aos modelos do AOV-graph, AspectualACME e aSideML originados durante o processo de desenvolvimento, permitindo que requisitos possam ser adicionados e propagados facilmente para arquitetura, e de arquitetura para projeto detalhado. Além disso, oferece suporte a rastreabilidade entre os modelos.
- a validação das regras usando dois estudos de caso, o *Health Watcher* e o *Mobile Media*, comumente usados para avaliação de estratégias orientadas a aspectos.

6.2. Trabalhos Futuros

Objetivando dar continuidade à pesquisa desenvolvida nessa dissertação, alguns trabalhos futuros podem ser relacionados, tais com:

- Utilização de outras linguagens representativas das fases de requisitos, arquitetura e projeto detalhado, (por exemplo,

Theme/UML), no processo de desenvolvimento proposto, visando identificar possíveis generalizações de regras de mapeamento

- Inclusão de transformações para a atividade de implementação.
- Definição e implementação de regras de transformação inversa de Projeto Detalhado para Arquitetura de Software
- Utilização de métricas para avaliar os resultados das transformações criadas. As possíveis métricas a serem utilizadas como base, são as fornecidas pelo documento *MDD Engineering metrics Baseline*, que está relacionado a Engenharia de Métricas.
- Especificação e construção de um ambiente de configuração e reconfiguração dinâmica que providencie suporte a transformações orientadas a aspectos entre as fases do ciclo de desenvolvimento de software.

Referências

AKSIT, M. Systematic analysis of crosscutting concerns in the model-driven architecture design approach. In: SYMPOSIUM ON HOW ADAPTABLE IS MDA?. 20., Netherlands . **CTTI Workshop Proceedings.** (The Netherlands), 2005.

ALVES, M. P., et al. CrossMDA: Arcabouço para integração de interesses transversais no desenvolvimento orientado a modelos. In: SIMPÓSIO BRASILEIRO DE COMPONENTES, ARQUITETURAS E REUTILIZAÇÃO DE SOFTWARE (SBCARS), Campinas, 2007. **Anais do Simpósio Brasileiro de Componentes, Arquitetura e Reutilização de Software.** Porto Alegre: Sociedade Brasileira de Computação, 2007.

AMAYA, P.; GONZÁLE, C; MURILLO, J. M. Towards a subject-oriented model-driven. In: INT. WORKSHOP ON ASPECT-BASED AND MODEL-BASED SEPARATION OF CONCERNS IN SOFTWARE SYSTEMS(AB-MB-SOC), 1., European Conference on MDA - Foundations and Applications (ECMDA-FA), 2005. **Proceedings of the Symposium on Software Architectures and Component Technology**. Nuremberg (Germany), 2005.

AOSD (Aspect-Oriented Software Development). Disponível em: <http://www.aosd.net>. Acesso em: janeiro de 2008.

ATLAS (Atlas Transformation Language). Disponível em: http://www.eclipse.org/m2m/atl>. Acesso em: janeiro de 2008.

BAKKER, J et al. **Characterization of early aspects approaches**. Chicago, USA**:** The Early Aspects Workshop, 2005.

BANIASSAD, E. CLARKE, S. **Theme: An approach for aspect-oriented analysis and design**. In: Proc. of the 7th International Conference on Software Engineering (ICSE'04). Scotland, 2004.

BASS, L. et al. **Software Architecture in Practice.** Boston: Addison Wesley, 2006.

BATISTA T. et al. **Aspectual Connectors**: supporting the seamless integration of aspects and ADLs. 2006. Trabalho apresentado no Simpósio Brasileiro de Engenharia de Software - SBES 2006, Florianópolis, Santa Catarina, 2006, p. 17-32.

BENEDÍ, J. P. et al. **Prisma: towards quality, aspect oriented and dynamic software architectures.** 2003. Trabalho apresentado no 3º Internacional Conference On Quality Software – QSIC, Washington, USA, 2003.

BUDINSKY, F. **Eclipse Modeling Framework:** A developers´s Guide. England: Addisson Wesley, 2004.

CARVALHO, A. E. S. de. et al. Uma estratégia para implantação de uma gerência de requisitos visando a melhoria dos processos de software. In: WORKSHOP EM ENGENHARIA DE REQUISITOS, Buenos Aires, Argentina, 2001. **Anais ...** Buenos Aires, Argentina, 2001.

CHAVEZ, C. V. **Um Enfoque Baseado em Modelos para Design Orientado a Aspectos**. Tese (Doutorado) – Pontifica Universidade Católica do Rio de Janeiro. Rio de Janeiro, 2004.

CHITCHYAN, R. et al. Relating AO Requirements to AO Architecture. In: WORKSHOP ON EARLY ASPECTS, INT. CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, TOOLS AND APPLICATIONS (OOPSLA), 20th., 2005. San Diego, California (USA), 2005.

CHITCHYAN, R. et al. **Survey of aspect-oriented analysis and design approaches**. Technical Report AOSD-Europe-ULANC-9, AOSD Europe, 2005. Disponível em: <http://www.aosd-europe.net/documents/index.htm>. Acesso em: 7 mar. 2008.

CHUNG, L. et al. **Non-Functional Requirements in Software Engineering**. England: Kluwe Academic Publishers, 2000.

CLARKE, S; WALKER, R. **"Composition Patterns: An Approach to Designing Reusable Aspects".** 2001. Trabalho apresentado no 23rd. Proceedings of the ICSE, p. 5-14, 2001.

CLARKE, S.; WALKER, R.. Towards a standard design language for AOSD. In INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 1., The Netherlands, 2003. **Proceedings of the 1st International Conference on Aspect-Oriented Software Development.** Enschede, The Netherlands, 2003. p. 113–119.

CLEMENTS, P. et al. **Document software architectures:** views e beyond. Boston: Addison Wesley, 2005.

CZARNECKI, K.; HELSEN, S. Feature-based survey of model transformation approaches. **IBM Systems Journal**, v. 45, n. 3, p. 621-645, jul. 2006.

ECLIPSE. Disponível em: <http://www.eclipse.org/>. Acesso em: janeiro de 2008.

ESPINDOLA, R. S. de. Uma abordagem baseada em gestão do conhecimento para gerência de requisitos em desenvolvimento distribuído de software. In: WORKSHOP EM ENGENHARIA DE REQUISITOS - WER, Porto, Portugal, p. 87-99, 2005.

ESPINDOLA, R. S. de. **Uma análise crítica dos desafios para engenharia de requisitos em manutenção de software.** In: Workshop em Engenharia de Requisitos , Tandil, Argentina, p. 226-238, 2004.

FIGUEIREDO, E. et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: INTERNATIONAL CONFERENCE ON SOFTWARE

ENGINEERING (ICSE), 30., 2008. **Proceedings of the 30th international conference on Software engineering.** Leipzig, Germany, p. 10-18, Maio 2008.

FILMAN, R. E. et al. **Aspect-Oriented Software Development**. Boston: Addison Wesley, 2005.

FIORINI, S. T. **Organizando Processos de Requisitos.** In: WORKSHOP EM ENGENHARIA DE REQUISITOS – WER , Maringá, Paraná, 1998. p. 1-8.

GALSTER, M. et. al. Transition from Requirements to Architecture: A Review and Future Perspective. In: ACIS INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ARTIFICIAL INTELLIGENCE, NETWORKING, AND PARALLEL/DISTRIBUTED COMPUTING (SNPD'06), 70., 2006. **Proceedings of the 7th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2006).** Nevada, USA, 2006. p. 9-16.

GARLAN, D.; SHAW, M. An Introduction to Software Architecture. In: AMBRIOLA, V.; TORTORA, G. (Ed.). **Advances in Software Engineering and Knowledge Engineering.** Singapore: World Scientific Publishing Company, 1993. p. 1-39. (Series on Software Engineering and Knowledge Engineering; v. 2).

GARLAN, D. et al. ACME: an architecture description interchange language. In: CASCON, 97., 1997. **Proceedings …** Toronto, Canadá, 1997.

GARLAND, J.; Anthony, R. **LargeScale Software Architecture :** a practical guide using UML. São Paulo: John Wiley & Sons, 2002.

GIORGINI, P. et al. **Reasoning with goal models.** 2002. Trabalho apresentado no 21th. International Conference On Conceptual Modeling. London, UK, 2002.

GONZALES, B. et al. **Visual variability analysis for goal models.** 2004. Trabalho apresentado no IEEE International Requirements Engineering Conference, Washington, USA, 2004.

GRUNDY, J. Multi-perspective specification, design and implementation of software components using aspects. **International Journal of Software Engineering and Knowledge Engineering**, v. 10, n. 6, p. 713–734, 2000.

INRIA. Disponível em: <http://modelware.inria.fr/>. Acesso em: janeiro de 2008.

JACOBSON, I., NG, P.: **Aspect-Oriented Software Development with Use Cases.** San Francisco, USA: Addison-Wesley, 2005.

JOUAULT, F; BÉZIVIN, J.; KURTEV, I. KM3: a DSL for Metamodelo Specification. In: IFIP INTERNATIONAL CONFERENCE ON FORMAL METHODS FOR OPEN OBJECT-BASED DISTRIBUTED SYSTEMS, 8., Bologna, Italy, 2003. **Proceedings *of* 8th *IFIP.*** Bologna, Italy, 2003. p. 171-185.

JOUAULT, F; BÉZIVIN, J.; KURTEV, I.. **TCS: a DSL for the specification of textual concrete syntaxes in model engineering.** GPCE 2006 – Generative Programming and Component Engineering: pp. 249-254, 2006.

KICZALES, G. et al. **Aspect-Oriented Programming**. 1997. Trabano apresentado no Proceedings European Conference on Object-Oriented Programming, 1997.

KONTOYA, G; SOMMERVILLE, I. Requirements engineering with viewpoints. **BCS/IEE Software Engineering Journal**, 1998.

KULKARNI, V., REDDY, S. Integrating aspects with model driven software development. In: AL-ANI, B.; ARABNIA, H., Mun, Y., (Ed.). **Software Engineering Research and Practice**. Las Vegas, Nevada (USA): CSREA Press, 2003. p. 186–197.

LAMSWEERDE, A. van. Goal-Oriented Requirements Engineering: a guided tour. **IEEE Transactional Software Engineering – IEEE,** 2001.

LARMAN, G. **Utilizando UML e Padrões:** Uma Introdução à análise e ao Projeto Orientado a Objetos e ao Processo Unificado**.** Rio de Janeiro: Bookman, 2005.

MACEDO, N. A.; LEITE, J. S. P L. **Elicit@99 um Protótipo de Ferramenta para a Elicitação de Requisitos**. 1999. Anais do Workshop em Engenharia de Requisitos, Buenos Aires, Argentina, Set. 1999. p. 45-55.

MEDEIROS, A. L et al. **Requisitos e Engenharia de Software Orientada a Aspectos: Uma Integração Sinérgica**. Trabalho apresentado no 21º Simpósio Brasileiro de Engenharia de Software (SBES), João Pessoa, PB, out. 2007a. p. 199-218,

_____. **Uma Combinação Harmônica entre Arquitetura de Software e Projeto Detalhado Orientado a Aspectos**. Trabalho apresentado no Latin Workshop on Aspect-Oriented Software Development (LA-WASP 2007), João Pessoa, PB, Out. 2007b. p. 55-66.

_____. **MARISA - Uma Ferramenta para mapeamento bidirecional de Modelos Orientados a Aspectos:** Requisitos e Arquitetura de Software. Trabalho apresetnado no Latin Workshop on Aspect-Oriented Software Development (LA-WASP 2007), João Pessoa - Pb, Out. 2007c. p. 55-66.

MENDES, A**. Arquitetura de Software:** Desenvolvimento Orientado para Arquitetura**.** São Paulo: Campus, 2002.

MOREIRA, A. et al. Multi-Dimensional Separation of Concerns in Requirements Engineering**.** In: INT. CONF. ON REQUIREMENTS ENG. (RE'05), 13., Paris. 2005. **Proceedings …** Paris: IEEE Computer Society, 2005. p. 285-296.

MYLOPOULOS, J. et al. Representing and Using Non-Functional Requirements: a Process-Oriented Approach. **Transactions on Software Engineering – IEEE**, 1992.

NIERSTRASZ, O.; TSICHRITZIS, D. **Object-Oriented Software Composition.** Boston: Prentice Hall International, 1995.

NUSEIBEH, B. **Crosscutting requirements.** Trabalho apresentado no 3º International Conference on Aspect-oriented software development. 2004.

OBJECT MANAGEMENT GROUP (OMG). **MOF QVT Final Adopted Specification** (ptc/05-11-01). Disponível em: <http://www.omg.org/docs/ptc/05-11-01.pdf>. Acesso em: janeiro de 2008.

_____. **MDA Model Driven Architecture.** Disponível em: <http://www.omg.org/mda>. Acesso em: janeiro de 2008.

OBJECT MANAGEMENT GROUP (OMG). **XMI – XML Metadata Interchance.** Disponível em: <www.omg.org/technology/documents/formal/xmi.htm>. Acesso em: janeiro de 2008.

PARK, S. et al. A scenario, goal, and feature oriented Domain Analysys Approach for Developing Software Product Lines. **Journal on Industrial Management & Data Systems**, 2004.

PINTO, M. et al. A Dynamic Component and Aspect-Oriented Platform. **The Computer Journal**, Oxford, UK, 2005.

RASHID, A. et al. **Early Aspects**: a model for aspect-oriented requirements engineering. Trabalho apresentado no IEEE Joint International Requirements Engineering Conference, United States, 2002.

_____. Modularization and composition of aspectual requirements. In: International Conference on Aspect-oriented Software Development, 2., 2003. **Proceedings of the Conference on Aspect Oriented Development. (AOSD'03).** USA: ACM, 2003. p. 11-20.

RENTSCH, T. Object-Oriented Programming**. ACME SIGLAN Notices**, v. 17, n.9, 1982.

ROBINSON, W. N. et al. **Requirements Interaction Management**. New York, USA: ACM Computing Surveys, 2003.

SANCHÉZ, P. et al. Towards MDD Transformations from AO Requirements into AO Architecture. In: EUROPEAN WORKSHOP ON SOFTWARE ARCHITECTURE, 3., 2006; FRENCH CONFERENCE ON SOFTWARE ARCHITECTURE, 2006, Nantes, France, 4-5 September. **Proceedings of the Third European Workshop on Software Architecture (EWSA).** Nates, France: Lecture Notes in Computer Science, 2006. p. 160-174.

SHAW, M.; GARLAN, D. **Software Architecture – Perspectives on an Emerging Discipline**. Upper Saddle River, NJ: Prentice Hall, 1996.

SILVA, L. F. da. **Uma Estratégia Orientada a Aspectos em Modelagem de Requisitos**. Tese (Doutorado) – PUC-Rio, Rio de Janeiro, 2006.

_____. **On the Symbiosis of Aspect-Oriented Requirements and Architectural Descriptions.** Trabalho apresentado no Proc. of Early Aspects co-located with AOSD 2007, Vancouver, Canada, March 2007.

SIMMONDS, D. et al. An aspect oriented model driven framework. In: IEEE INTERNATIONAL ENTERPRISE DISTRIBUTED OBJECT COMPUTING CONFERENCe, 9., 2005. **Proceedings …** The Netherlands: EEE CS, 2005. p.119–130.

SOARES, S. et al. **Implementing Distribution and Persistence Aspects with AspectJ**. Trabalho apresendado no Proceedings of the OOPSLA'02, Washington, USA, 2002.

SOMMERVILLE, I. **Engenharia de Software.** São Paulo: Pearsoned Brasil, 2003.

_____. **Requirements Engineering: a good practice guide.** New York, USA: John Wiley & Sons Ltd, 1997.

STAHL, T. et al. **Model-Driven Software Development, Technology, Engineering, Management**. England: John Wiley & Sons, 2006.

STEIN, D.; HANENBERG, S.; UNLAND, R. **Designing Aspect-Oriented Crosscutting in UML**". Trabalho apresentado no Workshop on Aspect-oriented Modeling with UML, Germany, 2002.

SWEBOOK. **Guide to the Software Engineering Body of Knowledge**. Disponível em: <http://www.swebok.org>. Acesso em: janeiro 2008.

VAROTO, A. C. **Visões em arquitetura de Software**. Dissertação (Mestrado) – USP, São Paulo, 2003.

WINCK, D. V.; G. JUNIOR, V. **AspectJ – Programação Orientada a Objetos em Java.** São Paulo, Novatec, 2006.

YU, Y. et al. **From Goals to Aspects:** discovering aspects from requirements goal models. 2004. Trabalho apresentado no IEEE International Symposium on Requirements Engineering - RE'04, United Kingdom, 2004.

ZANLORENCI, E. P. **Abordagem da Engenharia de Requisitos para Software Legado.** In: WORKSHOP EM ENGENHARIA DE REQUISITOS – WER, Porto, Portugal, 2005. p. 270-284.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo