Uma Estratégia baseada em Metamodelo para Geração de Código Orientado a Aspectos

Everton T. Guimarães[1], Nélio Cacho[1], Thais V. Batista[1]

Universidade Federal do Rio Grande do Norte (UFRN) – Natal – RN – Brasil

everton.dev@gmail.com, neliocacho@gmail.com, thais@ufrnet.br

Abstract. *This paper presents MARISA-AOCode, a strategy based on the Model Driven-Development (MDD) paradigm that supports model transformation from detailed project to different Aspect-Oriented Programming languages. MARISA-AOCode defines transformations rules between aSideML, a modeling language to aspect-oriented detailed project, and MetaSpin, a generic meta-model for aspect-oriented programming languages. The instantiation of the generic meta-model provided by MARISA-AOCode, the Metaspin model is illustrated by a transformation from Metaspin to the AspectLua language.*

Resumo. *Esse artigo apresenta MARISA-AOCode, uma estratégia baseada em MDD que suporta a transformação de projeto detalhado para diferentes linguagens de Programação Orientada a Aspectos. MARISA-AOCode define regras de transformação entre aSideML, uma linguagem de modelagem para projeto detalhado orientado a aspectos, e Metaspin, um metamodelo genérico para linguagens de programação orientadas a aspectos. A instanciação do modelo genérico provido por MARISA-AOCode, o modelo Metaspin, é ilustrada através da transformação do Metaspin para a linguagem AspectLua.*

1. Introdução

Nas últimas décadas, a engenharia de software vem pesquisando diversas formas de melhorar a qualidade e reduzir o custo do desenvolvimento de sistemas de software. Além de ter que lidar com constantes mudanças nos requisitos, projetos de software geralmente precisam tratar de mudanças de tecnologias. Novos paradigmas, tais como o desenvolvimento de software dirigido por modelos (MDSD) e o Desenvolvimento Orientado a Aspectos (AOSD) vêm sendo propostos como uma possível solução para os problemas relacionados ao projeto e evolução de sistemas de software.

O desenvolvimento de software dirigido por modelos (MDSD) [Stahl et al, 2006] usa uma abordagem interativa e *top-down* para desenvolver sistemas, onde modelos são os principais artefatos de desenvolvimento. Modelos são utilizados para capturar as principais características do sistema a ser desenvolvido. Modelos são definidos por meio de linguagens de modelagem que privilegiam o uso de abstrações que permitem especificar propriedades de um sistema segundo a perspectiva dos clientes, arquitetos e desenvolvedores. As tecnologias que suportam o MDSD buscam gerar automaticamente ou semi-automaticamente artefatos executáveis a partir de modelos. Modelos definidos em diferentes níveis de abstração são expandidos repetidamente em artefatos executáveis através de mapeamentos e transformações.

Por sua vez, o Desenvolvimento Orientado a Aspectos (AOSD) [Filman et al, 2005] é um paradigma de desenvolvimento que permite melhorar a modularidade de sistemas de software através do encapsulamento de conceitos transversais em abstrações chamadas aspectos. AOSD suporta a implementação destes conceitos por meio de uma sistemática identificação, representação e composição do conceito transversal com o código base da aplicação.

Apesar de estas duas abordagens serem diferentes em vários aspectos - MDSD define abstrações baseadas em modelos e AOSD oferece mecanismos para modularizar e compor conceitos transversais – elas possuem algumas características comuns. Vários estudos, tais como [Cottenier et al, 2007], [Evermann, J. 2007], [Fuentes e Sánchez, 2007] e [Groher e Baumgarth, 2004] têm explorado a combinação destes dois paradigmas. Ambos, MDSD e AOSD são tecnologias que visam melhorar a modularidade e a manutenibilidade dos sistemas de software. Estes dois paradigmas são complementares e podem se beneficiar mutuamente quando utilizados em conjunto. Por exemplo, o AOSD poderia ser utilizado para separar, no nível de modelos, os conceitos transversais da funcionalidade base da aplicação. Regras de mapeamento podem ser utilizadas para gerar artefatos executáveis a partir dos conceitos transversais definidos nos níveis superiores. Com isso, a separação dos conceitos transversais pode ser mantida em todos os níveis de abstração, garantindo uma melhor modularidade e manutenibilidade do sistema de software.

A definição de regras de mapeamento entre modelos abstratos e artefatos executáveis não é uma tarefa trivial. Em geral, isto ocorre por dois motivos: (i) a necessidade de gerar código que respeite a sintaxe/semântica das linguagens de programação requer a definição de um número elevado de regras de transformações, (ii) por não estarem consolidadas, as linguagens de programação orientadas a aspectos (OA) fornecem diferentes abstrações e mecanismos de composição, o que na prática complica ainda mais o processo de geração das regras de mapeamento.

No sentido de abstrair a complexidade de mapeamento existente entre os modelos abstratos definidos em nível de projeto detalhado e os artefatos executáveis gerados por várias linguagens OA, este trabalho apresenta *MaRiSA-AOCode*, uma abordagem baseada em transformações de modelos que, a partir de um projeto arquitetural orientado a aspectos, gera um esqueleto de código orientado a aspectos que é independente de uma linguagem orientada a aspecto específica. A independência é conferida pelo uso do Metaspin [Brichau et al, 2006], um metamodelo genérico de linguagens de programação orientadas a aspectos. A abordagem proposta visa permitir o mapeamento e navegação entre as atividades do ciclo de Desenvolvimento de Software Orientado a Aspectos, e a propagação de informações entre elas, definindo transformações dos artefatos da fase de projeto detalhado e codificação, através de regras de mapeamento. Para ilustrar a viabilidade da abordagem proposta em relação às atividades de projeto detalhado e codificação, foram escolhidas as linguagens aSideML [Chavez, 2004], uma linguagem de projeto detalhado orientada a aspectos, o Metaspin [Brichau et al, 2006] um modelo genérico para a codificação orientada a aspectos e a linguagem AspectLua [Cacho et al, 2005]. Portanto, o mapeamento entre projeto detalhado e código fonte é realizado através da especificação de transformações entre modelos aSideML e Metaspin e deste para a linguagem AspectLua.

Esse artigo está estruturado da seguinte forma. A seção 2 contém uma breve descrição dos conceitos básicos relativos à MDD, Metaspin e ATL. A seção 3 apresenta uma visão geral da abordagem proposta. A seção 4 define o mapeamento entre os elementos de aSideML e Metaspin (seção 4.1) e também define o mapeamento entre os elementos de Metaspin e AspectLua (seção 4.3). A seção 5 apresenta a ferramenta MaRiSA-AOCode e descreve o estudo de caso exemplificando a implementação das regras de mapeamento. As seções 6 e 7 apresentam trabalhos relacionados e conclusões.

2. Conceitos Básicos

2.1. Metaspin

O Metaspin [Brichau et al, 2006] é um metamodelo que foi concebido como resultado do estudo [Brichau, Haupt, 2005] de diversas linguagens de POA [Kiczales et al, 97], onde foram analisadas as semelhanças fundamentais, bem como as variabilidades entre as linguagens. O metamodelo visa representar os conceitos essenciais das linguagens de programação orientadas a aspectos (POA), tais como *join point, pointcut*, *advice*, e *weaving*. Os *join points* são os pontos onde o aspecto deve atuar, ou seja, são pontos bem definidos na execução de um programa, tais como execução ou invocação de métodos. Os *pointcuts* representam um conjunto de *join points*, onde através deles podem ser obtidos objetos em execução, argumentos de funções e atributos. O *advice* descreve a ação a ser tomada quando um ponto de junção é alcançado. O *advice* pode ser configurado para atuar antes (*advice before*), depois (*advice after*) e durante (*advice around*) o *join point*. O *weaving* é o processo de junção do código aspectual com o código base da aplicação.

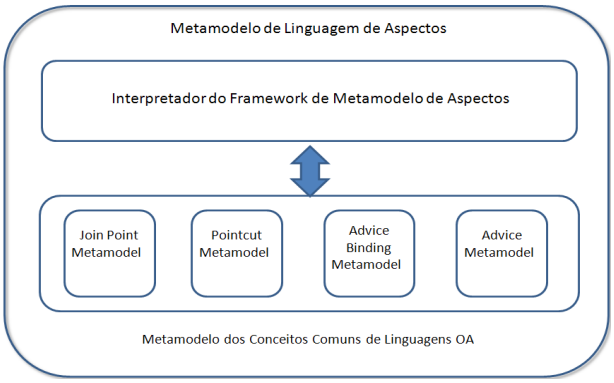

Figura 1. Partes do Metamodelo de Linguagens de Programação OA.

A principal característica do Metaspin é a possibilidade de representar qualquer linguagem de POA. Adicionalmente, o Metaspin é divido em quatro sub-metamodelos, conforme ilustra a Figura 1. São eles:

(i) **Pointcut Language Metamodel**: modela a funcionalidade do *pointcut*. Os *pointcuts* são representados como predicados sobre os *join points*. Isso significa que expressões de pointcut avaliam (*evaluate*) os *join points*. O conceito de *pointcut* é representado no metamodelo como um *Join Point Selector*. A figura 2(a) mostra que um seletor de *join point* (*Join Point Selector*) pode ser um seletor primitivo (*Primitive Selector*) o qual aplica um predicado simples em um *join point* ou pode ser um seletor composto (*Composed Selector*) que aplica múltiplos predicados a um *join point*.

(ii) **Advice Metamodel**: As ações que podem ser disparadas através de aspectos em *join points* particulares são descritos usando esse metamodelo. O *advice* expressa a funcionalidade que pode ser invocada por um aspecto. O Metaspin modela *advices* concretos usando ações de *advice* (*Advice Action*) (Figura 2b) que são compostos em uma estrutura de árvore, representando o *advice*. Tal estrutura pode ser composta por: expressões primitivas (*Primitive Base Level Expression*); (ii) expressões compostas no nível base (*Composed* Action) e expressões em meta-nível (*Meta-level Expression*).

(iii) **Advice Binding Metamodel**: define como os aspectos *são instanciados,* modularizados e como o*s advices* são vinculados aos *pointcuts.* Os aspectos são formados de *pointcuts*, *advices* e definições de variáveis. Um aspecto contém seletor de ligações de advice (*Selector Advice Binding*) (ver Figura 2c). Esses seletores são responsáveis por relacionar o *Join Point Selector* com as definições de *advice*.

(iv)***Join Point Metamodel* -** representa o conceito de *join points,* permitindo a representação de *join points* estáticos e dinâmicos. O *join point* dinâmico é aquele disponível a partir da execução de um programa, enquanto que o *join point* estático é aquele disponível a partir do texto do código fonte. O metamodelo de *join point* mostrado na Figura 2(d) define que um *join point* é formado de uma parte estrutural (*Structural Join Point Part*) e uma parte comportamental (*Behavioral Join Point Part*), onde a primeira refere-se a um local no código fonte de um programa onde o aspecto irá atuar, enquanto que a outra se refere ao estado de execução de um programa.

2.2. Desenvolvimento Dirigido por Modelos (MDD) e ATLAS *Transformation Language* (ATL)

A essência do Desenvolvimento Dirigido a Modelos [Stahl et al., 2006] está na especificação de modelos e na transformação desses modelos em outros modelos ou em artefatos de software, de forma a permitir a comunicação de diferentes fases do processo de desenvolvimento de software, onde os artefatos produzidos em cada fase podem ser representados por modelos. Os passos da transformação entre modelos podem ser realizados por ferramentas automatizadas, as quais reduzem os erros de programação e os custos de desenvolvimento. O mapeamento dos modelos de uma abstração para outra é feito através da definição de transformações. Para definir transformações entre modelos são usadas linguagens de transformação. ATL [Jouault et al, 2005] é uma linguagem de transformação entre modelos especificada a partir de um metamodelo e de uma sintaxe textual concreta. Um programa especificado em ATL é composto de regras (ATL *rules*) que definem como os elementos de um modelo de entrada são alcançados e navegados de forma a se criar e inicializar elementos de um modelo de saída.

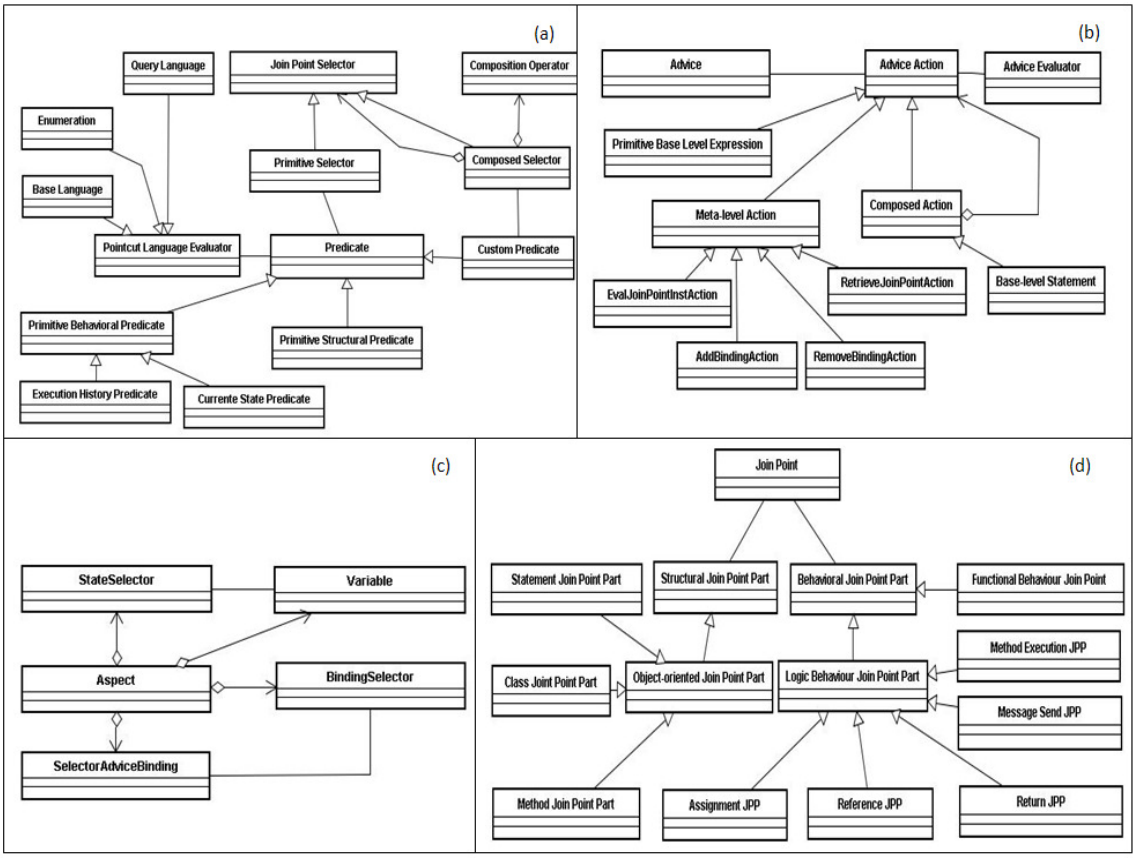

Figura 2. Metamodelos que compõe o Metaspin

3. Visão Geral da Abordagem

O processo de transformação (Figura 3) é organizado em três fases: (i) elaboração e verificação de modelos; (ii) Mapeamento intermediário entre modelos (aSideML e Metaspin); e (iii) Mapeamento entre modelos (Metaspin e AspectLua) e Geração de código em AspectLua. Adicionalmente, existe a fase denominada de preparação, onde são definidas as regras de transformação ATL entre as linguagens.

Para a implementação dos modelos e regras de transformação foi utilizado o ambiente Eclipse [Eclipse, 2009] com o auxílio do EMF (*Eclipse Modeling Framework*) [EMF, 2009]. O EMF é um *framework* que apóia a construção de ferramentas de modelagem, a partir de um modelo estruturado. A descrição dos modelos de aSideML, Metaspin e AspectLua foi utilizado o padrão *Ecore*. Na descrição dos metamodelos de aSideML, Metaspin e AspectLua foi utilizada a linguagem KM3 (*Kernel MetaMetaModel*) especificada em [Jouault, 2006]. Adicionalmente, para se obter a representação textual para modelos aSideML e AspectLua foi utilizado o componente TCS (*Textual Concret Syntax*) [Jouault, 2006]. Por fim, utilizamos a linguagem de transformação ATL (*ATLAS Transformation Language*) [ATL, 2006] para a especificação das regras de transformação entre modelos.

A entrada do processo de transformação entre aSideML e Metaspin é uma representação textual de um modelo aSideML. Esse modelo textual pode ser obtido de

duas formas: (i) manualmente; e (ii) automaticamente, através da transformação entre AOV-Graph [Silva,2006] e aSideML [Chavez, 2004], realizada pela ferramenta MaRiSA-MDD [Medeiros, 2008]. Quando o modelo textual aSideML é obtido manualmente, ele deve ser verificado de forma que somente os modelos bem-formados sejam utilizados no processo de transformação. Em contrapartida, quando o modelo é obtido automaticamente, assume-se que ele seja bem-formado, uma vez que foi obtido de um processo de transformação que possui uma verificação semântica rigorosa. Essa verificação é necessária para garantir que nenhum erro foi introduzido por falha humana. A Figura 3 ilustra o processo de transformação empregado neste artigo. Na Fase 1 é feita a especificação do sistema em aSideML e por conseguinte, essa especificação é verificada de acordo com a sintaxe textual especificada por uma BNF (*Bakus Naur Form*) de aSideML, de forma a validar se o modelo é bem formado.

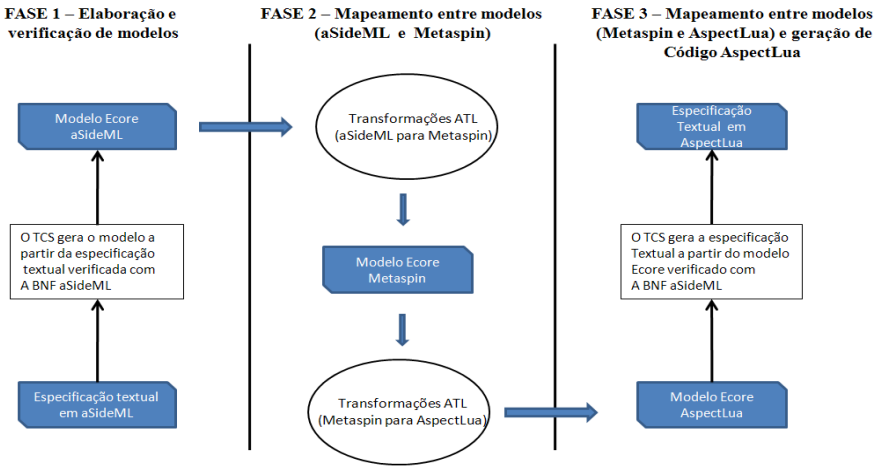

Figura 3.Processo de Transformação de aSideML, Metaspin e AspectLua.

Na fase 2 ocorre o mapeamento intermediário, onde é realizada a transformação entre os modelos de aSideML e Metaspin, bem como a verificação de acordo com os respectivos metamodelos. O modelo textual aSideML é transformado para o modelo Ecore de aSideML, onde esse é verificado conforme o metamodelo aSideML, anteriormente definido em KM3. Por fim, na fase 3 é realizada a transformação do modelo Metaspin para o modelo de uma linguagem de POA [Kiczales et al, 97] onde se deseje gerar código fonte OA. Como Metaspin é uma estratégia genérica, poderia ser gerado código para qualquer linguagem de POA. Nesse trabalho é gerado código fonte em AspectLua.

3.1. aSideML

aSideML [Chavez, 2004] é uma linguagem de modelagem construída para especificar e comunicar projetos OA. A linguagem aSideML tem como base o metamodelo aSideML, que utiliza o metamodelo de UML (*Unified Modeling Language*) como base e provê extensões (novas metaclasses e metassociações) para descrição de aspectos, características transversais e outros elementos de DSOA. Na linguagem os aspectos e características transversais são explicitamente tratados como elementos de primeira classe. Esses modelos servem como planos preliminares que devem ser desenvolvidos

na direção dos modelos de implementação de ferramentas e linguagens de POA [Chavez, 2004].

As principais características de 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.

Em aSideML, um *Template Parameter* define as operações que irão substituir algum comportamento de algum componente do código base, ou seja, corresponde a ser executada quando um determinado *join point* pertencente a um determinado *pointcut*. Adicionalmente, aSideML define o *Template Match* como sendo um conjunto de operações que serão substituídas pelas operações dos aspectos. Essas operações dizem respeito às operações do código base que são "observadas" e tratadas por um determinado aspecto.

3.2. AspectLua

AspectLua [Cacho et al, 2005] é uma extensão da linguagem de programação Lua [Ierusalimschy, 2006] para dar suporte a POA. AspectLua foi desenvolvida utilizando mecanismos básicos da própria linguagem sem a necessidade de alteração do seu interpretador e provê suporte à criação de aspectos, *pointcuts*, *join points* e *advices*.

```
1. a = Aspect:new()
2. a : aspect ({name = 'valor'},
3. {pointcutname = 'valor', designator ='valor', list = {'valor'},
4. {type = 'valor', action = 'valor'})
```

Figura 4: Código genérico para a criação de aspecto.

Para se criar um aspecto (Figura 4) usa-se a função *new ()* (linha 1) que cria uma instância da classe AspectLua. A criação de aspectos em AspectLua depende de três parâmetros: (i) O primeiro parâmetro do método define o nome do aspecto (linha 2); (ii) O segundo parâmetro define os elementos de *pointcut* (linha 2): seu nome, seu *designator* e as funções e variáveis que devem ser interceptadas. O *designator* define o tipo de pointcuts. AspectLua suporta os seguintes tipos: *call* para a chamadas a função; *callone* para a chamada de aspectos que devem ser executados somente uma vez; *introduction* para introduzir funções em tabelas (objetos em Lua); e métodos *get* e *set* aplicado sobre variáveis. O campo *list* define os *join points* pertencentes ao *pointcut* especificado. Adicionalmente, pode-se fazer utilização *wildcards* na definição desses *join points*; e (iii) o terceiro parâmetro é uma tabela Lua que define os elementos do *advice* (linha 3): o tipo (*after*, *before*, e *around*) e a ação que deve ser tomada quando um *pointcut* é atingido.

4. Regras de mapeamento

O processo de mapeamento de aSideML para código em AspectLua é realizado em três fases: (i) mapeamento dos conceitos relacionados a descrição aspectual de aSideML para o Metaspin; e (ii) mapeamento dos conceitos relacionados a descrição dos

elementos base de aSideML para o metamodelo *CoreElement*; (iii) mapeamento dos elementos do Metaspin e do *CoreElement* para os elementos do metamodelo da linguagem específica OA (AspectLua). O Metaspin provê abstrações especificamente para os conceitos ligados a OA. Dessa forma, com a necessidade de endereçar o mapeamento dos elementos base de uma aplicação representado pelo projeto detalhado em aSideML foi utilizado o metamodelo *CoreElement*. O *CoreElement* é um sub-conjunto do metamodelo *Core* da UML que provê abstrações para a representação de classes, operações, atributos, dentre outros.

4.1. Mapeamento de projeto detalhado para o metamodelo aspectual

A Tabela 1 resume o mapeamento de aSideML para Metaspin. Pela tabela, percebe-se que um aspecto no aSideML é mapeado diretamente para um aspecto no MetaSpin. O conceito de *Template Parameter* é representado no metamodelo Metaspin como um *Join Point Selector.* Os elementos definidos no *join point* do Template match (classes, métodos ou declarações) são mapeadas para um conjunto de join points que irão compor o conjunto de join points na definição do pointcut do aspecto, ou seja, correspondem ao conjunto de assinaturas de *join point* que o aspecto deve estabelecer. As características transversais comportamentais podem ser listadas em diferentes compartimentos da interface transversal (declaração inter-tipos, refinamentos e redefinições). As redefinições (*redefinitions*) e refinamentos (*refinements*) possuem elementos do tipo Adorno e *Operation*. Tais elementos irão compor o *advice* do metamodelo Metaspin. O elemento *Operation* é mapeado para a ação do *advice* (*Advice Action)*, que neste caso é especializado para uma ação de meta-nível (*Metalevel Action*), correspondendo a um elemento do tipo *Advice Action*. Por outro lado, o elemento Adorno é mapeado para um avaliador de advice (*Advice Evaluator)*, visto que cada ação de *advice* deve ter um avaliador relacionado. O avaliador de *advice* geralmente possui valores do tipo *before*, *after* e *around*.

<table>
<tr><th colspan="3">aSideML</th><th colspan="2">Metaspin</th></tr>
<tr><td colspan="3">Aspect</td><td colspan="2">Aspect</td></tr>
<tr><td colspan="3">Template Parameter</td><td colspan="2">Join Point Selector</td></tr>
<tr><td>Template Match</td><td colspan="2">Joinpoint</td><td colspan="2">Joinpoint (conjunto de operações escutadas pelo aspecto).</td></tr>
<tr><td rowspan="2">Crosscutting Interface</td><td rowspan="2">Redefinition / Refinement</td><td>Adorno</td><td colspan="2">Advice Evaluator: corresponde a quando a ação do advice deverá ser executada (before, after, around)</td></tr>
<tr><td>Operation</td><td>Advice Action : corresponde à ação a ser executada em um advice.</td><td>EvalJoinPoinInstructionAction</td></tr>
</table>

Tabela 1. Resumo do mapeamento de aSideML para Metaspin.

4.2. Mapeamento de projeto detalhado para metamodelo base

No mapeamento dos elementos de aSideML que denotam características não-transversais foi utilizado um metamodelo *CoreElement*. A tabela 2 ilustra o resumo do mapeamento dos elementos de aSideML para o *CoreElement*. As características transversais comportamentais de aSideML representam refinamentos ou redefinição de operações. Tais operações são mapeadas para o conceito de *Operation* no *CoreElement* , devendo vir precedidas por um comentário denotando qual o tipo de melhoria que essa função está fornecendo, seja um refinamento (*Refinement*) ou uma redefinição (*Redefinition*). Os *Uses* de aSideML representam as operações do código base da aplicação que o aspecto utiliza, sendo assim mapeadas para o conceito de *Operation* no

CoreElement. Por se tratar de características comportamentais requisitadas de aSideML, as operações devem vir precedidas da notação "*from* aSideML - *Uses*" em forma de comentário, denotando que esta operação pertence ao código base e simplesmente é utilizada pelo aspecto. A abstração de classes em aSideML é mapeada para *Class* no CoreElement, sendo seus os atributos e operações mapeados para os elementos do tipo *Attribute* e *Operation* respectivamente. As classes de aSideML geralmente possui referência para as interface que essa classe implementa. Dessa forma, as interfaces em aSideML são diretamente mapeadas para *Interface* no CoreElement, bem como as assinaturas de suas operações para os elementos do tipo Operation.

aSideML		**CoreElement**
Classe		*Class*
Attribute		Attribute
CrosscuttingInterface	*Redefinitions / Refinements/Uses*	*Operation*: precedida de uma notação especificando qual o tipo de caracteristicas transversal essa operação representa, seja ela comportamental, estrutural ou requisitada.
Interface		*Interface*

Tabela 2. Resumo do mapeamento de aSideML para CoreElement.

4.3. Mapeamento entre Metaspin e AspectLua

Esta seção mostra como os conceitos orientados a aspectos providos no Metaspin são mapeados nos elementos específicos da linguagem AspectLua. A tabela 3 mostra o mapeamento do modelo Metaspin para o modelo AspectLua. O elemento aspecto do Metaspin possui mapeamento direto para AspectLua, herdando o nome e os atributos do aspecto do modelo Metaspin. O elemento *JoinPointSelector* é mapeado para o *pointcut* em AspectLua, visto que ambas as abstrações representam um conjunto de *join points*. O *JoinPointSelector* possui um *PointcutLanguageEvaluator* que define qual o tipo de chamada (*call*, *callone*, etc), sendo esta mapeada para o *designator* do *pointcut* em AspectLua. Os *join points* do Metaspin são mapeados para o conceito de *join point* em AspectLua, os quais são representados pelo campo *list* na declaração do *pointcut* do aspecto. Logo, o conjunto de *join points* do modelo Metaspin são mapeados para a lista de *join points* que compõe o *pointcut* de um aspecto em AspectLua. Por fim, o elemento *AdviceEvaluator* do Metaspin representa "quando" (*before*, *after* e *around*) a ação do *advice* de um aspecto deve ser executado. Dessa forma, ele é mapeado para o *AdviceType*. Adicionalmente, o *advice* possui um *EvalJoinPointAction* que consiste numa especialização da uma ação de meta-nível (*Meta-level Action*) que será executada pelo *advice* do aspecto quando um *pointcut* é atingido. Assim, esse elemento é mapeado para um *AdviceAction*, visto que representa o mesmo conceito.

Metaspin		**AspectLua**
Aspect		*Aspect*
Join Point Selector		*Pointcut*
PointcutLanguageEvaluator		*PoincutExpression*
Joinpoint	Class Join Point Part	*Joinpoint*
	Method Join Point Part	
	Statement Join Point Part	
Advice Evaluator		Advice Type: corresponde a quando a ação do advice deverá ser executada (*before, after, around*)
Eval Join Point Action		*Advice Action* : corresponde à ação a ser executada em um advice.

Tabela 3. Resumo do mapeamento de aSideML para CoreElement.

4.4. Mapeamento entre *CoreElement* e AspectLua

Nesta seção mostramos o mapeamento entre os elementos dos metamodelo *CoreElement* e os elementos de Lua que representam o código base de uma aplicação OA. A tabela 4 ilustra o mapeamento entre o *CoreElement* e AspectLua. Os elementos *Class*, *Attribute* e *Operation* do metamodelo CoreElement possui correspondentes diretos no metamodelo de AspectLua, sendo mapeados para o conceito de *Class*, *Variable* e *Function* respectivamente. No entanto, os elementos *Interface* presentes no CoreElement não possuem mapeamento direto para um elemento correspondente em AspectLua. Dessa forma, para que não haja perda de informações na transformação entre os modelos, o elemento interface do CoreElement tem os seus elementos *Operation* mapeados para *function* em AspectLua, de forma que esses elementos são inseridos na classe que implementa essa interface.

CoreElement	**AspectLua**
Class	Class
Attribute	Variable
Operation: precedida de uma notação especificando qual o tipo de caracteristicas transversal essa operação representa, seja ela comportamental, estrutural ou requisitada.	Function
Interface	Function (com atributo isInherited, sinalizando que foi herdada da interface)

Tabela 4. Resumo do mapeamento de *CoreElement* para AspectLua.

5. MaRiSA-AOCode

5.1. Visão Geral de Ferramenta

MaRiSA-AOCode é uma ferramenta que implementa a estratégia apresentada neste trabalho, contemplando a geração de código para linguagens OA a partir do projeto detalhado . O processo de desenvolvimento de MaRiSA-AOCode inclui a especificação de metamodelos e de um conjunto de regras de mapeamento entre esses modelos. As regras de transformação que foram definidas para realizar o mapeamento do modelo do projeto detalhado em aSideML para um modelo Metapin (responsável por representar as abstrações de Orientação a Aspectos, como aspectos, *advice*, *pointcut*, *join point*) e CoreElement(responsável por modelar os elementos do código base, como classes, operações, atributos), bem como as regras responsáveis por realizar as transformações de CoreElement e Metaspin para AspectLua e Lua, estão disponíveis no endereço http://www.ppgsc.ufrn.br/~everton/marisaaocode.

5.2. Estudo de Caso (*Health Watcher*)

Nesta seção mostramos a aplicação das regras de mapeamento em um estudo de caso tradicionalmente usado em sistemas orientados a aspectos, o sistema *HealthWatcher* (HW) [Soares, 2002]. Esse sistema permite a coleta e gerenciamento de informações e denúncias relacionadas à área de saúde pública. A figura 5 mostra no contexto do HW a propagação das informações realizadas de forma automatizada, partindo do projeto detalhado em aSideML até a geração de código fonte em AspectLua e Lua. A representação do aspecto *Persistence_in_DB* nas três fases do mapeamento é descrito através da: (A) descrição textual em aSideML; (B) modelo genérico OA do Metaspin; e (C) do modelo textual em AspectLua

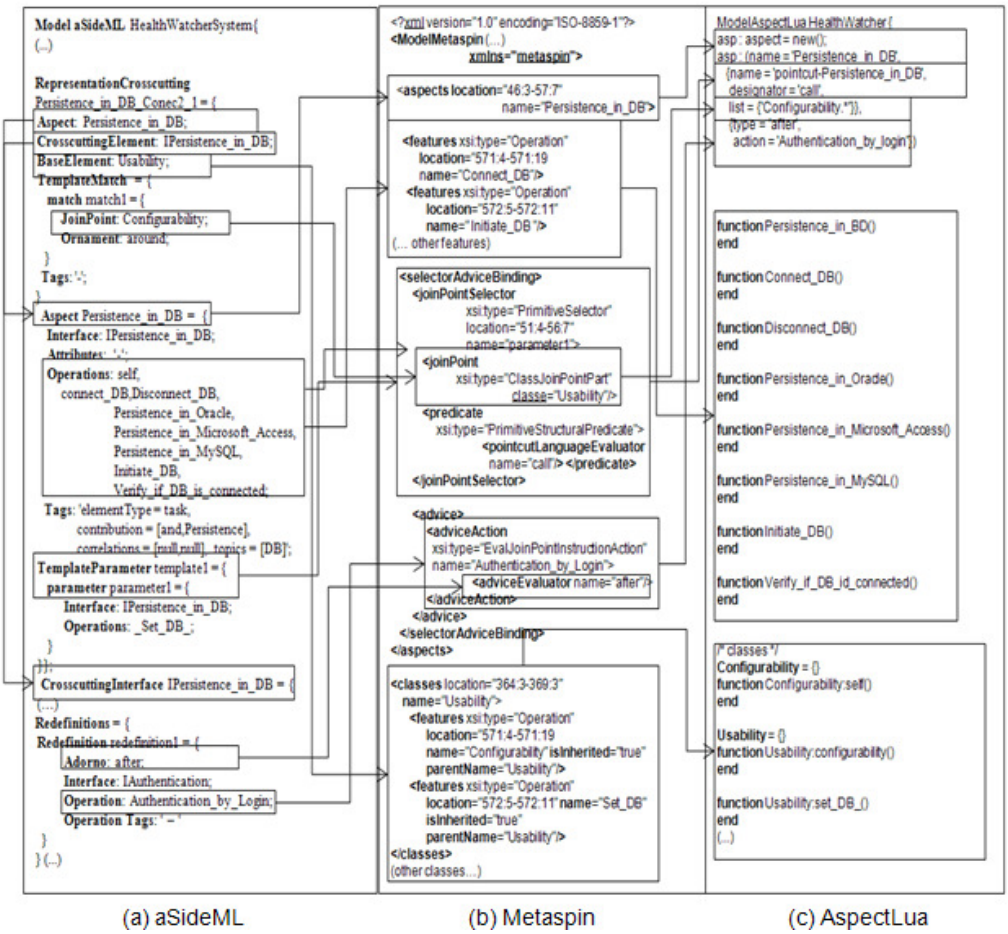

Figura 5. Código do Health Watcher System gerado pela transformação de aSideML para AspectLua utilizando o Metaspin.

Na figura 5(a) podemos visualizar o aspecto sendo constituído por um conjunto de operações (*operations*) que são mapeadas para um conjunto de *features* do tipo *Operation* no modelo Metaspin, sendo posteriormente gerado um elemento *function* em Lua, para cada uma das operações pertencentes ao aspecto. Similarmente, o *TemplateParameter* de aSideML é representado como um *SelectorAdviceBinding* que irá armazenar um conjunto de *join points*. Por sua vez, o *SelectorAdviceBinding* do modelo Metaspin é representado na forma de *pointcut* no modelo de AspectLua.

Na descrição aSideML, o código fonte afetado é representado pela classe *Configurability*, que está representada no elemento *JoinPoint* pertencente ao elemento *TemplateMatch*. Por se tratar de uma classe, esse *join point* é representado como um *ClassJoinPointPart* que consiste numa especialização do elemento *JoinPoint* no modelo Metaspin. Esse *join point* é então mapeado para o parâmetro *list* na declaração do *pointcut* do aspecto, que é responsável por armazenar os *join points* de tal aspecto. Adicionalmente, a classe *Configurability* e seu conjunto de operações são mapeados para a abstração *Class* e *Operation* no *CoreElement*, respectivamente. Por fim, os elementos *Class* e *Operation* são representados como classes (*Class*) e funções (*function*) na linguagem Lua.

Finalmente, a *Redefinition* ilustrada na interface transversal de aSideML, a qual o aspecto *Persistence_in_DB* faz referência, irá compor o *advice* do aspecto. O seu elemento Adorno é representado como um *AdviceEvaluator* no modelo Metaspin, que define o parâmetro type (*after*, *before* ou *around*) do *advice* na declaração do aspecto na linguagem AspectLua. Adicionalmente, o elemento *Operation* da *Redefinition* é mapeado para o elemento *EvalJoinPoinInstructionAction* no modelo Metaspin que corresponde a ação executada no *advice*. Dessa forma, o *EvalJoinPoinInstructionAction* é mapeado para o parâmetro *action* na declaração do *advice*.

6. Trabalhos Relacionados

Vários trabalhos tentam representar os conceitos transversais em modelos e gerar código através da transformação destes modelos. Em [Evermann, 2007] é proposto um meta-modelo capaz de suportar a definição de conceitos transversais em nível de modelo. Este meta-modelo utiliza os conceitos de AspectJ para definir aspectos, *advices* e pointcuts. Aspectos são definidos por meio de classes UML que podem possuir atributos e operações. *Advices* são modelados através de operações marcadas por estereótipos e definidas nos aspectos. Pointcuts são modelados através de atributos marcados por estereótipos que definem os diversos tipos de pointcuts (exe.: call, set). A partir de um modelo válido, a abordagem fornece uma ferramenta que utiliza transformações XSLT para gerar o esqueleto de código em AspectJ.

Similarmente, [Groher e Baumgarth, 2004], [Stein et al, 2002] e [Basch e Sanchez, 2003] descrevem notações que permitem modelar os conceitos transversais em nível de modelo, usando as abstrações fornecidas por AspectJ. Apesar de estas abordagens fornecerem abstrações para modelar e gerar código que respeita a separação entre o comportamento da aplicação e os conceitos transversais, tais soluções diferem da nossa por não serem capazes de abstrair, no nível de modelo, a linguagem de POA. Ou seja, o uso dos conceitos fornecidos por AspectJ para modelar os conceitos transversais restringe significantemente o número de linguagens em que o modelo pode, através de transformações, gerar código.

Alguns trabalhos sugerem que o comportamento da aplicação e seus respectivos conceitos transversais devem ser modelados através de um modelo UML executável. [Mosconi et al, 2008], [Fuentes e Sánchez, 2007] e [Cottenier et al, 2007] definem individualmente vários *profiles* UML que permitem capturar os conceitos transversais em modelos UML executáveis. Os modelos executáveis são geralmente definidos por meio de diagramas de atividades e máquinas de estados que modelam precisamente o comportamento de cada classe do sistema. Nestas abordagens, o comportamento transversal é modelado separadamente e unido ao modelo base da aplicação por meio de um processo de *weaving* de modelo. O processo de weaving resolve os pointcuts definidos e gera, por meio de transformações sucessivas, um modelo final que é independente dos conceitos da OA. Ou seja, os aspectos e advices são transformados em classes e métodos no modelo final, de modo que o código gerado a partir do modelo final também não possui qualquer referência aos conceitos da orientação a aspectos. Desta forma, estas abordagens diferem da nossa por não manterem, no nível de código, a separação entre conceitos transversais e código base da aplicação.

7. Conclusões

Este artigo apresentou MARISA-AOCode, uma estratégia de transformação entre projeto detalhado OA e um metamodelo genérico para linguagens de programação OA. MARISA-AOCode define um processo de transformação que, a partir do modelo aSideML obtém-se um modelo MetaSpin, que é independente de linguagem de POA especifica. A partir do Metaspin, pode-se gerar um esqueleto de código para diferentes linguagens de POA. Nesse trabalho foi gerado código para AspectLua, uma linguagem que segue a mesma filosofia de AspectJ. Esse trabalho está inserido no contexto de um projeto que tem como objetivo fornecer ferramentas baseadas em transformações de modelos que automatizem o mapeamento de requisitos OA para arquitetura AO [Medeiros et al, 2007] e desta para projeto detalhado OA [Medeiros et al, 2009] e, finalmente, para o código orientado a aspectos em várias linguagens de POA. Através desse mapeamento automático pode-se obter código compatível com os requisitos estabelecidos e facilitar o rastreamento entre os artefatos produzidos em diferentes fases do ciclo de vida do software orientado a aspectos. O código obtido através da transformação automática precisa da interferência do programador para adaptar as partes necessárias.

Como trabalhos futuros pretende-se gerar código para outras linguagens OA que não seguem o modelo de AspectJ, de modo a verificar a generalidade das regras de transformação entre o metamodelo Metaspin e as linguagens de POA.

8. Referências

ATL. “ATL: Atlas Transformation Language”. ATL User Manual 0.7, 2006.

Basch, M. e Sanchez, A. “Incorporating aspects into the UML”. In Proceedings of the AOM workshop at AOSD, 2003.

Brichau et al, 2006. ”An Initial Metamodel for Aspect-Oriented Programming Languages”. Technical report of AOSD-Europe, Fevereiro, 2006.

Brichau, J., Haupt, M. “Survey of Aspect-Oriented Languages and Execution Models. Technical report of AOSD-Europe, Maio, 2005.

Cacho N. et al. 2005. ”AspectLua: A Dynamic AOP Approach”. Journal of Universal Computer Science, vol. 11, no. 7, 2005, 1177-1197.

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.

Cottenier, T., van den Berg, A., Elrad, T. “Motorola weaver: Model weaving in a large industrial context”. In Proc. of the 6th Int. Conference on Aspect-Oriented Software Development, Industry Track (AOSD), British Columbia, Canada, 2007.

Eclipse. Disponível em: <http://www.eclipse.org/>. Acesso em: junho de 2009.

EMF. Disponível em: < http://www.eclipse.org/modeling/emf/>. Acesso em: junho de 2009.

Evermann, J. 2007. “A meta-level specification and profile for AspectJ in UML”. In Proceedings of the 10th international Workshop on Aspect-Oriented Modeling (Vancouver, Canada, March 12 - 12, 2007). AOM '07, vol. 209. ACM, New York, NY, 21-27.

Fuentes, L e Sanchez, P. 2007. “Towards executable aspect-oriented UML models”. In Proceedings of the 10th international Workshop on Aspect-Oriented Modeling (Vancouver, Canada, March 12 - 12, 2007). AOM '07, vol. 209. ACM, New York, NY, 28-34.

Filman, R. E. et al. “Aspect-Oriented Software Development”. Boston, Addison Wesley, 2005.

Groher, I. e Baumgarth, T. “Aspect-Orientation from Design to Code”. Munich, Alemanha. Proceedings of the Early Aspects 2004 - Aspect-Oriented Requirements Engineering and Architecture Design. Lancaster, Março, 2004.

Ierusalimschy, R. Programming in Lua. Second Edition, Rio de Janeiro, 2006.

Jouault, F., Bézivin, J., Kurtev, I. “KM3: a DSL for Metamodel Specification”. In: IFIP International Conference on Formal Methods For Open Object-Based Distributed Systems, Bologna, Italy, 2003. Proceedings of 8th IFIP, p. 171-185. Bologna, Italy, 2003.

Jouault, F. e Kurtev, I. “Transforming Models with ATL”. In Proceedings of the Model Transformation in Practice Workshop, October 3rd 2005, part of the MoDELS 2005 conference

Jouault, F, Bézivin, J., Kurtev, I. “TCS: a DSL for the specification of textual concrete syntaxes in model engineering”. GPCE - Generative Programming and Component Engineering: pp. 249-254, 2006.

Kiczales, G. et al. “Aspect-Oriented Programming”. In Proceedings of the European Conference on Object-Oriented Programming, 1997.

Medeiros, A. L, “MARISA-MDD: uma abordagem para transformações entre Modelos Orientados à Aspectos: dos requisitos ao Projeto Detalhado”. Dissertação (Mestrado) – UFRN, 2008.

Mosconi, M., Charfi, A., Svacina, J., Wloka, J. “Applying and evaluating AOM for platform independent behavioral UML models”. In Proceedings of the AOSD Workshop on Aspect-Oriented Modeling (Brussels, Belgium, April 01 - 01, 2008). AOM '08. ACM, New York, NY, 19-24.

Silva, L. “Uma Estratégia Orientada a Aspectos para Modelagem de Requisitos”. Tese de Doutorado em Engenharia de Software - PUC-Rio. Rio de Janeiro, 220p. 2006.

Soares, S. et al. (2002). “Implementing Distribution and Persistence Aspects with AspectJ”. Proc. of the OOPSLA’02, pp. 174-190, 2002.

Stahl, T., Voelter, M., Czarnecki, K. “Model-Driven Software Development, Technology, Engineering, Management”. England: John Wiley & Sons, 2006.

Stein, D., Hanenberg, S., Unland, R. “Designing aspect-oriented crosscutting in UML”. In Proceedings of the AOM with UML workshop at AOSD, 2002.