Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma Linguagem de Aspectos para QVT

Carlos Artur Nascimento Vieira

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande – Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Desenvolvimento Dirigido a Modelos

Professor Dr. Franklin de Souza Ramalho
(Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

V658l Vieira, Carlos Artur Nascimento.
Uma linguagem de aspectos para QVT / Carlos Artur Nascimento Vieira. – Campina Grande, 2016.
94 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2016.
"Orientação: Prof. Dr. Franklin de Souza Ramalho.
Referências.

1. Engenharia de Software. 2. MDA (Model-driven Architecture). 3. Aspectos. 4. *Cross-cutting concerns*. I. Ramalho, Franklin de Souza. II. Titulo.

CDU 004.41(043)

Resumo

MDA, ou Model-driven Architecture, é um padrão definido pelo Object Management Group (OMG) que permite a geração de modelos por outros modelos por meio das Transformações MDA, um conjunto de operações que relacionam elementos de um ou mais modelos de entrada e saída. As transformações podem ser escritas por meio de linguagens como Query/View/Transformation (QVT), outro padrão do OMG.

Como em linguagens de programação, pode-se fazer necessária a adição de funcionalidades que acabam se cruzando no código (como *logging* e rastreio, por exemplo), podendo resultar em *cross-cutting concerns* na transformação. Esse problema pode então ser enfrentado por meio do Paradigma Orientado a Aspectos.

Presentemente, não encontramos soluções que aplicam, especificamente, aspectos para a solução de *cross-cutting concerns* em transformações escritas na linguagem QVT. Assim, nós propomos e desenvolvemos uma linguagem de aspectos para QVT, de nome AQVT, além de um compilador que realiza o weaving do código AQVT com o código original QVT.

Utilizando-se de métricas sobre os programas QVT e a execução das transformações AQVT, realizamos um estudo empírico sobre a linguagem AQVT proposta. Identificamos nesse estudo que ocorre uma melhora fraca na qualidade de leitura e uma boa melhora na modularidade dos código QVT e AQVT, se comparado com uma transformação equivalente escrita puramente em QVT. Contudo, não foi possível realizar uma implementação completa de todas as funções do Paradigma de Aspectos e de alguns elementos da linguagem QVT em AQVT, deixando essas atividades como trabalhos futuros.

Palavras-chave: *MDA, Aspectos, Cross-cutting concerns*

Abstract

MDA (Model-Driven Architecture) is a standard specified by the Object Management Group (OMG) that allows a developer to generate models from models by means of MDA Transformations, a set of functions that match and bind elements between source and target models. Transformations can be written in languages such as Query/View/Transformations Language (QVT), another OMG standard.

As with programming languages, the addition of features in the code that cut through it (like logging and tracing) may cause *cross-cutting concerns* in the transformation. This problem can then be addressed through the use of the Aspect Oriented Paradigm.

Currently, we could not find solutions that used Aspects for the specific problem of cross-cutting concerns within QVT transformations. We proposed and developed an aspects language for QVT, named AQVT, along with a compiler that weaves both the QVT and AQVT codes into a single program.

With the assistance of metrics applied to the QVT programs and the execution of AQVT code, we have performed an empirical research for the proposed AQVT language. We identified that the quality of reading improved slightly and the modularity of the program increased, when comparing the code between two equivalent transformations (one with an AQVT module and the other without it). However, we were not able to completely implement all of the functions from the Aspects Paradigm nor some of the elements from the QVT language into AQVT and propose those activities as our future work.

Keywords: *MDA, Aspects, Cross-cutting concerns*

Agradecimentos

Agradeço inicialmente e sempre a Deus, pela vida e os caminhos que se abriram e se abrirão no futuro. Exponho também meus sinceros agradecimentos aos meus pais, irmão e irmãs por todo o apoio que foi dado não somente durante o período de Mestrado, mas também durante todos os esforços que foram realizados até chegar neste momento. Além disso, em particular, agradeço à minha sobrinha Lara: seu nascimento no ano passado uma bênção especial para toda a familia e que hoje só traz sorrisos e felicidade para todos.

Agradeço profundamente ao meu professor orientador Franklin Ramalho, que sempre mostrou compreensão, paciência e dedicação em todos os momentos desta caminhada, desde os períodos de graduação, com projetos de pesquisa, até este momento de Mestrado. Serei eterno grato pela força, incentivação e, especialmente, às correções e retornos dados, elementos que foram essenciais para este processo e minha formação profissional. Da mesma forma, agradeço também a todos os professores do curso de mestrado: sem o conhecimento dado por cada um deles nunca conseguiria ter chegado neste exato momento.

Obrigado aos meus companheiros e colegas de laboratório, no SPLAB, pelo apoio em tempos de dúvida, o excelente convívio e as comemorações mensais dos aniversariantes. Obrigado a todos os colegas e amigos que sempre me acompanharam nesta caminhada, se fazendo presentes tanto em momentos de trabalho quanto de descontração.

Expresso também minha gratidão à CAPES pelo apoio financeiro que foi dado para a realização deste trabalho.

Conteúdo

Lista de Símbolos

DDM - *Desenvolvimento Dirigido a Modelos*
OMG - *Object Management Group*
MDA - *Model-driven Architecture*
CIM - *Computation Independent Model*
PIM - *Platform Independent Model*
PSM - *Platform Specific Model*
M2M - *Model to Model*
M2T - *Model to Text*
QVT - *Query/View/Transformation*
AQVT - *Aspectos para QVT*
UML - *Unified Modeling Language*
MOF - *MetaObject Facility*
RDBMS - *Relational Database Management System*
RDB - *Relational Database*
GQM - *Goal, Question, Metric*
QP - *Questão de Pesquisa*
LDC - *Linhas de Código*
MLdCR - *Média de Linhas de Código por Regra*
MLdCH - *Média de Linhas de Código por Helper*
MCR - *Média de Condicionais por Regra*
MaxCRR - *Máximo para Chamadas de Regras em Regras*
MaxCHR - *Máximo para CHamada de Helpers em Regras*
MCRR - *Média de Chamadas de Regras em Regras*
MCHR - *Média de Chamadas de Helpers em Regras*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Dentre as diversas metodologias de desenvolvimento de software existentes, o processo de DDM, ou Desenvolvimento Dirigido a Modelos [52], se caracteriza principalmente pelo uso de artefatos representantes tanto do sistema a ser desenvolvido, quanto até mesmo das próprias etapas de desenvolvimento do projeto. Os **modelos,** que fazem parte desses artefatos de DDM, dão maior flexibilidade aos desenvolvedores, permitindo a padronização de problemas e partes do sistema, já que podem ser especificados de forma abstrata ou em baixo nível, sem incluir uma linguagem de programação ou plataforma específica. Já os **metamodelos**, ou modelos de modelos, são outros importantes artefatos de MDD, sob os quais são definidas ou especificadas instâncias (os próprios modelos) que sempre devem estar em conformidade com seu respectivo metamodelo.

Desse modo é possível definir, com DDM, modelos abstratos de um sistema qualquer e, posteriormente, seguir suas especificações para realizar a implementação do projeto em distintas linguagens de programação, como Java [20], C [25] ou Python [46], por exemplo. Um *framework* que utiliza essa metodologia de definição de modelos, liberando o sistema de uma plataforma específica, é MDA (*Model-driven Architecture* (MDA) [32]).

Além de permitir a definição de modelos tanto específicos quanto independentes de plataforma, MDA possibilita o reuso desses artefatos por meio de **transformações entre modelos**, onde um modelo UML [19] [40] pode ser utilizado para a geração de um modelo equivalente em Java, por exemplo. Transformações são, basicamente, compostas por regras de transformação, que casam elementos em modelo(s) de entrada com elementos em modelo(s) de saída, e podem ser escritas em linguagens específicas para transformação, como QVT [39] e ATL [10], e até mesmo linguagens que não foram desenvolvidas com esse propósito, como Java.

Devido a todas essas características, MDA é utilizada em diversas aplicações no mundo todo, sendo aplicada com sucesso até mesmo em projetos de larga escala e de alta importância para suas empresas, como em [29] e [18].

Quando partes de um programa ou sistema, seguindo uma funcionalidade específica, são "atravessadas" no código por outra funcionalidade, ocorre o que chamamos de *cross-cutting*

concerns [26]. Um exemplo disso seria a funcionalidade de tratamento de erros em um sistema, que pode ter seu código encontrado em diversas partes do programa, sem se resumir a um único módulo (como classe, método, procedimento, etc). Assim, com esse tipo de código espalhado, a modularidade do programa acaba sendo prejudicada. O Paradigma Orientado a Aspectos [26] foi criado justamente para permitir a separação desses *concerns* em módulos únicos, aumentando a modularidade do sistema.

O paradigma de Aspectos possibilita a adição de funcionalidades no sistema ou programa sem que seu próprio código seja modificado. Isso é realizado por meio da definição de pontos no código onde "algo acontece" (*Jonpoint* e *Pointcut*), como a instanciação de um objeto, por exemplo, e a execução do código dessa nova funcionalidade (*Advice*) no momento em que o fluxo normal de execução do programa atinge tais pontos.

1.1 Problema

Evolução e manutenção de software são atividades intrínsecas às etapas de desenvolvimento de um software. Nelas, as alterações ou adições de funcionalidades podem ter um impacto como um todo no presente e no futuro do desenvolvimento e, portanto, devem ser sempre realizadas com cautela. Uma mudança em um módulo de conexão à internet em um sistema de análise de dados, por exemplo, pode resultar em bugs ou maiores dificuldades para a implementação de novas funções. A modificação de protocolos de comunicação entre um servidor de base de dados pode originar esse problema, por exemplo. Existem trabalhos, como o de [Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, Mario Piattini] que utilizam modelos e transformações QVT para coletar informações de negócio e rastreio sobre essas mudanças.

Entretanto, da mesma forma que programas podem sofrer com mudanças, refletindo em possíveis impactos no desenvolvimento do software, as transformações também podem sofrer com alto acoplamento sobre suas regras de transformação e os modelos que são utilizados como entrada. No fim, ao invés de diminuir custos para os desenvolvedores, as transformações podem acabar realizando o inverso quando sua complexidade aumenta e as atividades realizadas para sua manutenção tornam-se não-triviais. Ao simplificar essas transformações e mantendo-as livres dessas questões que "cruzam" todo o projeto, esses custos podem ser bastante reduzidos. Essas mesmas questões são reconhecidas como "*cross-cutting concerns*", onde funcionalidades específicas de um programa acabam se cruzando no próprio código (como rastreio, logging, persistência), sem que possam ser agrupadas em um único módulo do programa.

Cross-cutting concerns não afetam apenas projetos que sofrem mudanças, e podem aparecer e prejudicar a modularidade e compreensão de programas que nunca passaram por

modificações, por exemplo. Uma transformação com dezenas de milhares de linhas de código e a aplicação de logging (*cross-cutting concern*) para a sua execução pode poluir o código original com elementos que não são necessários para o seu funcionamento normal. Portanto, pode-se fazer necessário encontrar uma forma de indicar que o "logging" deverá ser realizado no programa sem que isso apareça em seu código.

1.2 Exemplo em QVT

Como um padrão de linguagens de transformação, definido pelo *Object Management Group* (OMG), a linguagem QVT é bastante utilizada em projetos que fazem uso de transformações MDA, como o próprio Eclipse QVTo [15], o Palladio [41], o Sofa Project [51] [6] e o Q-ImPrESS Project [42]. Programas escritos nessa linguagem podem, como quaisquer outros programas de outras linguagens de programação, sofrer mudanças durante o desenvolvimento do sistema e, também, serem afetados por *cross-cutting concerns* (como *logging*, execução de múltiplas regras, etc). Tais problemas podem afetar a legibilidade e compreensão do código, além da modularidade dos programas. E uma forma de se resolver os *cross-cutting concerns* é por meio do Paradigma de Aspectos. Utilizando-se de Aspectos, é possível separar *concerns* em um único local no código, facilitando sua modularidade e compreensão.

Um exemplo de como *cross-cutting concerns* podem ocorrer em transformações e como eles podem ser resolvidos apresentado no Código 1.1.

O Código 1.1 (retirado de [50]) representa uma simples transformação QVT no estilo "Hello World!". Seu objetivo é receber como entrada um modelo qualquer ABC e gerar outro modelo ABC com a adição do texto " World!" em um dos elementos no modelo de saída. Para isso, a transformação se utiliza de três regras de transformação: *main* (ponto de entrada para o programa), *Root2Root* e *A2B*. Um *cross-cutting concern* que poderia ocorrer nesse programa seria o rastreio [23] [47] [54] da execução das regras citadas, por exemplo. Para isso, seria necessária a introdução de código de rastreio nas regras *Root2Root* e *A2B,* de modo que cada execução da regra possa gerar dados de rastreio (em um modelo extra de saída, por exemplo). Entretanto, para uma transformação em constante desenvolvimento ou de tamanho grande, com dezenas de regras diferentes e centenas ou milhares de linhas de código, essa atividade pode não se mostrar trivial (além de prejudicar a compreensão e a modularidade do código, por exemplo). Aspectos, então, seriam uma forma de resolver esse *concern*, com a adição de apenas algumas linhas de código em um novo módulo ou arquivo, realizando o rastreio de todas as regras da transformação sem que o

programador modifique o código original. Um pseudo-código de como isso poderia ser realizado para a transformação exemplo é dado abaixo, no Código 1.2.

```
1  modeltype ABC uses ABC('http:///ABC.ecore');
2
3  transformation HelloWorld(in source:ABC, out target:ABC);
4
5  main() {
6      source.rootObjects()[Root]->map Root2Root();
7  }
8
9  mapping Root :: Root2Root() : Root {
10     element += self.element->select(a |
11         a.oclIsKindOf(A))[A]->map A2B();
12 }
13
14 mapping A :: A2B() : B
15     when {
16         self.id > 0
17     }
18     {
19     result.id := self.id;
20     result.b := self.a + " World!";
21 }
22
```

Código 1.1. Transformação “Hello World!” (retirado de [50]) em QVT.

```
1. Aspecto Rastreio{
2. Para cada Regra em HelloWorld faça:
3.   gere um dado por execução
4. }
```

Código 1.2. Pseudo-código de Aspectos.

Outros exemplos de *cross-cutting concerns* que poderiam ocorrer em transformações QVT são: *logging*, verificação de erros, checagem de elementos de modelos, etc. Entretanto, após análise, não encontramos suporte de Aspectos para, especificamente, a linguagem de transformação QVT (ou até outras linguagens de transformação).

1.3 Objetivos

O **objetivo geral** deste trabalho de dissertação é propor uma abordagem de Aspectos para QVT por meio de uma nova linguagem, auxiliando assim na resolução de eventuais *cross-cutting concerns* que podem ocorrer em transformações.

Como **objetivos específicos** temos a criação de um compilador ou *weaver* de Aspectos para a linguagem QVT, juntamente com especificação linguagem de Aspectos para QVT (AQVT). A linguagem deverá contemplar elementos do Paradigma de Aspectos como Joinpoint, Pointcut e Advice, além de fazer uso de elementos de QVT, como Helpers, Regras, Bindings, entre outros. Além disso, o *weaving* realizado pelo compilador deverá incluir no código final do programa QVT as funcionalidades e operações especificadas no programa AQVT de forma corretamente.

1.4 Escopo

Definimos o escopo deste trabalho para Transformações M2M realizadas por meio da linguagem de transformação QVT-Operational, com a utilização dos seus seguintes elementos: regras (definidas em QVT-O por meio da palavra-chave *mapping*), *helpers* (definidos tanto pela palavra-chave *helper* quanto por *query*), objetos, *bindings* (conjunto de elemento de entrada e elemento de saída), elementos de entrada e de saída.

Já com o Paradigma de Aspectos, utilizamos as definições de *After* e *Before* para a execução de *Advices*. Entretanto, o *Around* não foi incluído no escopo por questões de complexidade e tempo de desenvolvimento, já que as mudanças aplicadas no código QVT por meio do *Weaver* necessitariam de uma quantidade bem maior de tratamentos no código do compilador. Especificamos para AQVT os seguintes *Joinpoints* relacionados ao contexto de QVT-O: chamada de regras em regras, execução de construtor, execução de regras, acesso a elemento no modelo de entrada ou saída, inicialização de objetos e geração de bindings.

Além disso, a linguagem de aspectos para QVT foi concebida inspirada em AspectJ, já que esta possui os principais elementos do Paradigma de Aspectos e é uma das mais conhecidas e utilizadas linguagems dessa área.

1.5 Relevância

Com o foco em desenvolvedores que utilizam MDA e QVT em seus projetos, esperamos que o trabalho contribua com a solução de *cross-cutting concerns* e a melhora na qualidade de leitura, compreensão e modularidade do código das transformações. Para essa área, contribuímos com a especificação de uma linguagem inicial de Aspectos para QVT, além do desenvolvimento de um compilador próprio para a linguagem.

1.6 Estrutura do Documento

Este trabalho de dissertação se encontra organizado da seguinte forma: o **Capítulo 2**, de Fundamentação Teórica, introduz e expande os conceitos de DDM e Transformações de Modelos (**2.1**), QVT (**2.2**), QVT-Operational (**2.2.1**) e Programação Orientada a Aspectos (**2.3**). O **Capítulo 3** apresenta a linguagem de Aspectos para QVT, AQVT, com a estrutura geral da linguagem e seus elementos em Joinpoints e Pointcuts (**3.1**), Advice (**3.2**), Exemplos de Uso (**3.3**) e o *Weaving* (**3.4**) de AQVT, ou a forma como seu compilador faz a aplicação das funcionalidades da linguagem de aspectos para QVT. A avaliação da linguagem AQVT é apresentada no **Capítulo 4**, juntamente com as conclusões obtidas. O **Capítulo 5** apresenta os trabalhos relacionados e, por fim, as considerações finais são dadas no **Capítulo 6**.

Capítulo 2

Fundamentação Teórica

Este capítulo fará a introdução e explicará os conceitos utilizados durante toda esta dissertação, como Transformações, QVT, e Aspectos.

2.1 DDM e Transformações de Modelos

Desenvolvimento Dirigido por Modelos (DDM) [52] é uma metodologia de desenvolvimento de software que foca na especificação de artefatos (modelos), representando as mais variadas partes de um sistema ou suas funcionalidades. Tais representações podem ser definidas tanto de forma específica quanto de forma abstrata, sendo independentes ou não de plataforma ou linguagem de programação. Desse modo, os desenvolvedores possuem mais flexibilidade em suas decisões de projeto, podendo padronizar a representação de cada problema a ser resolvido. Além disso, esses artefatos podem ser reutilizados por aplicações com tecnologias diferentes, possibilitando tanto a diminuição do esforço gasto com a produção de novas aplicações, como também o relacionamento entre diferentes sistemas. *Model-driven Architecture* (MDA) [32] é um padrão, definido pelo *Object Management Group* (OMG), e um *framework* de design de software que utiliza a abordagem DDM no desenvolvimento de um sistema.

Baseada especialmente na *Unified Modeling Language* (UML) [19] [40] e em outros padrões da indústria para visualização, armazenamento e intercâmbio de projetos de software e modelos, MDA define quatro tipos diferentes de modelos: independente de computação (CIM), independente de plataforma (PIM), específico de plataforma (PSM) e código. CIMs são representações do contexto de negócio e dos requisitos do sistema. PIMs são modelos que não dependem da plataforma ou linguagem do sistema que estão modelando, enquanto que PSMs representam o sistema definido de acordo com sua plataforma específica. Já o código é o próprio código fonte do software em questão. Com o uso desses diversos tipos de artefatos, MDA permite, principalmente, o reuso de modelos e sua utilização em aplicações com tecnologias diferentes.

Para realizar isso, pode ocorrer a necessidade de se gerar modelos (ou até mesmo código fonte) a partir de outros modelos: atividades definidas como Transformações MDA.

Transformações MDA são operações nas quais é possível, a partir de um modelo de entrada, gerar outro modelo de saída e/ou código em linguagem de programação específica, utilizando-se de um conjunto de regras de transformações. Além disso, para se realizar uma transformação, é necessário que os modelos utilizados no programa estejam de acordo com o seu metamodelo (modelo do modelo), que dá uma representação simplificada dos modelos da transformação.

Metamodelos são artefatos que definem como modelos deve ser construídos. Toda a estrutura de um modelo, suas semânticas e restrições deverão sempre estar em conformidade com seu respectivo metamodelo [32]. De acordo com [35], um metamodelo é um modelo de uma linguagem de modelagem, ou seja, ele especifica uma sintaxe abstrata para essa linguagem. Um exemplo básico seria o metamodelo de UML, que define como todos os elementos dessa linguagem de modelagem se relacionam e como devem ser utilizados.

As Transformações são divididas em dois grupos distintos: Modelo para Modelo (M2M) e Modelo para Texto (M2T). Transformações M2M recebem modelo(s) e metamodelo(s) como entrada em sua execução e, a partir de regras de transformação específicas, geram modelo(s) de saída. Já as Transformações M2T utilizam como entrada modelo(s) e metamodelo(s) e geram código fonte em linguagem definida de acordo com seu metamodelo e as regras de transformação definidas no programa. Um exemplo básico de utilizaçãp desses dois tipos de transformações seria a transformação de um modelo UML para código fonte Java, visto na Figura 2.1: uma transformação M2M receberia como entrada o modelo UML, seu metamodelo (UML) e o metamodelo de Java para, na saída, gerar um modelo Java. Em seguida, uma transformação M2T receberia como entrada o modelo Java que havia sido gerado (além do metamodelo Java, novamente) e, no fim da sua execução, geraria o código Java equivalente ao modelo inicial UML.

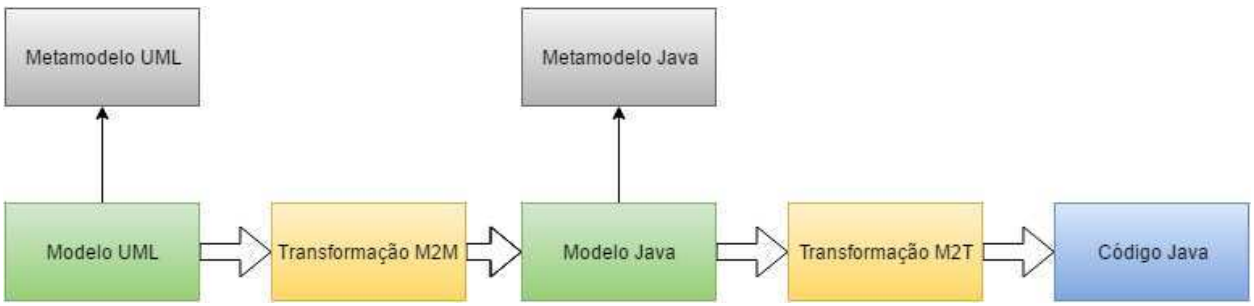

Figura 2.1. Exemplo de transformações M2M e M2T.

Programas que permitem a transformação entre modelos podem ser escritos em linguagens de transformação como QVT [39] e ATL [10], onde a primeira é um padrão definido pelo *Object Management Group* (OMG) e a última é uma variação desse padrão, criada pela *Eclipse Foundation*. Para Transformações de Modelo para Texto, além da própria linguagem ATL, existem as linguagens MOF to Text [37] e MOFScript [13] e o *framework* Epsilon [12], por exemplo. A estrutura dos programas de transformação é baseada em **regras**, que casam elementos de um ou mais modelos fonte para um ou mais modelos de saída, e **helpers**, que computam valores para serem utilizados em regras.

2.2 QVT

Query/View/Transformation (QVT) é um padrão de linguagens de transformação definido pelo OMG. Ele é formado basicamente por três linguagens que definem transformações unidirecionais, bidirecionais e alvos de transformação, respectivamente: *QVT-Operational*, *QVT-Relations* e *QVT-Core*.

QVT-Operational, ou *Operational Mappings*, é uma linguagem imperativa para a criação de transformações unidirecionais (ou seja, o modelo de saída não é constantemente sincronizado com o modelo de entrada) e que provê extensões em OCL (Object Constraint Language) [38]. Já *QVT-Relations* é uma linguagem declarativa, que permite a criação de transformações tanto uni quanto bidirecionais (onde tanto o modelo de saída quanto de entrada podem ser modificados por meio das regras de transformação). Por último, *QVT-Core* é uma linguagem simples, de nível mais baixo do que as linguagens anteriores, que define transformações de modelos de *Relations* para *Core*, mas não é usada diretamente pelo desenvolvedor. De acordo com o próprio OMG [34], pode ser feita uma analogia para as duas últimas linguagens, comparando-as com a linguagem Java e seu Java Bytecode, respectivamente. Uma transformação de *Relations* para *Core* seria como uma especificação do Compilador Java, produzindo *Bytecode* (ou modelo *Core*). Todas as três linguagens de QVT podem ser estendidas com a implementação de operações relacionadas à *MetaObject Facility,* ou MOF [36], que é um padrão para metamodelos de DDM também especificado pelo OMG.

Uma visão da arquitetura QVT é apresentada na Figura 2.2, extraída de [34].

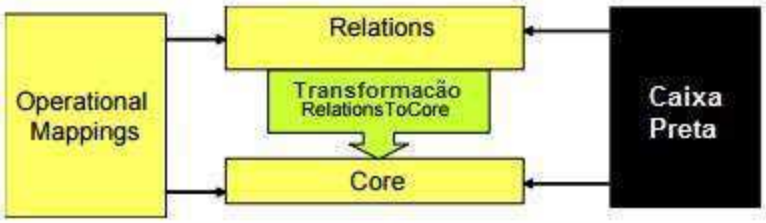

Figura 2.2. Arquitetura QVT. Extraída de [34].

A *Caixa Preta* pode ser uma ou várias implementações de operações MOF, realizadas em linguagens de programação que possuem *binding* com MOF, permitindo que operações complexas (ou que não possuem suporte em MOF) possam ser realizadas na transformação.

O Código 2.1 exemplifica uma transformação, *Book2Publication* (extraída de [34]), escrita na linguagem QVT Operational, que recebe como entrada modelos de livros (*Book*) e os transforma em modelos de publicações (*Publication*) em sua saída. A definição inicial da transformação e seus modelos de entrada (*bookModel*, instância do metamodelo *BOOK*) e saída (*pubModel*, instância do metamodelo *PUB*) são apresentados na Linha 1. O ponto de início de execução da transformação é seu "main()" (Linhas 3-5), que executa o *mapping* ou regra "book_to_publication" (Linhas 7-10) para objetos do tipo *Book* no modelo de entrada. Essa regra mapeia elementos de título do livro para elementos de título de publicação (Linha 8), além de incluir a quantidade de páginas do livro no modelo de publicação (Linha 9).

```
1  transformation Book2Publication(in bookModel:BOOK,out pubModel:PUB);
2
3  main() {
4    bookModel->objectsOfType(Book)->map book_to_publication();
5  }
6
7  mapping Class::book_to_publication () : Publication {
8    title := self.title;
9    nbPages := self.chapters->nbPages->sum();
10 }
```

Código 2.1. Transformação QVT de modelos "Book" para modelos "Publication". Extraída de [34].

Os metamodelos referentes a *Book* e a *Publication* são representados na Figura 2.3.

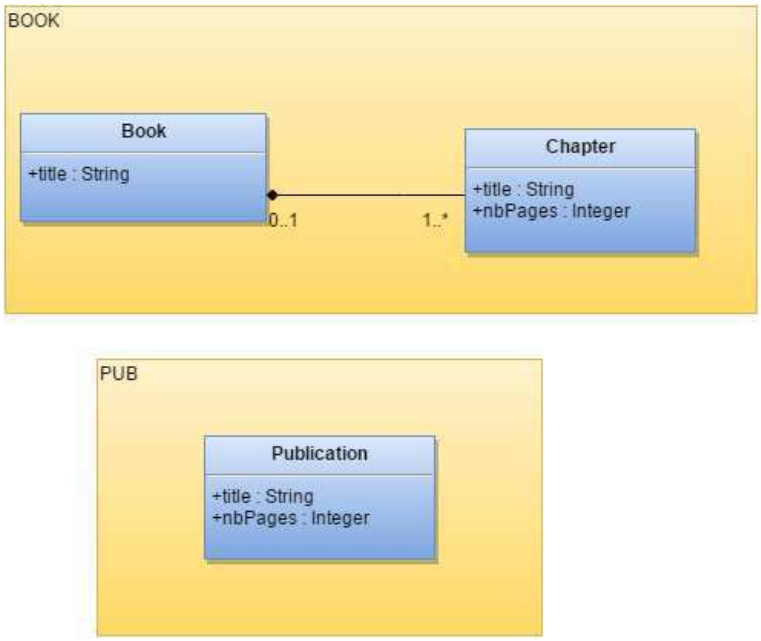

Figura 2.3. Metamodelos *Book* e *Pub* (adaptado de [34]).

Já um exemplo da linguagem QVT Relations pode ser visto no Código 2.2 (adaptado de [34]), que mostra um trecho de uma transformação escrita nessa linguagem. O objetivo desse programa é gerar um modelo de banco de dados relacional (RDBMS) a partir de um modelo UML. Na Linha 1, a transformação é identificada (por meio da palavra-chave *transformation*) e seus modelos e metamodelos definidos (respectivamente, *uml* e *rdbms*, *SimpleUML* e *SimpleRDBMS*). É então definida na transformação uma relação (especificada pela palavra-chave *relation*), ou "regra" de transformação (Linha 3), que irá mapear cada *Package* do modelo UML para um *Schema* em RDBMS (Linhas 7-8). Além disso, na Linha 5 é especificado um atributo *pn*, do tipo String, que será utilizado para representar os elementos *name* de *Package* e *Schema*.

```
1  transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS)
2  {
3  top relation PackageToSchema
4  {
5     pn: String;
6
7     checkonly domain uml p:Package {name=pn};
8     enforce domain rdbms s:Schema {name=pn};
9  }
```

Código 2.2. Trecho de código de transformação QVT-Relations (retirado de [34]).

Como mencionado anteriormente, a linguagem QVT é definida especificamente para transformações de modelo a modelo, não permitindo a realização de transformações textuais, onde código-fonte é gerado a partir de modelos dados como entrada, ou modelos são gerados a partir de código-fonte. Apesar dessa limitação, a sua definição da OMG permite que a linguagem seja estendida para o propósito de transformações textuais, como o que ocorreu com implementações como a de MofScript.

2.2.1 QVT-Operational

Com uma estrutura similar à de outras linguagens de transformação bastante utilizadas, como ATL, e sendo uma das linguagens QVT que mais possui suporte atualmente (Eclipse QVTo [14] e SmartQVT [22], por exemplo), QVT-Operational foi a linguagem utilizada como base para este trabalho. O Código 2.3 (extraído de [34]) a seguir exemplifica a estrutura de uma transformação escrita nessa linguagem e alguns de seus elementos e conceitos.

```
1  modeltype UML uses 'http://www.eclipse.org/qvt/1.0.0/Operational/examples/simpleuml';
2  modeltype RDB uses 'http://www.eclipse.org/qvt/1.0.0/Operational/examples/rdb';
3
4  transformation Simpleuml_To_Rdb(in uml : UML, out RDB);
5
6
7  main() {
8      uml.rootObjects()[UML::Model]->map model2RDBModel();
9  }
10
11
12 mapping UML::Model::model2RDBModel() : RDB::Model {
13     name := self.name;
14     schemas := self.map package2schemas()->asOrderedSet();
15 }
16
17 mapping UML::Property::primitiveAttribute2column(in targetType: UML::DataType) : RDB::TableColumn
18     when { self.isPrimitive() }
19 {
20     isPrimaryKey := self.isPrimaryKey();
21     name := self.name;
22     type := object RDB::datatypes::PrimitiveDataType { name := umlPrimitive2rdbPrimitive(self.type.name); };
23 }
24
25 helper umlPrimitive2rdbPrimitive(in name : String) : String {
26     return if name = 'String' then 'varchar' else
27         if name = 'Boolean' then 'int' else
28             if name = 'Integer' then 'int' else
29                 name
30             endif
31         endif
32     endif
33 }
```

Código 2.3. Trecho de código da transformação QVT-O *SimpleUML2RDB* (retirado de [34]).

Nas linhas 1-2, temos os tipos de modelos (ou metamodelos) utilizados na transformação, representados pelos símbolos *UML* e *RDB* e indicados por meio da palavra-chave *modeltype*. Em seguida, na linha 4, temos a assinatura da transformação, identificada por *Simpleuml_To_Rdb*, juntamente com a especificação do tipo de modelo que deverá ser dado como entrada (*UML*) e o tipo de modelo que deverá ser gerado na saída (*RDB*).

As linhas 7-9 indicam o ponto de entrada da transformação, ou o local por onde a execução de toda a transformação deverá começar, representado pela palavra-chave *main()*. Nesse ponto de entrada, especificamente na linha 8, pode-se observar a existência de um **Filtro** (*UML::Model*) aplicado sobre os objetos-raiz do modelo de entrada (*uml.rootObjects()*). Todo Filtro é especificado por meio de chaves ("[ ]") após coleções ou outros elementos do modelo que podem ser filtrados para elementos específicos. A função de um Filtro é coletar e retornar, de um elemento no modelo, um conjunto específico de elementos (no caso, todos os elementos *Model* de *UML*). Para todos os elementos obtidos com esse Filtro, é aplicada a regra *model2RDBModel*, como pode ser identificado pela utilização de "-> *map model2RDBModel*". A palavra-chave "*map*" é sempre seguida de uma referência para uma regra, indicando que ela deverá ser executada naquele momento na transformação.

Duas das regras da transformação, identificadas pela palavra-chave *mapping*, podem ser identificadas nas linhas 12-15 (*model2RDBModel*) e linhas 17-23 (*primitiveAttribute2column*).

A primeira regra, *model2RDBModel*, se encarrega da geração, no modelo de saída RDB, de elementos *Model RDB* a partir de elementos *Model UML*. É possível notar nas linhas 13 e 14, por exemplo, a existência de ***Bindings***, ou o conjuntos de valores que são casados no modelo de entrada e de saída (no caso, o atributo "name" de um *Model UML* com o atributo "name" de um *Model RDB,* e o atributo de saída "schemas" com um *OrderedSet* gerado por outra regra de transformação). Com a utilização da palavra-chave *self*, é possível acessar o elemento que é utilizado como entrada na regra, ou um *Model UML* no exemplo. Em QVT, regras podem executar outras regras, o que pode ser visto na linha 14: uma outra regra, *package2Schemas* (que é especificada no código completo do programa [Anexo A]), é executada por esta regra para a geração no modelo de saída de *Schemas RDB* a partir de *Packages UML* do modelo de entrada.

Já na segunda regra, *primitiveAttribute2column*, especificamente na linha 22, ocorre a inicialização de um **Objeto** (*PrimitiveDataType* no metamodelo RDB), uma estrutura de dados em QVT-O que pode ser utilizada em regras para gerar elementos completos no modelo de saída, definida por meio da palavra-chave *object*. Na mesma linha, é possível notar a utilização de um **Helper**, *umlPrimitive2rdbPrimitive* (linhas 25-33), para a obtenção do nome de um valor primitivo correspondente no modelo de saída RDB. Segundo [34], *Helpers* são operações realizadas sobre um ou mais objetos-fonte para prover um resultado. No código exemplificado, o *Helper* recebe uma *String* como entrada e retorna outra *String* em sua saída, com a verificação de diversas

expressões booleanas, permitidas no corpo do *Helper*. Todo o corpo da regra só será executado, entretanto, se a condição definida pela palavra-chave "*when*" for satisfeita (no caso, a *Property UML* deverá ser um primitivo, identificado pelo *Helper* "*isPrimitive*").

Além disso, existem **Construtores** (indicados pela palavra-chave *constructor*), que podem ser utilizados para definir e instanciar, diretamente na transformação, tipos dos metamodelos de entrada e saída. Tais estruturas podem ser vistas destacadas em vermelho no trecho do Código 2.4, modificado de [34].

```
mapping UML::DataType::dataType2primaryKeyColumns(in prefix : String,
        in leaveIsPrimaryKey : Boolean, in targetType : UML::DataType) : OrderedSet(RDB::TableColumn) {
    init {
        result := self.map dataType2columns(self)->select(isPrimaryKey)->
            collect(c | object RDB::TableColumn {
                name := prefix + '_' + c.name;
                domain := c.domain;
                type := new RDB::datatypes::PrimitiveDataType(c.type.name);
                isPrimaryKey := leaveIsPrimaryKey
            })->asOrderedSet();
    }
}

constructor RDB::datatypes::PrimitiveDataType::PrimitiveDataType(n : String) {
    name := n;
}
```

Código 2.4. Trecho de código com Construtores QVT-O. Modificado de [34].

No Código 2.4, pode-se notar que o corpo da regra foi definido especificado com a palavra-chave "*init*", que permite o retorno de coleções mutáveis em uma regra (OrderedSet, no exemplo), auxiliando na geração e modificação de novas coleções por outras regras.

Na execução de uma transformação QVT-O, o fluxo sempre inicia-se por meio do "main", executando-se então todas as regras que são chamadas nesse ponto de entrada. Caso uma regra não seja chamada por esse ponto ou por outras regras, ela não é executada e os elementos que ela poderia gerar para o modelo de saída não são gerados, algo que ocorre diferentemente em transformações QVT-R, onde regras ou "relations" definidas como *Top-level* são sempre executadas no programa, independentemente de sua chamada ou invocação por outras regras.

2.3 Programação Orientada a Aspectos

De acordo com [26], um "aspecto" é uma propriedade ou característica de um sistema que não pode ser claramente encapsulada em um único procedimento generalizado. Quando essa propriedade cruza com diversas outras abstrações de um programa (como métodos, classes, etc),

ocorre um *cross-cutting concern*. O Paradigma Orientado a Aspectos tem como objetivo facilitar a modularidade de sistemas, ao separar "aspectos" em partes coerentes do programa, encapsulando os *cross-cutting concerns*.

Um *Join Point Model* é um tipo de modelo que faz parte de linguagens orientadas a aspectos, definindo características para um *Join Point*, ou pontos em um programa onde um comportamento pode ser adicionado. Essas características são: quando um aspecto deve ser executado, como um *Join Point* deve ser especificado e como executar código no *Join Point*.

Advice é o código de aspectos que define a funcionalidade do próprio Aspecto, ou seja, seu comportamento no programa. É ele que vai ser executado quando o fluxo de execução atinge seu respectivo *Join Point*. Os *Join Points* são pontos no código onde acontece algo (por exemplo, execução de um método, criação de um objeto, entre outros). Esses pontos são especificados por meio de *Pointcuts*, que indicam exatamente em que ponto da execução o *Advice* deverá ser executado.

O Código 2.5 (extraído de [11]) exemplifica esses componentes em um trecho de programa escrito na linguagem AspectJ [8], aplicado sobre um hipotético sistema com interface gráfica, onde uma tela ou "Display" pode ser atualizada por meio do método "update". O *Pointcut* "set()" casa com a execução de qualquer método que inicia com o nome "set" em um objeto do tipo "Point". Usando o *Pointcut* declarado, o *Advice* indica que "Display.update()" deverá ser executado depois dos *Join Points* especificados (execução de métodos cujos nomes se iniciam com "*set*" na classe *Point*).

```
pointcut set() : execution(* set*(..) ) && this(Point);

after () : set() {
  Display.update();
}
```

Código 2.5. Exemplo de *Join Point/Pointcut* e *Advice* em AspectJ, extraído de [11].

O processo de interligação entre as abstrações do programa, usualmente denominado *Weaving*, pode ser realizado tanto em tempo de execução quanto de *build*, com código adicional ou modificado. A semântica desse processo não precisa ser compreendida pelo programador, apenas a própria linguagem de aspectos é necessária. Um exemplo disso pode ser visto na Figura 2.4 (modificada de [21]), que exemplifica o Weaving do Código 2.5, onde o *Advice* e o método "set()" da classe "Point" são recombinados para formar a funcionalidade desejada (atualizar o "Display" depois de um "Set"). Isso não aparecerá no código fonte, mas terá a semântica garantida pela ferramenta ou compilador de aspectos.

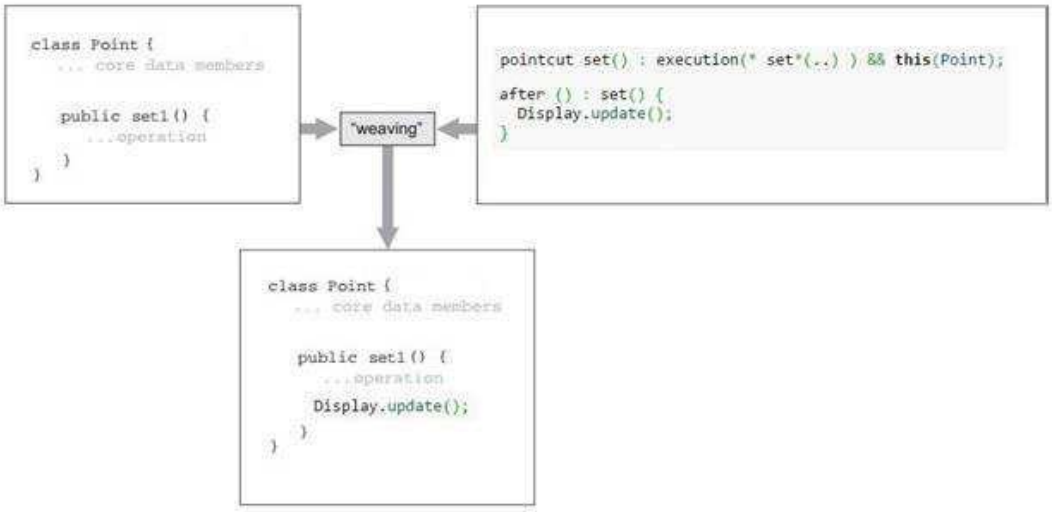

Figura 2.4. *Weaving* de classes separadas para uma operação (Modificada de [21]).

O Código 2.6 (extraído de [11]) mostra um trecho de código exemplificando mais uma característica de AspectJ. O Aspecto *HandleLiveness* deverá verificar, antes de qualquer chamada (indicado pelo símbolo "*") a um método público na classe *Handle* (Linha 2, indicado pela palavra-chave *public*), se o objeto *partner* é *null* ou não está "vivo" (Linha 3), lançando uma exceção caso qualquer uma das duas opções seja verdadeira (linha 4). Nesse caso, *os Joinpoints* do código estão na Linha 2, indicados pelas palavras-chave *target* (quando *Handle* é alvo de algo) e *call* (chamada de métodos públicos).

```
aspect HandleLiveness {
  before(Handle handle): target(handle) && call(public * *(..)) {
    if ( handle.partner == null  || !handle.partner.isAlive() ) {
      throw new DeadPartnerException();
    }
  }
}
```

Código 2.6. Trecho de código AspectJ com aspecto *HandleLiveness*. Extraído de [11].

Em linguagens como AspectJ, pode-se também declarar elementos da linguagem Java (como métodos e construtores, por exemplo) na especificação do Aspecto, ligando-os às classes ou tipos utilizados no Aspecto. Enquanto esses elementos novos são visíveis apenas aos seus respectivos aspectos, é possível definir que classes ou tipos devem implementar interfaces e/ou herdar outros tipos, com a condição de que qualquer método necessário seja definido na classe que foi modificada. Todo esse processo é chamado de *Inter-type Declaration*.

O Código 2.7 mostra exemplos de implementação dessas *Inter-type Declarations*, onde o "aspect A" define uma interface "HasName" em seu campo e declara que qualquer tipo que seja um "Point", "Line" ou "Square" deverá implementar tal interface. Em seguida, é definido que "HasName" possui um atributo "name" do tipo *String* e um método "getName" que retornará esse atributo. Em Aspectos, é possível definir atributos e métodos completos para interfaces.

```
aspect A {
  private interface HasName {}
  declare parents: (Point || Line || Square) implements HasName;

  private String HasName.name;
  public  String HasName.getName()  { return name; }
}
```

Código 2.7. Exemplo de Inter-type Declaration. Extraído de [11].

Capítulo 3

AQVT

Durante o processo de desenvolvimento de um sistema ou projeto, funcionalidades ou conceitos específicos, como segurança, persistência e *logging*, por exemplo, podem ficar espalhados no código do programa, prejudicando sua modularidade e dificultando sua manutenção. Esses problemas, chamados de *cross-cutting concerns*, podem ser resolvidos com o auxílio de Aspectos, que surgiram para facilitar questões de modularidade no sistema e, principalmente, resolver os problemas de *cross-cutting*.

Em QVT, *cross-cutting concerns* podem ocorrer com questões como rastreamento de execução de transformações, verificação de elementos em modelos de entrada, geração de objetos, entre outros. Dessa forma, buscando resolver esses problemas, foi desenvolvida neste trabalho uma linguagem de Aspectos sobre QVT: AQVT (Aspects for QVT).

A estrutura da linguagem AQVT é baseada no Paradigma Orientado a Aspectos, adaptando para QVT os conceitos de *joinpoint*, *pointcut* e *advice*. Além disso, para a especificação do *advice*, podem ser utilizadas estruturas próprias de QVT, como a definição de metamodelos a serem utilizados na transformação (por meio da palavra-chave *modeltype*), de regras de transformação (por meio da palavra-chave *mapping*) e de *helpers* (com a palavra-chave *helper*). A Figura 3.1 mostra uma representação em forma de metamodelo da estrutura de AQVT e sua relação com elementos de QVT (destacados em verde e definidos por [34]).

Um Aspecto é composto de zero ou mais *Pointcuts*. Esses então agrupam um ou mais *Joinpoints* e zero o mais *Advices*. Por sua vez, cada *Advice* é formado por expressões próprias da linguagem QVT, representados pela classe abstrata InnerType, no metamodelo. Essa estrutura de QVT foi adaptada do seu próprio metamodelo, definido por [34]. Os tipos diferentes de *Advice* aceitos são indicados de acordo com a *Enumeration AdviceKind: Before*, para que o código do *Advice* seja executado antes da execução do *Pointcut*, e *After*, para a execução do *Advice* depois do *Pointcut*. Já os tipos de *Joinpoints* contemplados por AQVT são representados pela *Enumeration JoinPointKind* (apresentados nas Tabelas 3.1 e 3.2). Ou seja, a estrutura geral de AQVT é a mesma definida pelo Paradigma Orientado a Aspectos, com Aspectos, *Joinpoints*,

Pointcuts e *Advices*. Entretanto, cada um desses elementos foi adaptado para o contexto de QVT-Operational e, especialmente, o *Advice* utiliza estruturas próprias dessa linguagem (como definição de regras e helpers), permitindo que o código seja executado na transformação QVT.

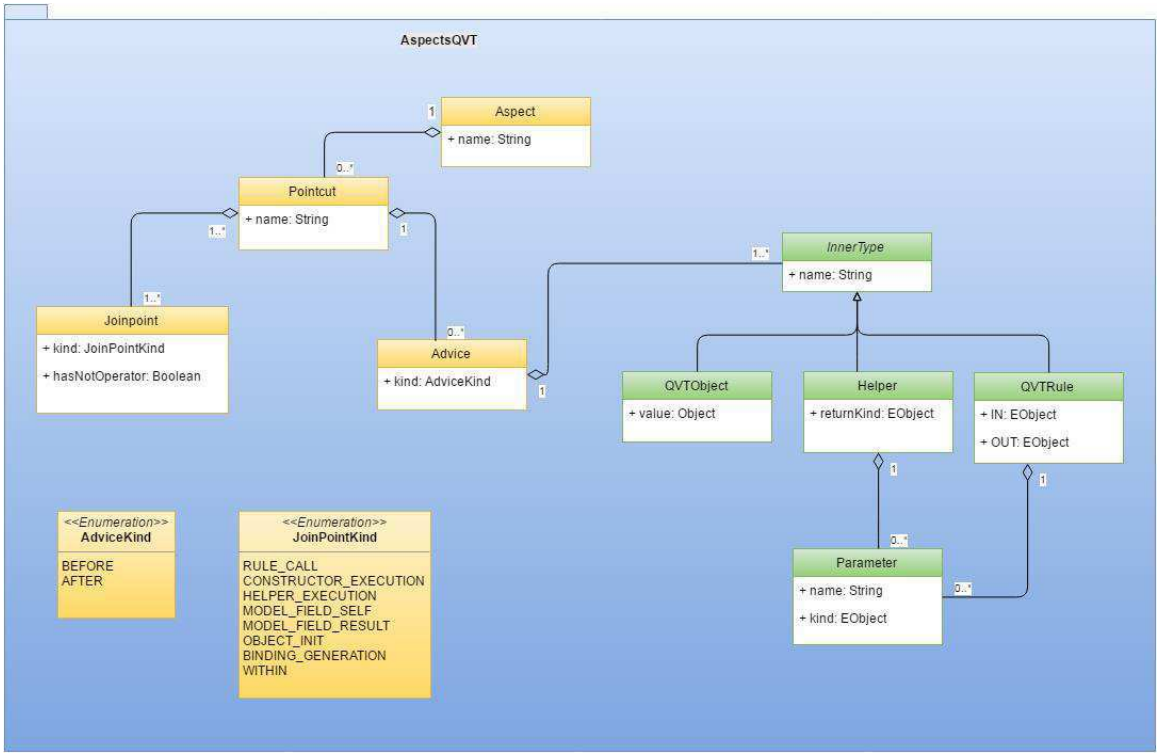

Figura 3.1. Metamodelo da Linguagem AQVT integrado com elementos QVT.

Joinpoint	Descrição
RULE_CALL	Indica quando uma regra é chamada por outra regra na transformação.
CONSTRUCTOR_EXECUTION	Indica quando um construtor é executado na transformação.
HELPER_EXECUTION	Indica quando um *helper* é executado por regras na transformação.
MODEL_FIELD_SELF	Indica quando um elemento no modelo de entrada é acessado na transformação.

Tabela 3.1. Variáveis Independentes no estudo.

Joinpoint	Descrição
MODEL_FIELD_RESULT	Indica quando um elemento no modelo de saída é acessado ou gerado na transformação.
OBJECT_INIT	Indica quando um objeto é inicializado na transformação.
BINDING_GENERATION	Indica quando um binding é formado na transformação.
WITHIN	Indica um local no código (Regra ou *Helper*) onde algo pode ocorrer durante a execução da transformação.

Tabela 3.2. Variáveis Independentes no estudo.

Um exemplo de como o código AQVT pode ser escrito é dado no Código 3.1.

```
Aspect TraceFiltered {
    pointcut filteredElements(e): Within(Rule package2schema) && FilterExecution(Filter *);

    after(): filteredElements(e) {
        modeltype Trace uses 'http://www.ufcg.edu.br/aqvt/traceqvt';

        mapping e::e trace_filteredElements(): Trace::FilteredElement {
            rulename := self.pointCut.rulename;
            element := self;
        }
    }
}
```

Código 3.1. Exemplo de programa AQVT.

O **aspecto** (Código 3.1) representa uma funcionalidade no programa que pode estar espalhada por todo o código (*crosscutting*). É nele que são definidos os *joinpoints*, *pointcuts* e *advices* do programa.

O ***joinpoint*** representa um ponto ou posição na execução do programa (Código 3.1, destacados em vermelho), enquanto que o ***pointcut*** é um conjunto de joinpoints para um aspecto (destacado em azul). Finalmente, o ***advice*** representa o código do aspecto que deverá ser executado quando a execução do programa atingir os pointcuts indicados no aspecto (destacado em verde).

O programa AQVT possui ao menos um aspecto (identificado pela palavra-chave "*Aspect"*), o qual poderá definir *pointcuts* (com a palavra-chave *"pointcut"*) que deverão executar seus respectivos códigos de *advice* quando seus *joinpoints* são atingidos durante a execução do programa. Cada *joinpoint* é definido juntamente com seu *pointcut*, utilizando-se das palavras-

chave citadas anteriormente e que serão melhor explicadas na próxima subseção. Além disso, cada um dos *joinpoints* são unidos por meio de *booleans*, "**&&**" (*and*) e "||" (*or*).

O código de um *advice* no aspecto é indicado com a utilização das palavras-chave *after* ou *before*, indicando que o código deverá ser executado depois ou antes do *pointcut* nomeado, respectivamente. A estrutura interna do advice é escrita utilizando-se uma forma básica do próprio código QVT, exemplificado no Código 3.1.

O código de uma regra deverá ser executado toda vez que ocorrer a execução de um filtro qualquer na regra "*package2schema*" do programa QVT, como indicado no *pointcut* "*filteredElements*". O trecho dessa regra de transformação pode ser visto no Código 3.2, onde é possível notar, na penúltima linha da regra, a existência de um filtro de elementos do tipo *Class UML,* aplicado sobre todos os elementos existentes dentro de uma *Package UML*. Ou seja, toda vez que esse filtro for executado, um mapeamento para elementos em um modelo de saída de rastreio (indicados por *Trace::FilteredElement* no Código 3.1) será executado.

```
mapping UML::Package::package2schema() : RDB::Schema
    when { self.hasPersistentClasses() }
{
    name := self.name;
    elements := self.ownedElements[UML::Class]->map persistentClass2table()->asOrderedSet()
}
```

Código 3.2. Trecho de código com regra de transformação QVT (retirado de [34]).

Além dos elementos de AQVT citados anteriormente, existe uma palavra-chave (*use*) que deverá ser sempre utilizada na primeira linha do programa, para referenciar o arquivo QVT sobre o qual os aspectos devem ser aplicados (ilustrado no Código 3.3).

```
use "C:\\Users\\Public\\Downloads\\workspace\\SimpleUML to RDB\\Simpleuml_To_Rdb.qvto"
```

Código 3.3. Trecho de código AQVT que indica o programa QVT a ser modificado.

3.1 Joinpoints e Pointcuts

Os Joinpoints contemplados pela linguagem AQVT são: "*Within*", "*FilterExecution*", "*BindingGeneration*", "*HelperExecution*", "*RuleCall*", "*ObjectInit*", "*ConstructorExecution*",

"*ModelFieldSelf*" e "*ModelFieldResult*". Um *pointcut* então é um conjunto formado por esses *joinpoints*.

A seguir, iremos ilustrar e explicar cada uma dessas estruturas de AQVT.

- ***Within***. *Sintaxe: Within(X Y)*, onde X é substituído por *Rule* ou *Helper*, enquanto que Y pode indicar um nome (completo ou parcial) para a regra ou helper, ou ***** para identificar "qualquer um". Identifica todo o código interno de uma regra ou um *helper*, e deve sempre ser utilizado em conjunto com outro *joinpoint* (através do booleano ***&&***, significando AND) para completar um *pointcut*. O Código 3.4 é um trecho de código AQVT que mostra o *pointcut ruleCall*, indicando que todas as chamadas da regra de transformação de nome "*primitiveAttributes2columns*" deverão ser observadas, quando realizadas **dentro** da regra "*dataType2columns*".

```
pointcut ruleCall(): Within(Rule dataType2columns) && RuleCall(Rule primitiveAttributes2columns);
```

Código 3.4. Trecho com *Joinpoint* "Within".

- ***FilterExecution***. *Sintaxe: FilterExecution(Filter Y),* onde Y é substituído por um nome (completo ou parcial) que identifica o tipo que está sendo filtrado, ou * para qualquer filtro. Quando o nome for parcial, serão verificados todos os filtros de elementos que começam com o prefixo dado. Este joinpoint indica o momento no qual um filtro é executado no código do programa QVT. O Código 3.5 é um trecho de código AQVT que mostra o *pointcut filteredElements*, indicando que todas as execuções de Filtros na regra "*package2schema*" deverão ser observadas. Além disso, em "*filteredElements(e)*", o parâmetro indicado por "e" serve para representar os elementos que foram filtrados, podendo ser utilizado no *Advice* do Aspecto com esse propósito.

```
pointcut filteredElements(e): Within(Rule package2schema) && FilterExecution(Filter *);
```

Código 3.5. Trecho com *Joinpoint* "FilterExecution".

- ***BindingGeneration***. *Sintaxe: BindingGeneration(Binding Y),* onde Y é substituído por um nome (completo ou parcial) que identifica qualquer um dos elementos que constituem o *binding* específico (tanto o elemento do modelo de entrada quanto o elemento do modelo de saída). Quando o nome for parcial, serão verificados todos os *bindings* de elementos que começam com o prefixo dado. Já quando Y é substituído por *, ele indica qualquer *binding* no código. Este joinpoint identifica o momento no qual um *binding* é gerado no código do programa QVT. O trecho revelado no Código 3.6 mostra o *pointcut traceBinding*, indicando que todos os *bindings* gerados dentro da

regra "*class2primaryKey*" serão observados. O parâmetro "b" em "*traceBinding(b)*" representa o *binding* que foi capturado pelo *pointcut*, podendo ser referenciado no *Advice*.

```
pointcut traceBinding(b): BindingGeneration(Binding *) && Within(Rule class2primaryKey);
```

Código 3.6. Trecho com *Joinpoint* "BindingGeneration".

- ***HelperExecution***. *Sintaxe: HelperExecution(Helper Y),* onde Y é substituído por um nome (completo ou parcial) que identifica o *helper* sendo executado (ou * para qualquer *helper* executado) em uma regra. Quando o nome for parcial, serão verificados todos os *helpers* executados que começam com o prefixo dado. Este *Joinpoint* identifica o momento no qual um *helper* é executado no código do programa QVT. O Código 3.7 revela o *pointcut helperRun*, que faz uso de dois *joinpoints* do tipo *HelperExecution.* Isso faz com que, toda vez que os *helpers* "*asDataType*" ou *"isIdentifying*" são executados em qualquer regra, o *pointcut* é atingido.

```
pointcut helperRun(): Within(Rule *) && (HelperExecution(Helper asDataType) || HelperExecution(Helper isIdentifying));
```

Código 3.7. Trecho com *Joinpoint* "HelperExecution".

- ***RuleCall***. *Sintaxe: RuleCall(Rule Y),* onde Y é substituído por um nome (completo ou parcial) que identifica a regra sendo chamada, ou * para qualquer regra na transformação. Quando o nome for parcial, serão verificadas todas as regras chamadas que começam com o prefixo dado. Este *joinpoint* identifica o momento no qual uma regra é chamada no código do programa QVT. O *pointcut ruleCall* exemplificado no Código 3.8 sempre será atingido quando a regra "*primitiveAttributes2columns*" for chamada na regra "*dataType2columns*".

```
pointcut ruleCall(): Within(Rule dataType2columns) && RuleCall(Rule primitiveAttributes2columns);
```

Código 3.8. Trecho com *Joinpoint* "RuleCall".

- ***ObjectInit***. *Sintaxe: ObjectInit(Object Y),* onde Y é substituído por um nome (completo ou parcial) que identifica o tipo do objeto sendo inicializado, ou * para qualquer objeto inicializado. Quando o nome for parcial, serão verificados todos os objetos inicializados que começam com o prefixo dado. Este *joinpoint* identifica o momento no qual um objeto é inicializado no código do programa QVT. O trecho do Código 3.9 mostra o *pointcut objectInitialization*, que indica a inicialização de

qualquer objeto na regra "*primitiveAttribute2Column*". Adicionalmente, o parâmetro "o" em "*objectInitialization(o)*" faz referência ao objeto capturado pelo *pointcut*.

```
pointcut objectInitialization(o): ObjectInit(Object *) && Within(Rule primitiveAttribute2Column);
```

Código 3.9. Trecho com *Joinpoint* "ObjectInit".

- ***ConstructorExecution***. *Sintaxe: ConstructorExecution(Constructor Y),* onde Y é substituído por um nome (completo ou parcial) que identifica a execução de um construtor específico em uma regra, ou qualquer construtor quando Y é substituído por *. Quando o nome for parcial, serão verificados todos os construtores executados que começam com o prefixo dado. Este *joinpoint* indica o momento no qual um construtor é executado no código do programa QVT. O Código 3.10 revela o *pointcut traceConstructor*, que observa a execução do construtor para "*PrimitiveDataType*" na regra "*dataType2primaryKeyColumns*".

```
pointcut traceConstructor(): ConstructorExecution(Constructor PrimitiveDataType) &&
                             Within(Rule dataType2primaryKeyColumns);
```

Código 3.10. Trecho com *Joinpoint* "ConstructorExecution".

- ***ModelFieldSelf***. *Sintaxe: ModelFieldSelf(Field Y),* onde Y é substituído por um nome (completo ou parcial) que identifica um campo específico no modelo de entrada que esteja sendo acessado, ou um campo qualquer no modelo de entrada quando o símbolo * é utilizado. Quando o nome for parcial, serão verificados todos os acessos a campos do modelo de entrada que começam com o prefixo dado. Este *joinpoint* identifica o momento no qual um elemento do modelo de entrada é acessado no código do programa QVT. O Código 3.11 mostra o *pointcut traceModelField*, indicando todos os acessos a elementos no modelo de entrada realizados na regra "*package2schema*". Além disso, o parâmetro "f" representa, para o *pointcut* e seu respectivo *advice*, o campo que foi acessado.

```
pointcut traceModelField(f): ModelFieldSelf(Field *) && Within(Rule package2schema);
```

Código 3.11. Trecho com *Joinpoint* "ModelFieldSelf".

- ***ModelFieldResult***. *Sintaxe: ModelFieldResult(Field Y),* onde Y é substituído por um nome (completo ou parcial) que identifica um campo específico no modelo de saída que esteja sendo acessado no momento, ou um campo qualquer no modelo de saída quando o símbolo * é utilizado.

Quando o nome for parcial, serão verificados todos os acessos a campos no modelo de saída (ou quando são gerados) que começam com o prefixo dado. Este *joinpoint* identifica o momento no qual um elemento do modelo de saída é acessado ou gerado no código do programa QVT. O trecho de código 3.12 revela o *pointcut traceModelField*, que é atingido toda vez que um campo qualquer no modelo de saída é acessado dentro da regra "*package2schema*". Adicionalmente, o parâmetro "f" representa, para o *pointcut* e seu respectivo *advice*, o campo que foi acessado no modelo.

```
pointcut traceModelField(f): ModelFieldResult(Field *) && Within(Rule package2schema);
```

Código 3.12. Trecho com *Joinpoint* "ModelFieldResult".

3.2 Advice

O *Advice* em AQVT é o código que será executado toda vez que o seu respectivo Joinpoint é atingido na execução do programa. Ele utiliza parte da gramática QVT para permitir a definição de regras, helpers e novos metamodelos de entrada e/ou saída, além de incluir a utilização da palavra-chave "self" para indicar o elemento ou Joinpoint que está sendo executado no momento. O *Advice* exemplificado no trecho do Código 3.13 define tanto a utilização de um metamodelo de saída adicional, "Trace", como uma nova regra, "traceRules", que necessitará desse metamodelo para a geração de um modelo de saída com dados sobre a execução do programa QVT, ou seja, o nome da regra que foi chamada e da que chamou, identificados por "self.pointCut.calledRule" e "self.pointCut.rulename", respectivamente. Em resumo, quando executado, o Advice do Código 3.13 gera rastros da execução de uma regra, com o auxílio da palavra-chave *self.pointcut*, que possui informações do *pointcut* atingido, como o nome da regra que está em execução no momento e a regra de transformação que foi executada pela regra anterior.

```
after(): ruleCall() {
    modeltype Trace uses 'http://www.ufcg.edu.br/aqvt/traceqvt';

    mapping Ecore::String traceRules(): Trace::RuleExecutions {
        rulename := self.pointCut.rulename;
        calledrule := self.pointCut.calledrule;

    }
}
```

Código 3.13. Advice para o *Pointcut* "ruleCall".

3.3 Exemplo de Uso

Um exemplo de uso da linguagem AQVT é o de rastrear a chamada de regras em outras regras *de acordo com sua execução por meio de diferentes modelos de entrada*. O trecho dado no Código 3.14 mostra a regra "*dataType2columns*", que gera um OrderedSet de colunas de tabela a partir de um DataType. Para realizar isso e computar os valores definidos, a regra faz a chamada de outras quatro regras, "*primitiveAttributes2columns*", "*enumerationAttributes2columns*", "*relationshipAttributes2columns*" e "*associationAttributes2columns*".

```
mapping UML::DataType::dataType2columns(in targetType : UML::DataType) : OrderedSet(RDB::TableColumn) {
    init {
        result := self.map primitiveAttributes2columns(targetType)->
            union(self.map enumerationAttributes2columns(targetType))->
            union(self.map relationshipAttributes2columns(targetType))->
            union(self.map associationAttributes2columns(targetType))->asOrderedSet()
    }
}
```

Código 3.14. Trecho de código QVT (Retirado de [34]).

Para identificar se a regra "primitiveAttribute2columns", por exemplo, é utilizada com frequência ou não na regra "dataType2columns", podemos utilizar o código AQVT (Código 3.15), que registrará em um novo modelo de saída o nome da regra que realizou a chamada (definido em "rulename") e o nome da regra chamada (definido em "calledRule"), para cada chamada realizada.

```
use "C:\\Users\\Public\\Downloads\\workspace\\SimpleUML to RDB\\Simpleuml_To_Rdb.qvto"

Aspect RuleCalls {
    pointcut ruleCall(): Within(Rule dataType2columns) && RuleCall(Rule primitiveAttributes2columns);

    after(): ruleCall() {
        modeltype Trace uses 'http://www.ufcg.edu.br/aqvt/traceqvt';

        mapping Ecore::String traceRules(): Trace::RuleExecutions {
            rulename := self.pointCut.rulename;
            calledrule := self.pointCut.calledrule;

        }
    }
}
```

Código 3.15. Programa AQVT para rastreamento de chamadas de regras.

O *pointcut ruleCall* define que seu advice deverá ser executado toda vez que a regra "*primitiveAttributes2columns*" for chamada pela regra "*dataType2columns*": um metamodelo de rastreio é então incluso na definição da transformação e a execução ou chamada da regra "*primitiveAttributes2columns*" é rastreada no novo modelo de saída.

Outro exemplo de uso é a aplicação da função de *logging* antes e depois da criação de *bindings*. O trecho dado no Código 3.16 mostra a regra "*model2RDBModel*", que gera um elemento do tipo *Model* em RDB a partir de um elemento *Model* em UML. Nessa regra existe a aplicação de dois bindings, relacionando os atributos nome dos elementos nos modelos de entrada e saída, e o atributo *schema* de RDB com *packages* de UML, por meio da execução de outra regra (*package2schemas*).

```
mapping UML::Model::model2RDBModel() : RDB::Model {
    name := self.name;
    schemas := self.map package2schemas()->asOrderedSet();
}
```

Código 3.16. Trecho de Código QVT.

Para checar a criação dos *bindings* nessa regra, por exemplo, podemos utilizar o Código 3.17, de AQVT, que aplicará os métodos *log()* antes e depois de cada *binding*.

```
use "C:\\Users\\Public\\Downloads\\workspace\\SimpleUML to RDB\\Simpleuml_To_Rdb.qvto"

Aspect AddLogging {
    pointcut logging(b): BindingGeneration(Binding *) && Within(Rule model2RDBModel);

    before(): logging(b) {
        log("Starting log");
    }

    after(): logging(b) {
        log("Ending log");
    }
}
```

Código 3.17. Programa AQVT para adição de logging em *bindings*.

O *pointcut logging* define que seu advice deverá ser executado toda vez que a regra "*model2RDBModel*" gerar um *binding* qualquer. da regra "*primitiveAttributes2columns*" é rastreada no novo modelo de saída. Antes que isso ocorra, é executado o código de *log()* e, depois que o binding é gerado, é executada mais uma linha de *log()*.

3.4 Weaving

Para que o código final gerado execute suas operações extras corretamente, há a necessidade de se realizar o *Weaving* do código original com o código de Aspectos. Dessa forma, construímos o compilador de AQVT, que realiza o *Weaving* do código AQVT com o código original QVT, gerando um código-fonte final a ser executado pelo compilador QVT (processo ilustrado na Figura 3.2). Para que isso ocorra, o compilador da linguagem de Aspectos em QVT inicialmente analisa o programa QVT inicial, coletando informações como relações entre regras, *bindings* e filtros executados, entre outros. Em seguida, é feita a interpretação do código AQVT, identificando-se no programa original os *Joinpoints* indicados pelo programa AQVT e os aspectos (código do *advice*) que devem ser aplicados no código original. Finalmente, o compilador gera o código necessário e realiza seu *weaving* com o programa QVT.

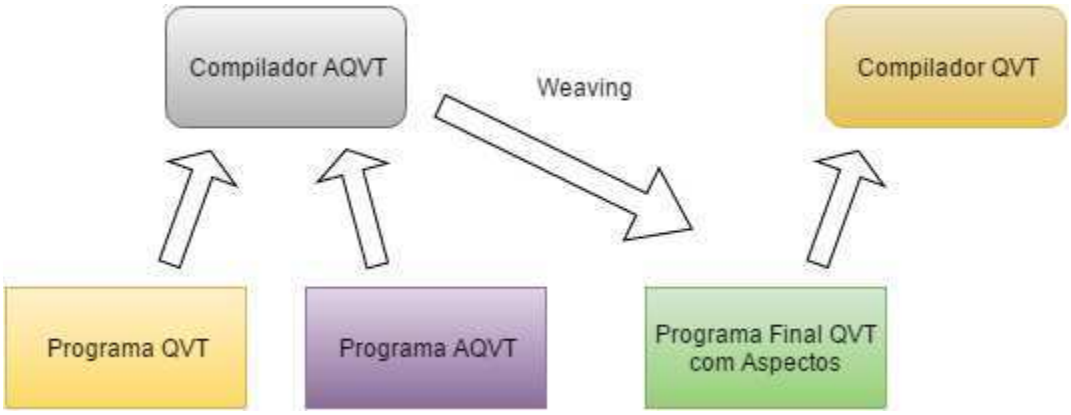

Figura 3.2. Processo de Weaving em AQVT.

Para exemplificar o processo de *Weaving*, temos o Código 3.18, que mostra um trecho de código do programa QVT "Simpleuml2RDB", que transforma um modelo simples UML para um modelo de banco de dados relacional. O trecho revela uma regra (ou *mapping*) que mapeia elementos UML do tipo *Packages* para elementos RDB do tipo *Schema,* quando os *Packages* possuem classes persistentes.

```
mapping UML::Package::package2schema() : RDB::Schema
    when { self.hasPersistentClasses() }
{
    name := self.name;
    elements := self.ownedElements[UML::Class]->map persistentClass2table()->asOrderedSet()
}
```

Código 3.18. Trecho do programa QVT "Simpleuml2RDB" (retirado de [34]).

Os elementos do *Schema* RDB são preenchidos de forma ordenada com a utilização de outra regra de mapeamento, "persistentClass2table()", aplicada sobre todos os elementos do tipo "Class" em UML. Esses elementos são obtidos por meio de um filtro, representado pelo trecho definido em colchetes "[UML::Class]". Como forma de se obter o momento em que esse filtro é executado (por questões de rastreamento, por exemplo), um desenvolvedor pode definir o código AQVT no Código 3.19.

```
use "C:\\Users\\Public\\Downloads\\workspace\\SimpleUML to RDB\\Simpleuml_To_Rdb.qvto"

Aspect TraceFiltered {
    pointcut filteredElements(e): Within(Rule package2schema) && FilterExecution(Filter *);

    after(): filteredElements(e) {
        modeltype Trace uses 'http://www.ufcg.edu.br/aqvt/traceqvt';

        mapping e::e trace_filteredElements(): Trace::FilteredElement {
            rulename := self.pointCut.rulename;
            element := self;
        }
    }
}
```

Código 3.19. Código AQVT Para rastreamento de Filtros na Regra "package2schema".

O Aspecto definido no Código 3.19 utiliza-se de um *PointCut* (nomeado "filteredElements") com os *Join Points* "Within(Rule package2schema)" e "FilterExecution(Filter *)", indicando que o *Advice* do Aspecto deverá ser executado sempre que um Filtro qualquer for executado na Regra "package2schema". Esse Advice irá gerar um modelo de saída, em conformidade com um metamodelo genérico de rastreio, indicando o nome da regra onde o filtro foi executado e o(s) elemento(s) filtrado(s).

O *Weaving* realizado pelo compilador AQVT faz a união dos códigos, mantendo a funcionalidade desejada, como pode ser visto no Código 3.20. Uma nova regra ou mapeamento de

transformação foi adicionada no programa final, e uma chamada desta regra foi incluída na regra "package2schema()" (contornados em vermelho), tendo os elementos filtrados como entrada.

```
mapping UML::Package::package2schema() : RDB::Schema
    when { self.hasPersistentClasses() }
{
    name := self.name;
    elements := self.ownedElements[UML::Class]->map persistentClass2table()->asOrderedSet();
    self.ownedElements[UML::Class] -> map aspect_filteredElementsPackage2schema();
}

mapping UML::Class::aspect_filteredElementsPackage2schema() : TraceMetamodel::FilteredElement {
    rulename := "package2schema";
    element := self;
}
```

Código 3.20. Trecho do código final QVT com *Weaving*.

Para exemplificar a execução do programa com *weaving*, damos o modelo da Figura 3.3 como entrada e a saída normal, o modelo da Figura 3.4, gerada com o programa original. Já com a inclusão do Aspecto, um modelo extra é dado na saída, como visto na Figura 3.5, onde existe um elemento para cada execução do filtro no programa QVT.

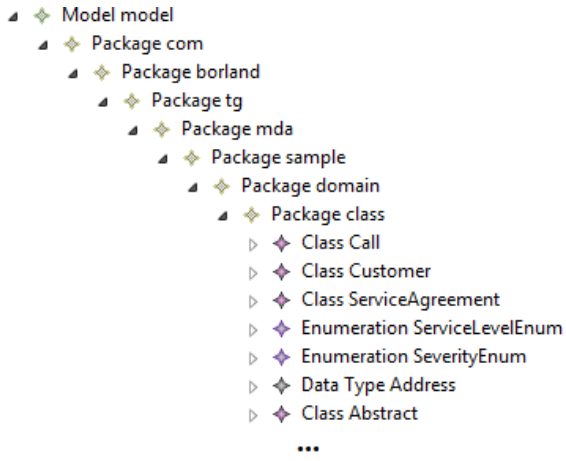

Figura 3.3. Modelo exemplo de entrada para o programa QVT.

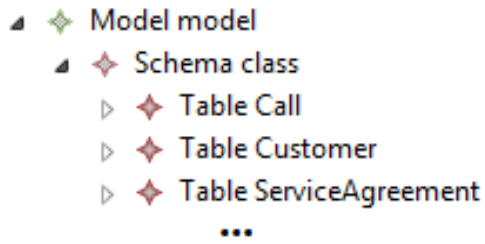

Figura 3.4. Modelo de saída após a execução normal do programa.

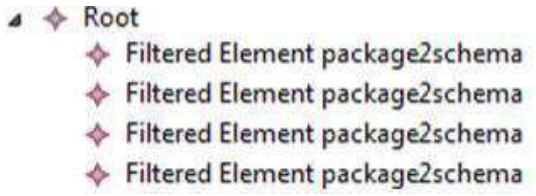

Figura 3.5. Modelo extra de saída após a execução com Aspectos.

Como pode ser visto nos exemplos anteriores, o *weaving* realizado pelo compilador utiliza também uma abordagem relacionada às *Inter-type Declarations* de AspectJ, onde regras e helpers (por exemplo) que estão sendo utilizadas no Advice de um Aspecto são implementadas no programa final gerado após o *weaving*, mantendo o comportamento que foi definido com a escrita do programa AQVT. Esses novos elementos não são visíveis ao programador, e apenas são executados normalmente com o programa QVT, sem a intervenção do usuário.

O Compilador AQVT foi implementado na linguagem de programação Xtend [16], uma extensão de Java, auxiliado por meio de uma gramática de AQVT (dada no Apêndice A), que foi definida por meio da linguagem e *framework* Xtext [17]. Uma das funções realizadas pelo Compilador AQVT é a análise estática [5] do código QVT, que tem o objetivo de identificar todos os *Joinpoints* no código possíveis e que são contemplados em AQVT. Com essas informações coletadas, o compilador utiliza o programa AQVT para indentificar quais *Joinpoints* devem ser utilizados e gera o código de weaving necessário, seguindo o *Advice* de AQVT. Finalmente, o código gerado é um programa puro QVT com as funcionalidades que haviam sido definidas no código de Aspectos.

O código do Advice, como visto nos exemplos anteriores, não é copiado diretamente para o código final gerado pelo compilador. São criadas nos pontos do código atingidos pelo Pointcut as chamadas para as regras e helpers que foram definidos no escopo do advice, enquanto que essas estruturas de QVT são adicionadas especificadas normalmente no código final QVT.

Para checar se o código final gerado pelo compilador está de acordo com as especificações tanto de AQVT como QVT, o programa é executado com e sem aspectos (mas equivalentes) e, no

fim, os modelos de saída gerados são comparados, verificando-se sua igualdade. Para a transformação gerada em si, verificamos a existência ou não de erros de sintaxe por meio do próprio compilador QVT e, por fim, analisamos visualmente se a semântica dos programas gerados estava de acordo com todas as funcionalidades que foram definidas tanto na transformação QVT quanto no programa AQVT. Esses testes foram realizados de forma manual sobre as transformações e modelos gerados com sua execução, mas uma verificação mais robusta poderia ser feita por meio de técnicas de teste formais.

Capítulo 4

Avaliação

Após o desenvolvimento de AQVT e seu compilador, fez-se necessário realizar sua avaliação, com o intuito de verificar se o compilador realiza o weaving do código corretamente, e também se a linguagem auxilia realmente em questões de qualidade de leitura, compreensão e modularidade do código.

Esta seção apresenta a avaliação que foi realizada, seu planejamento, seus resultados e, finalmente, uma discussão sobre os dados obtidos.

4.1 Planejamento da Avaliação

Cross-cutting concerns, como *logging* e rastreio de execução, são problemas que podem ocorrer em transformações QVT e a utilização de Aspectos pode ser uma forma de resolver esses problemas. Entretanto, não sabemos como sua aplicação poderá afetar a modularidade do código, além de sua qualidade de leitura e compreensão. Além disso, a utilização de um compilador de Aspectos, sem que suas operações sejam testadas, poderá diminuir a confiança na linguagem caso erros sejam introduzidos no processo de geração de código. Desse modo, definimos o seguinte modelo GQM para nosso estudo, sintetizando-o na Tabela 4.1:

Analisar **a linguagem de aspectos em QVT e seu compilador** com a intenção de **avaliá-los e caracterizá-los** com respeito **à compreensão e modularidade do código, além da eficiência na geração de código** do ponto de vista **dos desenvolvedores de projetos QVT** no contexto de **desenvolvedores programando na linguagem QVT e a utilização da linguagem de aspectos QVT com um novo compilador**.

Objetos do estudo	Finalidade	Foco de qualidade	Perspectiva	Contexto
Linguagem de Aspectos em QVT	Avaliar	Compreensão Modularidade	Desenvolvedores de projetos QVT	Desenvolvedores programando na linguagem QVT Utilização de AQVT

Tabela 4.1. Objetivos GQM.

4.2 Questões de Pesquisa

Realizamos um experimento controlado para a pesquisa, utilizando a execução de programas QVT-Operational **com** e **sem** a adição de aspectos (AQVT). Os programas utilizados foram coletados de repositórios online, como os da Eclipse Foundation [14]. Definimos certas métricas de leitura de código e modularidade e as aplicamos nas transformações, tanto com, como sem aspectos. Tais métricas foram elaboradas e adaptadas com base em outras métricas já concebidas na literatura, relacionando tanto modularidade quanto leitura e compreensão do código (como em [7] [2] [24] [55]). Optamos principalmente por utilizar esse estilo de experimento, com métricas objetivas, devido à dificuldade de encontrarmos desenvolvedores, com conhecimento em QVT, disponíveis para realizar um estudo de caso, além da complexidade inicial de prepararmos o ambiente de trabalho necessário para realizar a pesquisa.

Assim, definimos duas Questões de Pesquisa:

QP1	
As duas formas de programação (QVT puro e com aspectos) apresentam diferenças entre a qualidade de leitura e compreensão de programas?	
H1-0	A utilização da linguagem de aspectos em QVT é igual à programação sem aspectos em QVT no quesito de qualidade de leitura e compreensão do código.
H1-1	A utilização da linguagem de aspectos em QVT é melhor do que a programação sem aspectos em QVT no quesito de qualidade de leitura e compreensão do código.

QP2	
As duas formas de programação (QVT puro e com aspectos) apresentam diferenças na questão de modularidade dos programas?	
H2-0	A utilização da linguagem de aspectos em QVT é igual à programação sem aspectos em QVT no quesito de modularidade do código.
H2-1	A utilização da linguagem de aspectos em QVT é melhor do que a programação sem aspectos em QVT no quesito de modularidade do código.

A qualidade de leitura foi escolhida com base em métricas e discussões realizadas em [7] e [2], por exemplo. Enquanto que em [7] a qualidade de leitura é afetada pelo tamanho do programa (linhas de código) e sua quantidade de construções ou elementos, em [2] é a complexidade desses elementos (vários elementos diferentes em um único método, por exemplo) que também pode afetar a leitura completa do código. Os autores de [7] e [2] consideram a qualidade de leitura como a facilidade de compreensão do programa observando-se o seu tamanho e quantidade de elementos, onde programas maiores são normalmente mais "difíceis" de se compreender do que programas menores. Quanto menor o valor dessa métrica, melhor seria a qualidade ou o grau de leitura do programa. Assim, levamos em conta questões que podem tornar mais oneroso o processo de análise e compreensão do código, como a maior quantidade de linhas de código e utilização de condicionais *When* e *Where*.

Já a definição de modularidade do código foi baseada em métricas de trabalhos como em [24] e [55], onde o espalhamento e a maior utilização de regras ou helpers por outras regras afeta e dificulta a separação do programa em módulos. Nesses trabalhos, a modularidade é vista especificamente como o nível de acoplamento entre os diferentes elementos do programa. Quanto mais relações existem entre suas diferentes construções, mais acoplado é o programa e pior é a sua modularidade.

4.3 Variáveis

As variáveis independentes que utilizamos no estudo são apresentadas na Tabela 4.2.

Variável Independente	Descrição
Técnica de Programação QVT	Aplicação da linguagem de aspectos ou não sobre os programas QVT.
Tamanho do Programa QVT	Programas QVT de diferentes tamanhos: **pequeno** (menos de 200 linhas de código), **médio** (entre 200 e 400 linhas de código) e **grande** (mais de 400 linhas de código).

Tabela 4.2. Variáveis Independentes no estudo.

Para os tamanhos dos programas QVT, em particular, definimos os três grupos (pequeno, médio e grande) de modo que cada grupo se aproxime de um número igual de transformações estudadas. Os tamanhos dos programas foram utilizados para o agrupamento de transformações com tamanhos similares e sua seleção dentre todas as transformações coletadas dos repositórios, de modo que as transformações selecionadas não sejam completamente enviesadas para um tamanho ou outro.

Com o objetivo de responder às Perguntas de Pesquisa, realizamos uma análise de trabalhos que envolveram Aspectos ou Transformações entre Modelos, coletando informações sobre métricas relacionadas a esses temas [7] [2] [24] [55]. Desse modo, elaboramos métricas de avaliação do código, representando as varíaveis dependentes ou de resposta do estudo, como pode ser visto nas Tabelas 4.3, 4.4, 4.5 e 4.6.

Métricas de Leitura e Compreensão		
Nome	**Valor**	**Descrição**
LDC	*LinhasDeCódigo*	Esta métrica representa a quantidade de linhas de código. Quanto maior for seu valor, mais difícil pode ficar a compreensão do programa QVT. Adaptada de [2] e [24].

Tabela 4.3. Métricas de Leitura e Compreensão utilizadas como Variáveis Dependentes.

Métricas de Leitura e Compreensão		
Nome	**Valor**	**Descrição**
MLdCR	**M**édia de **L**inhas **d**e **C**ódigo por **R**egra $\frac{LinhasDeCódigoDeRegras}{NúmeroDeRegras}$	Esta métrica representa a média da quantidade de linhas de código de regras pelo número de regras na transformação. À medida em que esse valor cresce, a compreensão do programa QVT pode ser prejudicada. Adaptada de [55].
MLdCH	**M**édia de **L**inhas **d**e **C**ódigo por **H**elper $\frac{LinhasDeCódigoDeHelpers}{NúmeroDeHelpers}$	Esta métrica representa a média da quantidade de linhas de código de helpers pelo número de helpers na transformação. À medida em que esse valor cresce, a compreensão do programa QVT pode ser prejudicada. Adaptada de [55].
MCR	**M**édia de **C**ondicionais por **R**egra $\frac{NúmeroTotalDeCondicionais}{NúmeroDeRegras}$	Esta métrica representa a média de condicionais *when* e *where* por regra. Quanto maior seu valor, mais difícil pode ficar a compreensão ou leitura da transformação. Adaptada de [24].
Qualidade de Leitura e Compreensão	$\frac{\Sigma(Normalizar(M_i))}{TotalDeMetricasP1}$	Utilizando as métricas LDC, $MLdCR$, $MLdCH$ e MCR e normalizando-as individualmente para os resultados obtidos com todas as transformações coletadas, faz-se o somatório das métricas de um programa e, então, sua divisão pelo número total de métricas que afetam a leitura ou compreensão do código. Um valor muito próximo de 1 pode indicar baixa **legibilidade** e dificuldade para a **compreensão** do programa, enquanto que valores mais próximos de 0 podem indicar boa legibilidade e compreensão da transformação.

Tabela 4.4. Métricas de Leitura e Compreensão utilizadas como Variáveis Dependentes.

Métricas de Modularidade		
Nome	**Valor**	**Descrição**
MaxCRR	Valor **Máx**imo para **C**hamadas de **R**egras em **R**egras $Max(\frac{ChamadasDeRegra(i)}{NúmeroTotalChamadasDeRegra})\forall i \in Regras$	Esta métrica indica um valor que representa a regra **mais** utilizada por outras **regras** na transformação. Seu valor é calculado com a ajuda da função *ChamadasDeRegra(i)*, que retorna a quantidade total de chamadas de uma regra "i" no programa. Esse número é então dividido pelo número total de chamadas de regras na transformação. Finalmente, o valor é calculado para todas as regras do programa, retornando-se apenas o maior número encontrado. À medida em que o valor desta métrica se aproxima de 1, ele pode indicar a existência de uma regra que é muito mais utilizada na transformação do que outras regras, indicando alta dependência no código e prejudicando sua modularidade. Adaptada de [7].
MaxCHR	Valor **Máx**imo para **C**hamadas de **H**elpers em **R**egras $Max(\frac{ChamadasDeHelper(i)}{NúmeroTotalDeChamadasDeHelper})\forall i \in Helpers$	Esta métrica indica um valor que representa o helper **mais** utilizado por **regras** na transformação. Seu valor é calculado com a ajuda da função *ChamadasDeHelper(i)*, que retorna a quantidade total de chamadas de um helper "i" no programa. Esse número é então dividido pelo número total de chamadas de helpers na transformação. Finalmente, o valor é calculado para todos os helpers do programa, retornando-se apenas o maior número encontrado. À medida em que o valor desta métrica se aproxima de 1, ele pode indicar a existência de um helper que é muito mais utilizado na transformação do que outros helpers, indicando alta dependência no código e prejudicando sua modularidade. Adaptada de [7].

Tabela 4.5. Métricas de Modularidade utilizadas como Variáveis Dependentes.

Métricas de Modularidade		
Nome	**Valor**	**Descrição**
MCRR	**M**édia de **C**hamadas de **R**egras em **R**egras $\frac{NúmeroTotalDeChamadasDeRegra}{NúmeroDeRegras}$	Esta métrica representa a **média** do total de **chamadas de regras** pelo número de **regras**. À medida em que seus valores crescem, a modularidade do programa pode ser afetada. Adaptada de [7].
MCHR	**M**édia de **C**hamadas de **H**elpers em **R**egras $\frac{NúmeroTotalDeChamadasDeHelper}{NúmeroDeHelpers}$	Esta métrica representa a **média** do total de **chamadas de helpers** pelo número de **helpers**. À medida em que seus valores crescem, a modularidade do programa pode ser afetada. Adaptada de [7].
Modularidade	$\frac{\Sigma(Normalizar(M_i))}{MetricasTotalP2}$	Utilizando as métricas $MaxCRR$, $MaxCHR$, $MCRR$ e $MCHR$ e normalizando as duas últimas, individualmente para os resultados obtidos com todas as transformações, faz-se o somatório das métricas de um programa e, então, sua divisão pelo número total de métricas que afetam a modularidade do código. Um valor muito próximo de 1 pode indicar problemas de **modularidade** do programa, enquanto que valores mais próximos de 0 podem indicar melhor modularidade na transformação.

Tabela 4.6. Métricas de Modularidade utilizadas como Variáveis Dependentes.

Foi realizada a decisão de normalizar as métricas para permitir que os dados possam ser comparados e análisados de forma mais clara, além de faciliar a busca de possíveis relações entre os resultados obtidos, como a tendência de alguns programas de transformação em exibir resultados similares para a qualidade de leitura e modularidade, por exemplo. Além disso, a normalização é necessária para impedir que valores muito altos ou baixos, como número de linhas de código, afetem fortemente os resultados e também permitir que valores de escalas diferentes

possam ser agrupados e comparados. Algo similar foi realizado em [24], com a normalização conjunta de diferentes métricas definidas no trabalho, como número de helpers e regras, linhas de código, entre outros. Foi possível normalizar essas variáveis devido ao escopo restrito para tais elementos de QVT, que possuem escalas bastante próximas.

Já para a análise dos programas gerados pelo compilador AQVT, realizamos testes simples de acordo com a Tabela 4.7.

Teste	Descrição
Comparação de Transformação.	Fazemos a comparação da transformação gerada a partir do weaving do compilador AQVT com uma transformação ideal (definida com antecedência) sobre a qual o weaving foi aplicado corretamente. Verificamos a existência ou não de linhas de código com erros sobre todo o programa QVT. Para todas as transformações, não encontramos erros de sintaxe ou semântica.

Tabela 4.7. Testes de programas gerados pelo compilador AQVT.

4.4 Unidades Experimentais

As unidades experimentais foram os programas QVT, disponibilizados em repositórios abertos na internet. Aplicamos a esses programas de transformação certos tratamentos que representam *cross-cutting concerns*, como logging e execução de regras em regras, tanto com a utilização de AQVT quanto com a utilização pura de código QVT. Dessa forma, obtivemos dois grupos de transformações utilizados no estudo: um com e outro sem AQVT.

4.5 Instrumentos

O estudo foi realizado com os seguintes instrumentos:

- Programas QVT, modelos e meta-modelos de repositórios de projetos MDA, como os do Eclipse;
- Eclipse IDE [9] juntamente com o framework QVT, com a inclusão do compilador de Aspectos em QVT;
- Ferramenta R [45] para a aplicação de métodos estatísticos e a criação de diagramas para os dados.

Os programas QVT utilizados, sua fonte e descrição são apresentadas nas Tabelas 4.8 e 4.9.

Transformação	Fonte	Descrição
SimpleUML2RDBMS	Exemplo disponibilizado pelo projeto QVT-O Eclipse [14]	Realiza a transformação de um modelo simples UML para um modelo de banco de dados relacional.
transformation-psm1	Projeto "Event based Communication in PCM" [27]	Transforma elementos específicos de eventos para elementos clássicos PCM, que são suportados por um engenho de simulação no projeto.
outputSimqpnResults	Projeto "PCM2QPN" [30]	Faz o retorno de vários resultados de execuções do projeto (dados em modelos de entrada) em uma única saída.
pcm2qpe	Projeto "PCM2QPN"	Realiza a transformação principal de um Palladio Component Model [4] para Queueing Petri Nets [3].

Tabela 4.8. Descrição dos Programas QVT Coletados.

Transformação	Fonte	Descrição
transformation-psm2	Projeto "PCM2QPN"	Transforma elementos específicos de eventos para elementos clássicos PCM, no escopo do projeto PCM2QPN.
SAMM2PCM_allocation	"Project Q-ImPrESSQ-Impress" [42]	Faz a alocação de elementos específicos da Q-Impress para PCM. (Disponível no Anexo B)
SAMM2PCM_seff	"Project Q-ImPrESSQ-Impress"	Faz a inicialização de um repositório de elementos PCM.
uml2simpleSTM	Projeto "qvto-flatten-stm" [43]	Realiza a clonagem (ou seja, a cópia) de um modelo de máquina de estados.
Allocation	Retirado do projeto "QVTR2" [44]	Cria um modelo de alocação a partir de modelos de componente e hardware.
Uml2Rdbms_Replication	Projeto "QVTR2"	Cria um modelo de banco de dados relacional a partir de um modelo UML, com mais atributos do que a transformação exemplo do Eclipse. (Disponível no Anexo C)
Sofa2adl	Projeto "SOFA2" [51]	Converte modelos SOFA2 para uma versão ADL.

Tabela 4.9. Descrição dos Programas QVT Coletados.

Para realizarmos o experimento, definimos os tratamentos das Tabelas 4.10 e 4.11 a serem aplicados às transformações coletadas.

#	Tratamento	Descrição
1	**RR**	Aplicação do **R**astreio de **R**egras na transformação. Fazemos a adição de uma regra auxiliar de rastreio e chamadas para essa regra para cada chamada de regras na transformação original.
2	**RH**	Aplicação do **R**astreio de **H**elpers na transformação. Fazemos a adição de uma regra auxiliar de rastreio e chamadas para essa regra para cada execução de *helpers* na transformação original.
3	**RF**	Aplicação do **R**astreio de **F**iltros na transformação. Fazemos a adição de uma regra auxiliar de rastreio e chamadas para essa regra para cada execução de filtros na transformação original.
4	**RB**	Aplicação do **R**astreio de **B**indings na transformação. Fazemos a adição de uma regra auxiliar de rastreio e chamadas para essa regra para cada execução de *bindings* na transformação original.
5	**LR**	Aplicação do **L**ogging de **R**egras na transformação. Fazemos a adição de linhas de código de *Logging* (disponíveis na linguagem QVT-O) em cada regra na transformação original.
6	**LH**	Aplicação do **L**ogging de **H**elpers na transformação. Fazemos a adição de linhas de código de *Logging* (disponíveis na linguagem QVT-O) em cada helper na transformação original.
7	**LF**	Aplicação do **L**ogging de **F**iltros na transformação. Fazemos a adição de linhas de código de *Logging* (disponíveis na linguagem QVT-O) para cada filtro existente na transformação original.

Tabela 4.10. Tratamentos aplicados aos programas QVT.

#	Instrumentação	Descrição
8	**MES**	Aplicação de **M**odificação de **E**lementos de **S**aída na transformação. Fazemos a adição de linhas de código nas regras de modo que os elementos no modelo de saída sejam modificados (ou duplicados).

Tabela 4.11. Tratamentos aplicados aos programas QVT.

Assim, apresentamos na Tabela 4.12 quais foram os tratamentos aplicados em cada transformação coletada, escolhidos de forma aleatória mas de modo que todos os tratamentos sejam utilizados ao menos uma vez.

Tratamento / Transformação	**RR**	**RH**	**RF**	**RB**	**LR**	**LH**	**LF**	**MES**
SimpleUML2RDBMS	X							
transformation-psm1						X		
outputSimqpnResults							X	
pcm2qpe								X
transformation-psm2		X						
SAMM2PCM_allocation			X					
SAMM2PCM_seff			X					
uml2simpleSTM				X				
Allocation	X							
Uml2Rdbms_Replication								X
Sofa2adl					X			

Tabela 4.12. Transformações QVT e quais tratamentos foram aplicados em cada uma.

4.6 Execução da Pesquisa

Os passos da execução do estudo, realizados pelo próprio pesquisador devido à não disponibilidade de outros sujeitos com conhecimento QVT, são enumerados a seguir.

1. Coletamos inicialmente 108 programas QVT de repositórios online. Destes, por questões de complexidade e tempo, selecionamos aleatoriamente um total de 11 programas QVT (indicados nas Tabelas 4.8 e 4.9) de diferentes tamanhos e tipos, buscando, entretanto, agrupá-los de modo que cada grupo de tamanho (da transformação) se aproxime de cada outro em número de transformações. Buscamos, dessa forma, não deixar os resultados obtidos com o estudo enviesados para transformações de um tamanho específico;
2. Para cada programa, definimos aleatoriamente (mas de modo que faça sentido na transformação) um *cross-cutting concern* que deveria ser resolvido por meio de mudanças ou adições no código (vistos nas Tabelas 4.10 e 4.11), aplicando essas mudanças em seguida (com e sem aspectos);
3. Coletamos as métricas relacionadas às perguntas de pesquisa com as duas versões de código QVT;
4. Coletamos os resultados da execução do compilador AQVT.
5. Analisamos os resultados obtidos, discutindo e respondendo às questões definidas inicialmente no estudo.

4.7 Ameaças

Consideramos as seguintes ameaças à validade da pesquisa:

- **Ameaças à validade de conclusão**: Existe a possibilidade de que a quantidade de amostras coletadas para o estudo não seja suficiente para se obter uma resposta conclusiva. Com poucas transformações analisadas, o conjunto de resultados pode ser enviesado tanto para o lado dos benefícios da linguagem quanto para os problemas. Procuramos diminuir essas ameaças ao agrupar as amostras de acordo com seu tamanho e analisá-las tanto junto como separadamente.

- **Ameaças à validade interna**: Os instrumentos utilizados podem afetar a validade interna caso o compilador não realize suas operações corretamente. Realizamos uma análise do processo de weaving ou geração do código para buscar e identificar potenciais problemas neste quesito.

- **Ameaças à validade externa**: Os programas QVT selecionados dos repositórios podem não ser bem representativos do universo de programas utilizados em projetos MDA com QVT. Com transformações muito similares, existe a possibilidade de que os resultados sejam válidos apenas para aquele conjunto específico de programas, e não para o conjunto geral de transformações QVT. Para tentar mitigar essa questão, buscamos e coletamos exemplos *open source* de diferentes projetos e com diferentes funções.

- **Ameaças à validade de construto**: Existe a possibilidade de que os resultados obtidos podem ter sido causados por variáveis que não foram consideradas no experimento. Para tentar mitigar esse problema, buscamos utilizar nas variáveis do estudo elementos de QVT de modo que a maior parte da linguagem seja coberta, seguindo o escopo definido por AQVT. Foram contemplados os seguintes elementos de AQVT: Filtros, Bindings, Helpers, Regras e suas execuções. Devido à falta de exemplos nas transformações coletadas, não foram considerados Objetos e Construtores. Além disso, utilizamos métricas (e suas adaptações) já utilizadas em outros estudos ([5], [7] e [28]), buscando aproximar o que pretendíamos medir com o que as métricas realmente mediram, como as que buscam checar a modularidade por meio da execução de regras em regras e helpers, por exemplo. Segundo esses trabalhos, essas métricas são utilizadas em diversos estudos diferentes sobre modularidade e compreensão de código com relativa aceitação.

4.8 Perfil dos Dados Coletados

Como definidas anteriormente, as métricas utilizadas no estudo são divididas em dois grupos distintos: métricas de leitura/compreensão e métricas de modularidade. Especificamos as métricas de leitura de modo que seus resultados possam se aproximar de uma possível qualidade de leitura quando comparadas entre si. De modo semelhante, as métricas de modularidade foram definidas e utilizadas para a realização de comparações entre os diferentes programas QVT e suas versões equivalentes, adicionados de aspectos.

Os *cross-cutting concerns* aplicados para cada transformação foram escolhidos de forma aleatória, mas apenas para as transformações em que fizessem sentido (um *concern* de execução de helper não faria sentido em uma transformação sem helpers, por exemplo). Esses tratamentos ou *cross-cutting concerns* foram concebidos para o estudo por meio de análise das transformações coletadas em repositórios online, onde identificamos a execução de filtros e regras em regras, por exemplo. Além disso, utilizamos *cross-cutting concerns* identificados e analizados em trabalhos como [49] e [33], onde a execução de Bindings e a geração de elementos por regras nos modelos de saída foram estudados por seus autores.

Entretanto, os tratamentos ou *concerns* que foram definidos para as transformações não cobrem a linguagem AQVT completamente, excluindo a execução de Construtores e Objetos, devido à falta desses mesmos elementos nas transformações obtidas e utilizadas no estudo.

As características das transformações utilizadas (juntamente com seu tratamento aplicado) podem ser vistas nas Tabelas 4.13 e 4.14. É importante notar que consideramos uma transformação com aspectos sendo idêntica à transformação original, ou seja, sem modificações. Fazemos a utilização desse requisito porque o código de aspectos é escrito em um módulo ou arquivo separado da transformação e, portanto, não afetará diretamente na qualidade de leitura e compreensão do código da transformação em si. O mesmo vale para a questão de modularidade: já que não há modificação direta no código, a transformação com aspectos é considerada idêntica à transformação sem modificações.

Para a qualidade de leitura e compreensão do código da transformação, as métricas adaptadas de [24] e [55] cobrem todas as construções especificadas para este estudo com AQVT (Bindings, Helpers, Regras, Filtros, Elementos de modelos), sem a utilização de Construtores e Objetos. Similarmente, as métricas que foram adaptadas de [7] buscaram refletir a modularidade do programa QVT as mesmas construções definidas para a qualidade de leitura e compreensão.

Dessa forma, o "código com aspectos" nada mais é do que o código QVT original apoiado por um outro arquivo, o programa AQVT. Estamos então comparando como as mudanças com a aplicação de requisitos nas transformações podem afetar o código normal e, assim, explicitar as vantagens da separação de *concerns* em módulos (AQVT).

Transformação	Linhas de Código	# de Regras	# de Helpers	Tratamento
SimpleUML2RDBMS	215	16	12	Para o estudo de aspectos, aplicamos o requisito de rastreio sobre todas as chamadas de regras.
transformation-psm1	1508	1	22	Para o estudo de aspectos, aplicamos o requisito de logging sobre todas as execuções dos helpers que possuem a palavra "process" em seu nome.
outputSimqpnResults	333	1	26	Para o estudo de aspectos, aplicamos o requisito de logging sobre todos os filtros aplicados em helpers na transformação

Tabela 4.13. Características e tratamentos nas transformações utilizadas.

Transformação	Linhas de Código	# de Regras	# de Helpers	Tratamento
pcm2qpe	2866	42	117	Para o estudo de aspectos, aplicamos o requisito de inclusão, em todos os elementos de saída do tipo String, um prefixo com o nome de sua respectiva regra.
transformation-psm2	943	1	16	Para o estudo de aspectos, aplicamos o requisito de rastreio sobre todos os helpers da transformação.
SAMM2PCM_allocation	38	3	0	Para o estudo de aspectos, aplicamos o requisito de rastreio sobre todos os filtros da transformação. (Disponível no Apêndice B)
SAMM2PCM_seff	195	19	8	Para o estudo de aspectos, aplicamos o requisito de rastreio sobre todas as execuções de filtros na regra "toRDBehaviour".
uml2simpleSTM	39	2	1	Para o estudo de aspectos, aplicamos o requisito de rastreio sobre todos os bindings executados na transformação.
Allocation	258	9	4	Para o estudo de aspectos, aplicamos o requisito de rastreio sobre a chamada de regras em regras na transformação.
Uml2Rdbms_Replication	520	23	2	Para o estudo de aspectos, aplicamos o requisito de duplicação de elementos no modelo de saída.
Sofa2adl	168	22	0	Para o estudo de aspectos, aplicamos o requisito de inclusão de logging em cada regra.

Tabela 4.14. Características e tratamentos nas transformações utilizadas.

4.9 Leitura e Compreensão

Aplicando as métricas de Leitura e Compreensão para as transformações QVT utilizadas no estudo, com a aplicação de aspectos obtivemos os resultados da Tabela 4.15, onde "CA" representa o valor obtido sobre programas **C**om **A**spectos e "SA" representa o valor obtido com programas **S**em **A**spectos.

Transformação	**LDC**		**MLDCR**		**MLDCH**		**MCR**		**Leitura**	
	CA	**SA**	**CA**	**SA**	**CA**	**SA**	**CA**	**SA**	**CA**	**SA**
SimpleUML2RDBMS	215	238	6.9	7.8	3.7	3.7	0.3	0.29	0.11	0.12
Transformation-psm1	1508	1512	21	25	13.2	13.2	0	0	0.44	0.5
outputSimqpnResults	333	337	14	18	9.8	9.8	0	0	0.22	0.27
pcm2qpe	2866	2887	27.2	28	8.17	8.13	0.07	0.07	0.58	0.59
transformation-psm2	943	967	14	11.5	24.1	24.1	0	0	0.43	0.4
SAMM2PCM_allocation	38	47	5	6	0	0	0	0	0	0.01
SAMM2PCM_seff	195	210	4.4	4.8	4.9	4.9	0	0	0.08	0.08
uml2simpleSTM	39	59	8.5	10.7	8	6.5	0	0	0.12	0.13
Allocation	258	268	8.8	8.9	23.7	23.7	2.2	2	0.56	0.53
Uml2Rdbms_Replication	520	543	10.7	11.7	12	12	0.22	0.22	0.25	0.27
Sofa2adl	168	195	6.4	7.4	0	5	0	0	0.03	0.09

Tabela 4.15. Resultados das aplicações de métricas de leitura em código com e sem aspectos.

Já na Tabela 4.16, temos os valores de cada uma das quatro métricas iniciais após a normalização de seus valores (de 0 a 1), com os números destacados em azul indicando que houve uma melhora com o uso de Aspectos, enquanto que valores destacados em vermelho apontam uma piora para o uso de AQVT.

Transformação	LDC		MLDCR		MLDCH		MCR		Leitura	
	CA	SA	CA	SA	CA	SA	CA	SA	CA	SA
SimpleUML2RDBMS	0.06	0.07	0.08	0.12	0.15	0.15	0.14	0.13	0.11	0.12
Transformation-psm1	0.52	0.52	0.7	0.87	0.55	0.55	0	0	0.44	0.5
outputSimqpnResults	0.1	0.1	0.39	0.57	0.41	0.41	0	0	0.22	0.27
pcm2qpe	0.99	1	0.97	1	0.34	0.34	0.03	0.03	0.58	0.59
transformation-psm2	0.32	0.33	0.39	0.28	1	1	0	0	0.43	0.4
SAMM2PCM_allocation	0	0	0	0.04	0	0	0	0	0	0.01
SAMM2PCM_seff	0.06	0.06	0.02	0.03	0.24	0.24	0	0	0.08	0.08
uml2simpleSTM	0	0.01	0.15	0.25	0.33	0.27	0	0	0.12	0.13
Allocation	0.08	0.08	0.17	0.17	0.98	0.98	1	0.91	0.56	0.53
Uml2Rdbms_Replication	0.17	0.18	0.25	0.29	0.5	0.5	0.1	0.1	0.25	0.27
Sofa2adl	0.05	0.06	0.06	0.1	0	0.21	0	0	0.03	0.09

Tabela 4.16. Resultados normalizados das métricas de leitura em código com e sem aspectos.

A Figura 4.1 revela os dados normalizados para comparação em um gráfico de barras agrupado, onde os resultados com aspectos são identificados pelas barras de cor mais clara, enquanto que os resultados sem aspectos aparecem representados por barras de tom mais escuro. Nessa representação, pode-se notar que, apesar da pequena diferença entre cada par de resultados, o grupo que utilizou aspectos saiu-se um pouco melhor na questão de qualidade de leitura dos programas.

Contudo, praticamente nenhum valor passou de 0.5, com a maioria dos dados sendo bem mais próximos do valor 0, podendo indicar a necessidade de utilização de mais transformações no estudo.

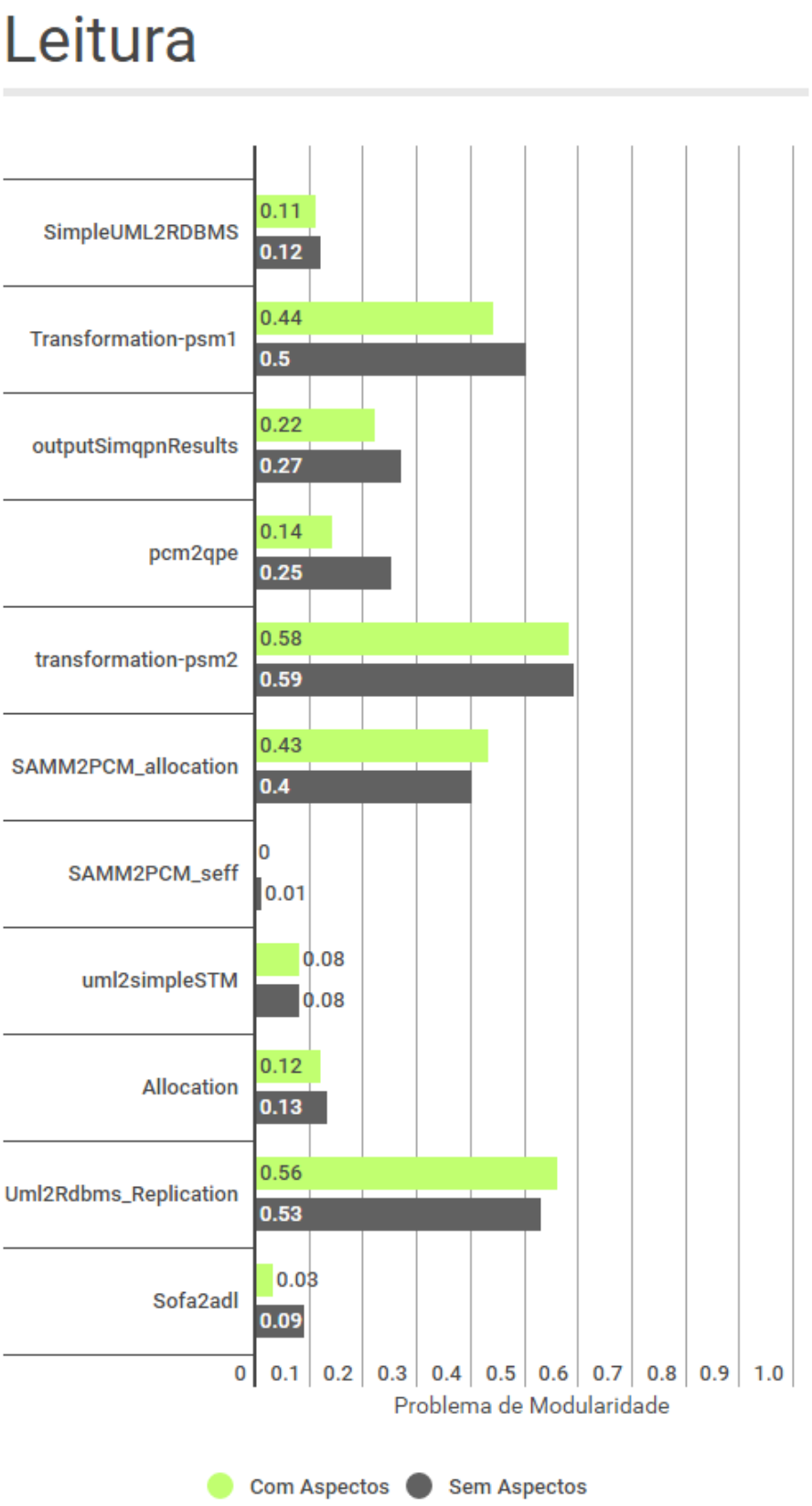

Figura 4.1. Gráfico de barras para a métrica de leitura.

Analisando-se alguns desses resultados individualmente, temos que:

SimpleUML2RDBMS - Esta transformação obteve um grau de leitura de **0.11** quando aplicada com aspectos e **0.12** sem aspectos. Essa pequena diferença se deu principalmente devido à quantidade de helpers e regras existentes na transformação, que acabaram tornando suaves as diferenças entre as outras quatro métricas estudadas (*LDC*, *MLDCR*, *MLDCH* e *MCR*). Os valores iguais para *MLDCH* se devem ao fato de nenhuma das opções envolver a criação ou modificação de um ou mais helpers no código. Apenas as regras desta transformação foram afetadas, como pode ser observado nos resultados de *MLDCR* e *MCR*. A quantidade de linhas de código cresceu para o programa sem aspectos devido à inclusão de uma nova regra e chamadas para a mesma no código. Além disso, por causa da inclusão de uma nova regra sem condicionais, sua média em regras diminuiu para a transformação implementada sem aspectos.

pcm2qpe - Como a maior transformação avaliada neste trabalho, esta transformação possui os maiores valores para o grau de leitura, com **0.58** para sua versão com aspectos e **0.59** para a versão sem a aplicação de aspectos. É possível claramente notar que, por causa do seu tamanho e sua quantidade de regras e helpers, as diferenças nos resultados obtidos com as métricas são extremamente sutis. Além disso, o requisito aplicado para esta transformação (a inclusão de prefixos em *Strings* dos modelos de saída), é baseado fortemente na adição de linhas de código em regras, o que causou os efeitos vistos na Tabela 4.17 (**0.99** para **1** e **0.97** para **1**).

SAMM2PCM_allocation - Em contraste com "pcm2qpe", esta transformação foi o menor programa avaliado neste trabalho, fato que é refletido nos resultados obtidos por meio das métricas, onde o valor da métrica de leitura foi de **0** para o programa com aspectos e **0.01** para sua versão sem aspectos. Com os menores valores obtidos para as métricas *LDC, MLDCR, MLDCH* e *MCR*, a normalização dos dados fez com que a maioria dos seus valores se tornassem **0**. Entretanto, como explicado anteriormente para outras transformações com poucas regras e *helpers*, as métricas podem ser bastante afetadas até com pequenas mudanças em um ou mais desses elementos de transformação.

Allocation - Esta transformação obteve um grau de leitura de **0.56** quando aplicada com aspectos e **0.53** sem aspectos. A diferença, negativa e pequena, foi resultado da inclusão de uma nova regra no código da transformação, sem a utilização de um condicional. Por causa disso, a média geral de condicionais nas regras da transformação sem aspectos diminuiu. Consequentemente, o valor de Leitura também diminuiu. Os valores *LDC, MLDCR* e *MLDCH* se mantiveram praticamente constantes, devido ao tamanho do programa (que acabou minimizando os efeitos de uma mudança pequena).

Uml2Rdbms_Replication - O requisito aplicado para esta transformação (duplicação de elementos no modelo de saída) fez com que a versão da transformação sem aspectos necessitasse da adição de código em suas regras apenas, sem modificar sua quantidade de regras e helpers, resultando em um valor final de **0.25** para o programa auxiliado por aspectos e **0.27** para o programa com código QVT puro. Como não houve mudanças em helpers, *MLDCH* se manteve constante.

4.9.1 Análise Estatística

Com os valores da métrica de Leitura e Compreensão obtidos, verificamos o QQPlot e o histograma desses dados, vistos na Figura 4.2. Além disso, verificamos sua normalidade por meio dos testes de Shapiro-Wilk [48] e Anderson-Darling [1], com seus resultados apresentados na Figura 4.3.

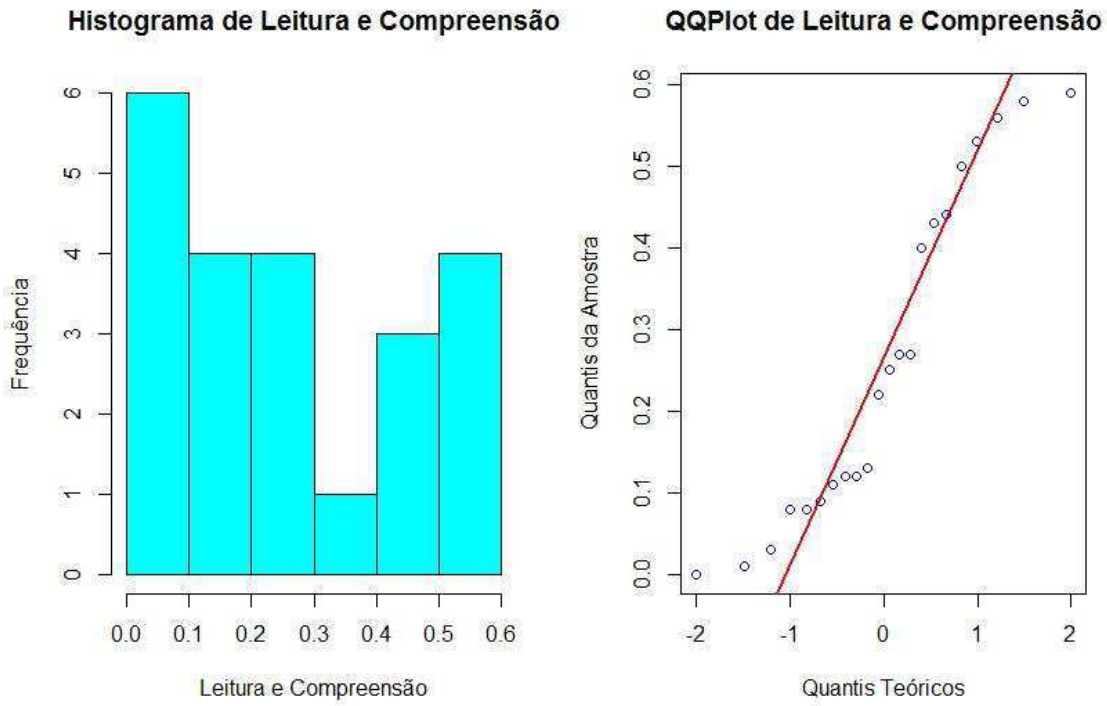

Figura 4.2. Histograma e QQPlot da métrica “Leitura e Compreensão”.

```
        Shapiro-Wilk normality test

data:  data$LeituraECompreensao
W = 0.89671, p-value = 0.02555


        Anderson-Darling normality test

data:  data$LeituraECompreensao
A = 0.82667, p-value = 0.02746
```

Figura 4.3. Resultados dos testes Shapiro-Wilk e Anderson-Darling para a métrica de Leitura e Compreensão.

Observando os resultados das Figuras 4.2 e 4.3, podemos verificar que os dados da métrica de Leitura e Compreensão provavelmente não seguem uma distribuição normal, já que as hipóteses nulas dos testes (distribuição normal) foram rejeitadas e tanto o histograma quanto o QQPlot dos dados possuem um claro viés. Assim, para responder às hipóteses de pesquisa, devemos utilizar testes não paramétricos.

Considerando-se que estamos tentando comparar dois grupos dependentes de dados para uma distribuição não normal, utilizamos o Wilcoxon test [57] para a hipótese nula "igual a" e a alternativa "menor que 0" (Figura 4.4), ou seja, verificamos se os resultados da métrica de Leitura e Compreensão são menores para o experimento **Com Aspectos** do que para o experimento **Sem Aspectos**. Obtivemos então um p-value de **0.06445** e, apesar de ser um valor muito baixo, não é suficiente para rejeitarmos a hipótese nula de que os dois grupos são "iguais".

```
        Exact Wilcoxon-Pratt Signed-Rank Test

data:  y by x (pos, neg)
         stratified by block
Z = -1.559, p-value = 0.06445
alternative hypothesis: true mu is less than 0
```

Figura 4.4. Resultado do Wilcoxon test para a métrica de Leitura e Compreensão.

Comparando-se os resultados com uma visão geral de todos, é possível notar que, apesar da diferença ser pequena, os códigos com aspectos parecem apresentar uma melhor qualidade de leitura e compreensão de código. Entretanto, notamos, com a análise dos valores obtidos pelas métricas, que ocorre uma variação grande para as métricas *MLDCR* e *MLDCH* quando a quantidade de regras ou **helpers** se aproxima de **0** na transformação. Isso ocorre devido ao uso de

médias, que variam muito quando são aplicadas sobre pequenos valores. Além disso, a análise da complexidade do código apenas com os condicionais *When* e *Where* mostrou-se, de certa forma, ineficaz, visto que até mesmo as grandes transformações estudadas fazem utilização escassa desse tipo de elemento. Uma possível análise mais robusta poderia ser feita com o agrupamento de transformações de acordo com seu número de regras e *helpers*.

4.10 Modularidade

Aplicando as métricas de Modularidade para as transformações QVT utilizadas no estudo, com a aplicação de aspectos obtivemos os resultados da Tabela 4.17, onde "CA" representa o valor obtido sobre programas **C**om **A**spectos e "SA" representa o valor obtido com programas **S**em **A**spectos. A Tabela 4.18 mostra os resultados com os valores das métricas MCRR e MCHR normalizados (de 0 a 1).

Transformação	**MaxCRR**		**MaxCHR**		**MCRR**		**MCHR**		**Modularidade**	
	CA	**SA**	**CA**	**SA**	**CA**	**SA**	**CA**	**SA**	**CA**	**SA**
SimpleUML2RDBMS	0.25	0.44	0.1	0.1	1.25	2.1	0.83	0.83	0.23	0.35
Transformation-psm1	0	1	0.5	0.5	0	0.5	0.09	0.09	0.13	0.42
outputSimqpnResults	0	0	0.33	0.33	0	0	0.07	0.07	0.08	0.08
pcm2qpe	0.03	0.03	0.21	0.4	0.76	0.76	0.43	0.72	0.14	0.25
transformation-psm2	0	1	0.33	0.33	0	1.5	0.18	0.18	0.09	0.47
SAMM2PCM_allocation	0.5	0.6	0	0	0.67	1.25	0	0	0.18	0.25
SAMM2PCM_seff	0.41	0.3	0.28	0.28	0.89	1.15	1.75	1.75	0.33	0.33
uml2simpleSTM	1	0.89	1	0.89	0.5	3	0.33	4.5	0.56	0.92
Allocation	0.25	0.5	0.37	0.37	0.44	0.8	2	2	0.29	0.38
Uml2Rdbms_Replication	0.12	0.12	0.5	0.5	0.74	0.74	1	1	0.27	0.27
Sofa2adl	0.18	0.18	0	1	1.23	1.23	0	5	0.15	0.65

Tabela 4.17. Resultados das aplicações de métricas de modularidade em código com e sem aspectos.

Transformação	MaxCRR		MaxCHR		MCRR		MCHR		Modularidade	
	CA	SA	CA	SA	CA	SA	CA	SA	CA	SA
SimpleUML2RDBMS	0.25	0.44	0.1	0.1	0.42	0.7	0.17	0.17	0.23	0.35
Transformation-psm1	0	1	0.5	0.5	0	0.17	0.02	0.02	0.13	0.42
outputSimqpnResults	0	0	0.33	0.33	0	0	0.01	0.01	0.08	0.08
pcm2qpe	0.03	0.03	0.21	0.4	0.25	0.25	0.09	0.14	0.14	0.25
transformation-psm2	0	1	0.33	0.33	0	0.5	0.04	0.04	0.09	0.47
SAMM2PCM_allocation	0.5	0.6	0	0	0.22	0.42	0	0	0.18	0.25
SAMM2PCM_seff	0.41	0.3	0.28	0.28	0.3	0.38	0.35	0.35	0.33	0.33
uml2simpleSTM	1	0.89	1	0.89	0.17	1	0.07	0.9	0.56	0.92
Allocation	0.25	0.5	0.37	0.37	0.15	0.27	0.4	0.4	0.29	0.38
Uml2Rdbms_Replication	0.12	0.12	0.5	0.5	0.25	0.25	0.2	0.2	0.27	0.27
Sofa2adl	0.18	0.18	0	1	0.41	0.41	0	1	0.15	0.65

Tabela 4.18. Resultados normalizados das métricas de modularidade em código com e sem aspectos.

A Figura 4.5 revela os dados normalizados para comparação em um gráfico de barras agrupado, onde os resultados com aspectos são identificados pelas barras de cor mais clara, enquanto que os resultados sem aspectos aparecem representados por barras de tom mais escuro. Nessa representação para os resultados de métrica de modularidade, pode-se notar que existe uma pequena divergência para mais no grupo que não utilizou aspectos.

Comparando esses dados com os da qualidade de leitura, temos que a diferença entre os grupos com e sem aspectos para modularidade foi bem maior.

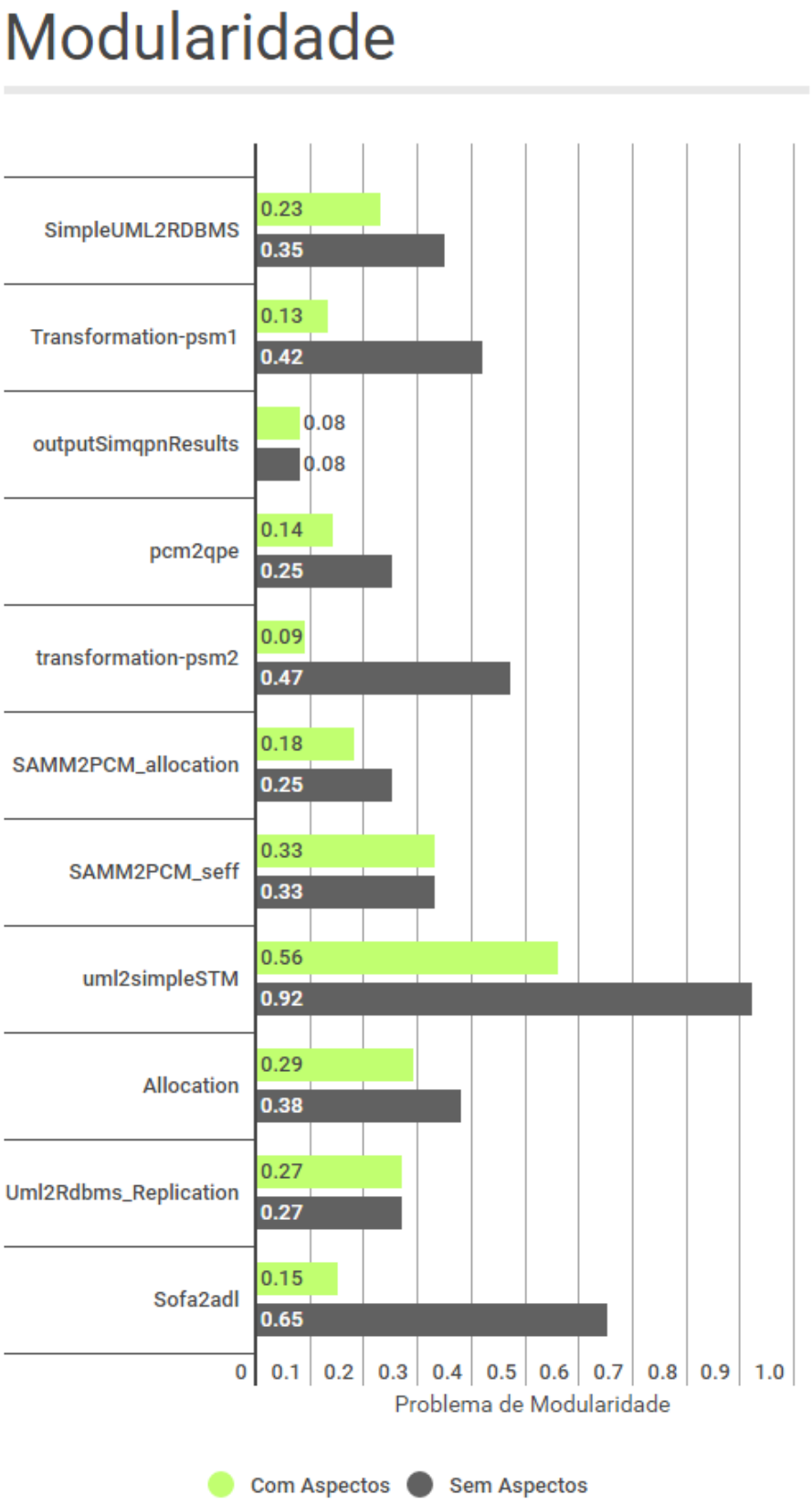

Figura 4.5. Gráfico de barras para a métrica de modularidade.

Analisando-se alguns desses resultados individualmente, temos que:

SimpleUML2RDBMS - Esta transformação obteve um grau de modularidade de **0.23** quando aplicada com aspectos e **0.35** sem aspectos. Os valores de **0.25** para *MaxCRR* com aspectos e **0.44** sem aspectos podem ser explicados devido à quantidade de chamadas a uma nova regra que foi adicionada no código sem aspectos, com o objetivo de auxiliar na coleta de dados de execução das outras regras. Da mesma forma, MCRR, com o aumento na quantidade de chamadas de regras, teve um resultado pior para a transformação sem aspectos (**0.7** contra **0.42**). Como o requisito aplicado sobre a transformação não envolveu a modificação ou criação de helpers, os valores de *MaxCHR* e *MCHR* se mantiveram constantes.

pcm2qpe - O grau de modularidade coletado para a versão desta transformação sem aspectos foi de **0.25**, enquanto que sua versão com aspectos revelou um grau de **0.14**. Como apresentado anteriormente, esta foi a maior transformação avaliada neste trabalho e, principalmente por causa desse fato, as diferenças entre as métricas com e sem aspectos são pequenas. Podemos entender que, por causa do seu tamanho, apenas mudanças drásticas com relação aos seus *helpers* e suas regras possam afetar de forma clara os resultados das métricas aplicadas. Além disso, o requisito aplicado para esta transformação (a inclusão de prefixos em Strings dos modelos de saída), incluiu várias chamadas de helpers em regras na versão sem aspectos, o que afetou seu *MaxCHR* (**0.4** contra **0.21** com aspectos) e *MCHR* (**0.14** contra **0.09** com aspectos).

SAMM2PCM_seff - O resultado geral da métrica de modularidade se manteve igual para as versões do programa com e sem aspectos (**0.33**). Como o requisito aplicado para a transformação (rastreio sobre todas as execuções de filtros na regra "toRDBehaviour") incluiu uma quantidade pequena de chamadas de execução de regra em regra (seis), o valor de *MaxCRR* para aspectos foi de **0.41** enquanto que o programa sem aspectos resultou em um valor de **0,3**. Houve uma diminuição na porporção de chamadas de regras em regras (se comparado com o código original da transformação, que recebe aspectos), levando-se em conta as regras que mais são utilizadas por outras regras na transformação (*MCRR* de **0.3** para aspectos e **0.38** para o programa sem aspectos).

uml2simpleSTM - O resultado do índice de modularidade obtido para o programa com aspectos (**0.56**) foi melhor do que o índice para a transformação sem aspectos (**0.92**). Essa grande diferença pôde ser observada devido ao pequeno número de chamadas de helpers e regras na transformação, o que resultou em diferenças especialmente grandes para as métricas de médias quando os requisitos do estudo foram aplicados à transformação (inclusão de várias chamadas de regras e helpers no programa). *MCRR* obteve **0.17** para aspectos e o valor **1** para o programa sem aspectos, enquanto que *MCHR* revelou **0.07** e **0.9**.

Sofa2adl - Como uma transformação que inicialmente não possuía helpers, a inclusão de um, para auxiliar no *logging* de execução de regras, fez com que o valor de *MaxCHR* pulasse de **0**, no programa com aspectos, para **1** em um programa sem aspectos. Da mesma forma, *MCHR* também foi de **0** (com aspectos) para **1** (sem aspectos), devido à quantidade de chamadas a helpers que foram adicionadas. Tal ocorrência afetou o resultado do valor final de modularidade, que ficou em **0.15** para aspectos e **0.65** para a versão sem aspectos.

4.10.1 Análise Estatística

Com os valores da métrica de Modularidade obtidos, verificamos o QQPlot e o histograma desses dados, vistos na Figura 4.6. Além disso, verificamos sua normalidade por meio dos testes de Shapiro-Wilk e Anderson-Darling, com seus resultados apresentados na Figura 4.7.

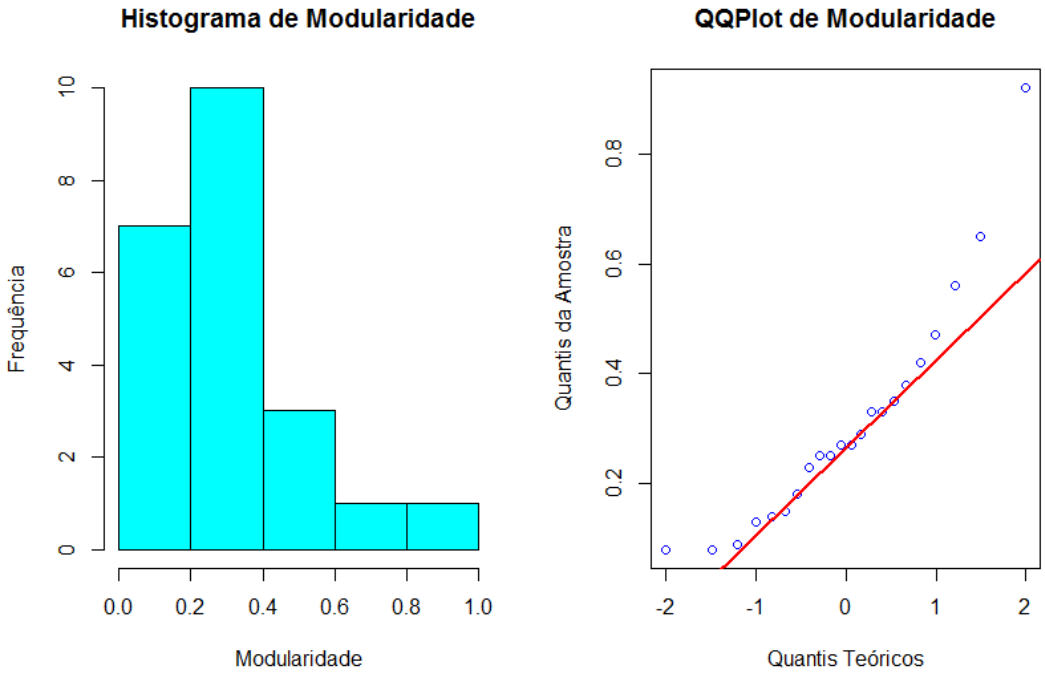

Figura 4.6. Histograma e QQPlot da métrica “Modularidade”.

```
        Shapiro-Wilk normality test

data:  data$Modularidade
W = 0.88208, p-value = 0.01324

        Anderson-Darling normality test

data:  data$Modularidade
A = 0.72773, p-value = 0.04923
```

Figura 4.7. Resultados dos testes Shapiro-Wilk e Anderson-Darling para a métrica de Modularidade.

Observando os resultados das Figuras 4.6 e 4.7, podemos verificar que os dados da métrica de Modularidade provavelmente não seguem uma distribuição normal, já que as hipóteses nulas dos testes (distribuição normal) foram rejeitadas e tanto o histograma quanto o QQPlot dos dados possuem um claro viés. Assim, para responder às hipóteses de pesquisa, devemos utilizar testes não paramétricos.

Considerando-se que estamos tentando comparar dois grupos dependentes de dados para uma distribuição não normal, utilizamos o Wilcoxon test para a hipótese nula "igual a" e a alternativa "menor que 0" (Figura 4.8), ou seja, verificamos se os resultados da métrica de Modularidade são menores para o experimento **Com Aspectos** do que para o experimento **Sem Aspectos**. Obtivemos então um p-value de **0.003906**, o suficiente para rejeitarmos a hipótese nula de que os dois grupos são "iguais" e considerarmos que o valor de Modularidade para o grupo **Com Aspectos** é melhor (valores menores indicam melhor modularidade), se comparado com o grupo **Sem Aspectos**.

```
        Exact Wilcoxon-Pratt Signed-Rank Test

data:  y by x (pos, neg)
        stratified by block
Z = -2.705, p-value = 0.003906
alternative hypothesis: true mu is less than 0
```

Figura 4.8. Resultado do Wilcoxon test para a métrica de Modularidade.

Com uma visão geral dos dados obtidos e comparando-os, é possível notar que existe uma diferença entre a modularidade de programas que usaram aspectos para os que não utilizaram tal abordagem, com os programas de aspectos auxiliando na modularidade das transformações. Entretanto, notamos, com a análise dos valores obtidos pelas métricas, que pode ocorrer uma variação grande para as métricas utilizadas quando a quantidade de regras ou *helpers* se aproxima

de **0** na transformação. Isso ocorre devido ao uso de médias, que variam muito quando são aplicadas sobre pequenos valores. Uma possível análise mais robusta poderia ser feita com o agrupamento de transformações de acordo com seu número de regras e *helpers*.

Capítulo 5

Trabalhos Relacionados

A análise feita por [33] buscou revelar e compreender os problemas de *cross-cutting* em transformações entre modelos, utilizando-se de programas escritos na linguagem QVT para realizar seu estudo de caso. Com o auxílio de grafos de dependência sobre as transformações, obtidos a partir da análise de suas regras e da relação dos seus elementos de entrada e saída, o autor identificou pontos em que ocorre *cross-cutting* de problemas (segurança e persistência). É proposto no trabalho que as informações de rastreio da execução das transformações podem ser utilizadas para gerar grafos de dependência, identificando problemas de *cross-cutting*. O autor identifica que o *concern* mais relevante para o estudo foi o de rastreio da execução de regras e os elementos dos modelos de entrada e saída relacionados a cada uma delas. Entretanto, não é apresentada nem referenciada uma solução para tais problemas, apenas uma forma de identificá-los mais facilmente. Além disso, o autor não especifica ou sugere uma nova linguagem de aspectos e *weaver* que poderia ser aplicada para transformações, focando-se apenas na questão de como as transformações podem ser afetadas.

Tecnologias orientadas a aspectos podem ser utilizadas para auxiliar na separação de *cross-cutting concerns* de outras funcionalidades em um projeto, segundo [49]. Para isso, seu trabalho propôs um *Framework* para a separação dos *cross-cutting concerns* em módulos, ou seja, aspectos. Representados em forma de modelos, eles podem então ser utilizados normalmente em transformações de modelo, passando de um aspecto ou modelo independente de plataforma (PIM) para um dependente de plataforma (PSM). Os autores identificam nas transformações utilizadas no trabalho concerns como execução de regras e bindings de seus elementos. Não há, contudo, uma busca ou proposta para a resolução de possíveis problemas de *cross-cutting* nas próprias transformações utilizadas no *Framework*. De modo similar, [31] faz a proposta de um Framework que relaciona questões de Programação Orientada a Aspectos e MDA, mas apenas incorpora esses conceitos, como a equivalência entre *pointcuts* e *queries* em QVT. Ambos, entretanto, não especificam uma linguagem de aspectos nem *weaver* para a própria linguagem QVT, sendo o foco dos estudos os conceitos modelados com a linguagem, e não sua estrutura.

O estudo realizado em [53] traz conclusões sobre as possíveis classificações de High-Order Transformations (HOTs), que são transformações aplicadas sobre outros programas de transformação. Uma dessas classificações envolve a adição de código em transformações, mais especificamente o *weaving* com a inclusão de *cross-cutting concerns* no programa, como *debugging* e rastreio. É possível projetar, segundo o trabalho, uma transformação que seja completamente independente da lógica da transformação inicial, facilitando o reuso da HOT em outras transformações. Finalmente, os autores mencionam que, em linguagens onde não existe um suporte nativo a aspectos, um mecanismo orientado a aspectos poderia ser considerado como uma HOT que é aplicada no momento em que o programa original é executado. A ideia geral de aplicação de HOTs como forma de *weaving* de aspectos foi também apresentada, similarmente, em [23]. Contudo, apesar de mencionar a possibilidade de resolução do problema de rastreio por meio do uso de aspectos, os autores não vão além disso, e não especificam nenhuma linguagem nova.

Utilizando-se de grafos de dependência sobre transformações, [56] faz uma análise de possíveis problemas de *cross-cutting* nesses programas, propondo uma definição de *cross-cutting* que deverá envolver ao menos dois níveis ou domínios diferentes e uma relação entre eles (por exemplo, as possíveis entradas e saídas de uma transformação). Entretanto, apesar de definir *cross-cutting concerns* (como rastreio de transformações), o trabalho não apresenta uma solução para esses problemas, como o desenvolvimento de uma linguagem de aspectos para QVT realizamos.

O *Framework* desenvolvido por [28] faz uma identificação e separação de restrições OCL, ou regras de definição aplicadas em modelos, que passam por *cross-cutting* em transformações de modelos, encapsulando os resultados na forma de aspectos. O trabalho utiliza-se de um Sistema de Transformação e Modelagem Visual (VMTS) para a definição de transformações e a aplicação de restrições OCL, estas controladas pelo *Framework* por meio de uma série de algoritmos que identificam, extraem e realizam o weaving dos aspectos obtidos com as restrições. Ou seja, os autores utilizam a linguagem OCL para incluir restrições na transformação e controlam com o *Framework* as restrições que se cruzam no código. Os cross-cutting concerns em OCL mais relevantes que foram identificados no trabalho foram os de definição de *Profiles* UML na transformação. Apesar de definirem uma linguagem simples para ser utilizada na aplicação de aspectos, isso não é feito na trasformação como um todo, apenas no código escrito na linguagem OCL, diferentemente da utilização de uma linguagem de aspectos nova e um weaver para o código dos programas QVT com aspectos.

Capítulo 6

Considerações Finais

Este trabalho teve como objetivo principal o desenvolvimento de uma linguagem de aspectos para a linguagem de transformação QVT, provendo auxílio no desenvolvimento desses programas com relação aos *cross-cutting concerns* que podem ocorrer em seu código. AQVT, ou Aspectos para QVT, permite a definição de aspectos para a separação desses *concerns,* como logging e rastreio, utilizando-se de construções específicas do paradigma de programação orientada a aspectos.

A linguagem AQVT possui uma gramática concreta, definida e escrita com o auxílio do framework Xtext, com as construções específicas para o Paradigma de Aspectos (*Aspect, Pointcut*, *Joinpoint* e *Advice*) e elementos próprios de QVT, como Regra, Helper, Binding e elementos de modelos de entrada e saída. A implementação do compilador e o *Weaving* da linguagem foi também realizada com o auxílio de Xtext e a linguagem Java de programação. O compilador realiza a análise tanto do código QVT quanto do código de aspectos, retirando as informações necessárias dos elementos de cada programa para então gerar um programa QVT único com a adição das funcionalidades definidas pelo programa AQVT.

Foi realizado um estudo empírico com diversas transformações disponíveis e coletadas de repositórios online, tanto de projetos-exemplo do Eclipse como projetos reais (Q-ImPrESSQ, entre outros). Após sua coleta, realizamos uma seleção dos programas de forma semi-aleatória, levando em conta o tamanho de cada programa, de modo que o estudo não se concentrasse apenas em transformações muito pequenas ou grandes. Em seguida, definimos *cross-cutting concerns* para aplicarmos em cada transformação, buscando resolvê-los tanto com código puro QVT quanto com o código AQVT. Esses *concerns* foram identificados tanto nas transformações coletadas quanto em trabalhos já realizados sobre *cross-cutting concerns* em transformações QVT. Com a utilização de AQVT, foi possível obter uma melhora na legibilidade do código QVT e sua modularidade, se comparado com a versão equivalente do programa escrita puramente em QVT.

Entretanto, o desenvolvimento de algumas operações se mostrou bastante complexo, e AQVT ainda não contempla todas as especificações do paradigma de Aspectos, como a definição de *Advices* com *Around*, e alguns elementos de QVT, como execução de Construtores e

inicialização de Objetos. Além disso, não foi possível realizar uma avaliação mais aprofundada da linguagem, com a participação de sujeitos utilizando a linguagem na pesquisa.

Como trabalhos futuros, pretendemos realizar a conclusão da linguagem, com o desenvolvimento das funcionalidades ausentes, e estendê-la para todas as formas da linguagem QVT (Core e Relations). Para o compilador, pretendemos analisar a possibilidade de uso de técnicas formais para verificarmos sua corretude. Adicionalmente, para obtermos mais informações sobre a utilização da linguagem por desenvolvedores e então avaliá-la nesse contexto, um estudo e um questionário envolvendo sujeitos deverá ser realizado.

Bibliografia

[1] ANDERSON, T. W.; DARLING, D. A. **Asymptotic theory of certain "goodness-of-fit" criteria based on stochastic processes**. Annals of Mathematical Statistics 23, pp. 193-212, 1952.

[2] BARTSCH, M.; HARRISON, R. **An exploratory study of the effect of aspect-oriented programming on maintainability**. Software Quality Journal, Volume 16, Issue 1, pp 23-44, 2008.

[3] BAUSE, F. **Queueing Petri Nets - A formalism for the combined qualitative and quantitative analysis of systems**. Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on.

[4] BECKER, S.; KOZIOLEK, H.; REUSSNER, R. **The Palladio component model for model-driven performance prediction**. Journal of Systems and Software, Volume 82, Issue 1, pp 3-22, 2009.

[5] BOULANGER, J.L. **Static Analysis of Software: The Abstract Interpretation**. Wiley-ISTE, 1st ed. 2011.

[6] BURES, T.; HNETYNKA, P.; PLASIL, F. **SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model**. Proceedings of SERA 2006, Seattle, USA, IEEE CS, ISBN 0-7695-2656-X, pp.40-48, 2006.

[7] CHIDAMBER, S. R.; KEMERER, C. F. **A metrics suite for object oriented design**. IEEE Transactions on Software Engineering, vol. 20, no. 6, 1994.

[8] COLYER, A.; CLEMENT, A.; HARLEY, G.; WEBSTER, M. **Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools**. Addison-Wesley Professional, 1st edition, 2004.

[9] ECLIPSE - **Eclipse IDE**. Disponível em <https://eclipse.org/downloads/> Acesso em agosto de 2016.

[10] ECLIPSE ATL - **ATL, a model transformation technology**: <https://eclipse.org/atl/> Acesso em agosto de 2016.

[11] ECLIPSE AspectJ - **The AspectJ Programming Guide**. Disponível em <https://eclipse.org/aspectj/doc/next/progguide/index.html> Acesso em agosto de 2016.

[12] ECLIPSE Epsilon - **Epsilon Project**. Disponível em < http://www.eclipse.org/epsilon/> Acesso em agosto de 2016.

[13] ECLIPSE MOFScript - **MOFScript**. Disponível em < https://eclipse.org/gmt/mofscript/> Acesso em agosto de 2016.

[14] ECLIPSE QVTo - **GIT Eclipse QVTo Repository**: <http://git.eclipse.org/c/mmt/org.eclipse.qvto.git> Acesso em agosto de 2016.

[15] ECLIPSE QVTo - **QVT Operational**: <https://projects.eclipse.org/projects/modeling.mmt.qvt-oml> Acesso em agosto de 2016.

[16] ECLIPSE Xtend - **Xtend**. Disponível em <http://www.eclipse.org/xtend/> Acesso em agosto de 2016.

[17] ECLIPSE Xtext - **Xtext**. Disponível em <https://eclipse.org/Xtext/> Acesso em agosto de 2016.

[18] EDCUBED. **NOVA – Payment System.** Disponível em: <http://www.omg.org/mda/mda_files/EDCubed%20postgirot%20bankstory.htm> Acesso em agosto de 2016.

[19] FOWLER, M. **UML Distilled: A Brief Guide to the Standard Object Modeling Language**. Addison-Wesley Professional, 3rd ed. 2003.

[20] GOSLING, J.; JOY, B.; STEELE Jr, G. L.; BRACHA, G. BUCKLEY, A. **The Java Language Specification, Java SE 8 Edition**. Addison-Wesley Professional, 1st ed. 2014.

[21] GROVES, M. D. **AOP in .NET: Practical Aspect-Oriented Programming**. Manning Publications, 1st ed. 2013.

[22] GUILLARD, F. **SmartQVT**. Disponível em <https://sourceforge.net/projects/smartqvt/> Acesso em agosto de 2016.

[23] JOUAULT, F. **Loosely Coupled Traceability for ATL**. In Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, 2004.

[24] KAPOVÁ, L.; GOLDSCHMIDT, T.; BECKER, S.; HENSS, J. **Evaluating maintainability with code metrics for model-to-model transformations**. QoSA'10 Proceedings: research into Practice - Reality and Gaps, pp 151-166, 2010.

[25] KERNIGHAN, B. W.; RITCHIE, D. M. **The C Programming Language**. Prentice Hall, 2nd ed. 1988.

[26] KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V. et al. **Aspect-oriented programming**. 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings.

[27] KLATT, B. **PCM Event-Based Communication**. Disponível em <https://sdqweb.ipd.kit.edu/wiki/PCM_Event-Based_Communication> Acesso em agosto de 2016.

[28] LENGYEL, L.; LEVENDOVSZKY, T.; CHARAF, H. **Identification of *Cross-cutting concerns* in Constraint-Driven Validated Model Transformations**. In: Proceedings of the Third Workshop on Models and Aspects, ECOOP 2007.

[29] LOCKHEED MARTIN AERONAUTICS. **The F-16 Modular Mission Computer Application Sotware**. Disponível em: <http://www.omg.org/mda/mda_files/LockheedMartin.pdf> Acesso em agosto de 2016.

[30] MEIER, P.; KOUNEV, S.; KOZIOLEK, H. **PCM2QPN**. Disponível em <https://sdqweb.ipd.kit.edu/wiki/PCM2QPN> Acesso em agosto de 2016.

[31] MELLOR, S. J. **A Framework for Aspect-Oriented Modeling**. Workshop on Aspect-Oriented Modelling with UML (UML 2003).

[32] MELLOR, S. J.; SCOTT, K.; UHL, A.; WEISE, D. **MDA Distilled**. Addison-Wesley Professional, 2004.

[33] NGUYEN, H. Q. **Analysis of *Cross-cutting concerns* in QVT-based Model Transformations**. Master's Thesis, Computer Science MSc - EEMCS: Electrical Engineering, Mathematics and Computer Science, University of Twente, Netherlands, 2006.

[34] OMG - **Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification**. Disponível em <http://www.omg.org/spec/QVT/1.3> Acesso em agosto de 2016.

[35] OMG - **MDA Specification**: <http://www.omg.org/mda/specs.htm> Acesso em agosto de 2016.

[36] OMG MOF - **Meta Object Facility (MOF) Core**: <http://www.omg.org/spec/MOF/> Acesso em agosto de 2016.

[37] OMG MOF - **Model to Text Transformation Language (MOFM2T)**: < http://www.omg.org/spec/MOFM2T/1.0/> Acesso em agosto de 2016.

[38] OMG OCL - **Object Constraint Language**: < http://www.omg.org/spec/OCL/> Acesso em Agosto de 2016.

[39] OMG QVT - **OMG Formal Versions Of QVT**: <http://www.omg.org/spec/QVT/> Acesso em agosto de 2016.

[40] OMG UML - **OMG Formal Versions of UML**: <http://www.omg.org/spec/UML/> Acesso em agosto de 2016.

[41] PALLADIO - **Palladio Component Model**: <https://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model> Acesso em agosto de 2016.

[42] QIMPRESS - **The Q-ImPrESS Project**. Disponível em: <http://www.q-impress.eu/wordpress/> Acesso em agosto de 2016.

[43] QVTO-FLATTEN-STM - **The QVTo-flatten-stm transformation**. Disponível em: <https://code.google.com/archive/p/qvto-flatten-stm/> Acesso em agosto de 2016.

[44] QVTR2 - **QVT-Rational Project**. Disponível em: <https://code.google.com/archive/p/qvtr2/> Acesso em agosto de 2016.

[45] R-Project - **R**. Disponível em <https://www.r-project.org/> Acesso em agosto de 2016.

[46] ROSSUM, G. van; DRAKE Jr, F. L. **The Python Language Reference Manual**. Network Theory Ltd., 2011.

[47] SHAH, S. M. A.; ANASTASAKIS, K.; BORDBAR, B. **From UML to Allow and Back Again**. In Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, 2009.

[48] SHAPIRO, S. S.; WILK, M. B. **An analysis of variance test for normality (complete samples)**. Biometrika 52 (3-4), pp. 591-611, 1964.

[49] SIMMONDS, D.; SOLBERG, A.; REDDY, R.; FRANCE, R.; GHOSH, S. **An Aspect Oriented Model Driven Framework**. In: Proc. of the 9th IEEE International EDOC Enterprise Computing Conference, 2004.

[50] SIQUEIRA, F. L. **"Hello world" in Eclipse QVTo**: <http://www.levysiqueira.com.br/2011/01/eclipse-qvto-hello-world/> Acesso em agosto de 2016.

[51] SOFA2 - **Sofa 2.0 Project**. Disponível em: <http://websvn.ow2.org/listing.php?repname=sofa&> Acesso em agosto de 2016.

[52] STAHL, T.; VOELTER, M. **Model-Driven Software Development: Technology, Engineering, Management**. Wiley, 1st ed. 2008.

[53] TISI, M.; JOUAULT, F.; FRATERNALI, P.; CERI, S; Bézivin, J. **On the Use of Higher-Order Model Transformations**. In Model Driven Architecture - Foundations and Applications. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp.18–33.

[54] TISI, M.; MARTINEZ, S.; CHOURA, H. **Parallel Execution of ATL Transformation Rules**. MoDELS, Miami, United States, pp. 656-672, 2013.

[55] VAN AMSTEL, M.; BOSEMS, S.; KURTEV, I.; PIRES, L. F. **Performance in Model Transformations: Experiments with ATL and QVT**. Volume 6707 of the LNCS series, pp 198-212, 2011.

[56] VAN DEN BERG, K.; TEKINERDOGAN, B.; NGUYEN, H. **Analysis of Cross-cutting in Model Transformations**. In: Proceedings of the ECMDA-TW, Traceability Workshop, pp 51–64, 2006.

[57] WILCOXON, F. **Individual comparisons by ranking methods**. Biometrics Bulletin 1 (6), pp. 80-83, 1944.

Apêndice A

Gramática Xtext de AQVT

```
grammar org.xtext.qvt.Aqvt with org.eclipse.xtext.common.Terminals

generate aqvt "http://www.xtext.org/qvt/Aqvt"

AqvtGrammar:
	path=Path elements+=AspectElement*
;

Path:
	'use' path=STRING
;

AspectElement:
	'Aspect' name=ID '{'
	  elements+=AbstractElement*
	  '}'
;

AbstractElement:
	pe=PointcutElement  ae=AdviceElement
;

PointcutElement:
	'pointcut' name=ID param=PointcutParam jes=Expression ';'
;

PointcutParam:
	{PointcutParam} '():' | '(' pCheck=ParamCheck '):'
;

ParamCheck:
	name=ID
;

Expression:
	Or
;

Or returns Expression:
	And ({Or.left=current} '||' right=And)*
;

And returns Expression:
	JoinpointElement ({And.left=current} '&&' right=JoinpointElement)*
;
```

```
JoinpointElement returns Expression:
	jp=Joinpoint | '(' Expression ')'
;

Joinpoint:
	jpKind=JOINPOINT_KIND '(' jpParams=JOINPOINT_PARAMS ')'
;

JOINPOINT_PARAMS:
	jpParamsEnd=JOINPOINT_PARAMS_END name=ID | {JOINPOINT_PARAMS_END}
jpParamsEnd=JOINPOINT_PARAMS_END '*'
;

AdviceElement:
	kind=ADVICE_KIND '():' name=[PointcutElement] jpParam=JoinpointOnAdviceParam
'{' imports+= QvtImport* qvtOps+=ModuleOperationCS* '}'
;

AdviceParam:
	{AdviceParam} '():' | '(' param=ParamCheck '):'
;

JoinpointOnAdviceParam:
	{JoinpointOnAdviceParam} '()' | '(' name=ID ')'
;



terminal JOINPOINT_PARAMS_END:
	'Rule' | 'Helper' | 'Object' | 'Binding' | 'Filter' | 'Element' |
'Constructor' | 'Field'
;

terminal ADVICE_KIND:
	'before' | 'after' | 'around'
;

terminal JOINPOINT_KIND:
	'RuleExecution' | 'RuleCall' | 'HelperExecution' | 'HelperCall' | 'Within' |
'BindingGeneration' |
                                'ElementGeneration' | 'FilterExecution' |
'ObjectInit' | 'ObjectFieldGet' | 'ObjectFieldSet' |
                                'ModelFieldSelf' | 'ModelFieldResult' |
'ConstructorExecution'
;

//QVT-O Eclipse:

QvtImport:
	'modeltype' model=ID 'uses' modelPath=STRING ';'
;

ModuleOperationCS:
```

```
	map=MappingOperationCS | helper=HelperOperation
;

MappingOperationCS:
	declaration=MappingDeclarationCS | definition=MappingDefinitionCS
;

MappingDeclarationCS returns MappingOperationCS:
	header=MappingOperationHeaderCS ';'
;

MappingDefinitionCS returns MappingOperationCS:
	headerCs=MappingOperationHeaderCS '{'
		content+=MappingExpression*
	'}'
;

MappingOperationHeaderCS returns MappingOperationCS:
	"mapping" (inName=ID '::' inElement=ID)? name=ID '():' outName=ID '::'
outElement=ID

;

MappingExpression:
	outAttribute=ID ':=' in=InExpression ';'
;

InExpression:
	InExpressionSelf | InExpressionStatic | HelperCall
;

InExpressionSelf returns InExpression:
	name='self' ('.' ID)* | name='self' ('.' ID)* '+' next=InExpression
;

InExpressionStatic returns InExpression:
	StringExpression | NumberExpression
;

StringExpression:
	{StringExpression} STRING | STRING '+' next=InExpression
;

NumberExpression:
	{NumberExpression} INT | INT '+' next=InExpression
;

HelperCall:
	helper=[HelperOperation] '(' (ownedParams+=(InExpressionSelf |
InExpressionStatic) (',' ownedParams+=(InExpressionSelf | InExpressionStatic))*)? ')'
(('+' otherExpr+=InExpression)*)?
;

Parameter:
	name=ID ':' type=ID
```

```
;

HelperOperation:
	declaration=HelperDeclaration | definition=HelperDefinition
;


HelperDeclaration returns HelperOperation:
	header=HelperOperationHeader ';'
;

HelperDefinition returns HelperOperation:
	headerCS=HelperOperationHeader '{'
		content+=HelperExpressionHeader*
	'}'
;

HelperOperationHeader returns HelperOperation:
	"helper" (inName=ID '::' inElement=ID)? name=ID '(' (ownedParams+=Parameter
(',' ownedParams+=Parameter)*)? '):' returnType=HelperReturnType
;

HelperReturnType:
	name=ID | model=ID '::' type=ID
;

HelperExpressionHeader:
	name='return' expression=HelperExpression
;

HelperExpression:
	if=IfExpression | in=InExpression
;

IfExpression:
	'if' if=Boolean 'then' then=BooleanResult (=>'else' else=BooleanResult)?
;

Boolean:
	ifexp=IfExpression | inexpLeft=InExpression '=' inexpRight=InExpression
;

BooleanResult:
	ifexp=IfExpression | inexp=InExpression
;

enum InitOp : EQUALS='=' | COLON_EQUALS=':=' | COLON_COLON_EQUALS='::=';

enum DirectionKindCS : in='in'| out='out' | inout='inout';
```

Apêndice B

Transformação SAMM2PCM_allocation com Aspectos escritos em QVT (sem AQVT)

```
import SAMM2PCM_system;
import SAMM2PCM_resourceenvironment;

modeltype SAMM_SS uses 'http://q-impress.eu/samm/staticstructure';
modeltype SAMM_CORE uses 'http://q-impress.eu/samm/core';
modeltype PCM_CORE uses 'http://sdq.ipd.uka.de/PalladioComponentModel/Core/4.0';
modeltype PCM_REP uses 'http://sdq.ipd.uka.de/PalladioComponentModel/Repository/4.0';
modeltype PCM_SYS uses 'http://sdq.ipd.uka.de/PalladioComponentModel/System/4.0';
modeltype SAMM_AL uses 'http://q-impress.eu/samm/deployment/allocation';
modeltype SAMM_HW uses 'http://q-impress.eu/samm/deployment/hardware';
modeltype SAMM_TE uses 'http://q-impress.eu/samm/deployment/targetenvironment';
modeltype SAMM_SEFF uses 'http://q-impress.eu/seff';
modeltype SAMM_QOS uses 'http://q-
impress.eu/qualityannotationdecorator/seffdecorator';
modeltype PCM_RE uses
'http://sdq.ipd.uka.de/PalladioComponentModel/ResourceEnvironment/4.0';
modeltype PCM_RT uses
'http://sdq.ipd.uka.de/PalladioComponentModel/ResourceType/4.0';
modeltype PCM_AL uses 'http://sdq.ipd.uka.de/PalladioComponentModel/Allocation/4.0';
modeltype Trace uses "http://ufcg.edu.br/aqvt/TraceMM"

transformation SAMM2PCM(in rep: SAMM_SS, in sys: SAMM_SS, in qos : SAMM_QOS, in
pcm_res :PCM_REP, in samm_hw : SAMM_HW, in samm_te : SAMM_TE, in pcm_rt : PCM_RT, out
pcm_rep: PCM_REP, out pcm_sys: PCM_SYS, out pcm_re: PCM_RE, out pcm_alloc: PCM_AL)
        extends transformation SAMM2PCM_system(in rep: SAMM_SS, in sys: SAMM_SS, in
seff : SAMM_SEFF, in qos : SAMM_QOS, in pcm_res :PCM_REP, in samm_hw : SAMM_HW, in
samm_te : SAMM_TE, in pcm_rt : PCM_RT, out pcm_rep: PCM_REP, out pcm_sys: PCM_SYS,
out pcm_re: PCM_RE);

configuration property allocationModelName : String;

main() {
        sys.rootObjects()[SAMM_SS::ServiceArchitectureModel]->map toAllocation();
        sys.rootObjects()[SAMM_SS::ServiceArchitectureModel]->map toTrace("main");
}

mapping SAMM_SS::ServiceArchitectureModel::toAllocation() : PCM_AL::Allocation @
pcm_alloc {
        entityName := allocationModelName;
        system_Allocation := sys.rootObjects()[SAMM_SS::ServiceArchitectureModel]-
>any(true).late resolveone(PCM_SYS::System);
```

```
        sys.rootObjects()[SAMM_SS::ServiceArchitectureModel]- >map
toTrace("toAllocation");
        targetResourceEnvironment_Allocation :=
samm_te.rootObjects()[SAMM_TE::TargetEnvironment]->any(true).late
resolveone(PCM_RE::ResourceEnvironment);
        samm_te.rootObjects()[SAMM_TE::TargetEnvironment]-> map
toTrace("toAllocation");
        allocationContexts_Allocation := self.service->map toAllocationContext()-
>asSet();
}
mapping SAMM_AL::Service::toAllocationContext() : PCM_AL::AllocationContext {
        entityName := self.subcomponentInstance.name;
        resourceContainer_AllocationContext := self.container.late
resolveone(PCM_RE::ResourceContainer);
        assemblyContext_AllocationContext := self.subcomponentInstance.late
resolveone(PCM_CORE::composition::AssemblyContext);

}

mapping SAMM_SS::ServiceArchitectureModel::toTrace(name : String) :
TraceFilter::Trace {
            rulename := name;
            entity := self;
}
```

Anexo A

Transformação SimpleUML2RDB

```
modeltype UML uses 'http://www.eclipse.org/qvt/1.0.0/Operational/examples/simpleuml';

modeltype RDB uses 'http://www.eclipse.org/qvt/1.0.0/Operational/examples/rdb';


transformation Simpleuml_To_Rdb(in uml : UML, out RDB);


main() {
	uml.rootObjects()[UML::Model]->map model2RDBModel();
}

mapping UML::Model::model2RDBModel() : RDB::Model {
	name := self.name;
	schemas := self.map package2schemas()->asOrderedSet();
}


mapping UML::Package::package2schemas() : Sequence(RDB::Schema) {
	init {
		result := self.map package2schema()->asSequence()->
			union(self.getSubpackages()->map package2schemas()->flatten());
	}
}




mapping UML::Package::package2schema() : RDB::Schema
	when { self.hasPersistentClasses() }
{
	name := self.name;
	elements := self.ownedElements[UML::Class]->map persistentClass2table()-
>asOrderedSet()
}


mapping UML::Class::persistentClass2table() : RDB::Table
	when { self.isPersistent() }
{
	name := self.name;
	columns := self.map class2columns(self)->sortedBy(name);
	primaryKey := self.map class2primaryKey();
	foreignKeys := self.attributes.resolveIn(
```

```
                                UML::Property::relationshipAttribute2foreignKey,
                                RDB::constraints::ForeignKey)->asOrderedSet();
}


mapping UML::Class::class2primaryKey() : RDB::constraints::PrimaryKey {
	name := 'PK' + self.name;
	includedColumns := self.resolveoneIn(UML::Class::persistentClass2table,
RDB::Table).getPrimaryKeyColumns()
}


mapping UML::Class::class2columns(targetClass: UML::Class) :
OrderedSet(RDB::TableColumn) {
	init {
		result := self.map dataType2columns(targetClass)->
			union(self.map generalizations2columns(targetClass))-
>asOrderedSet()
	}
}


mapping UML::DataType::dataType2columns(in targetType : UML::DataType) :
OrderedSet(RDB::TableColumn) {
	init {
		result := self.map primitiveAttributes2columns(targetType)->
			union(self.map enumerationAttributes2columns(targetType))->
			union(self.map relationshipAttributes2columns(targetType))->
			union(self.map associationAttributes2columns(targetType))-
>asOrderedSet()
	}
}




mapping UML::DataType::dataType2primaryKeyColumns(in prefix : String, in
leaveIsPrimaryKey : Boolean, in targetType : UML::DataType) :
OrderedSet(RDB::TableColumn) {
	init {
		result := self.map dataType2columns(self)->select(isPrimaryKey)->
			collect(c | object RDB::TableColumn {
				name := prefix + '_' + c.name;
				domain := c.domain;
				type := object RDB::datatypes::PrimitiveDataType {
					name := c.type.name;
				};
				isPrimaryKey := leaveIsPrimaryKey
			})->asOrderedSet();
	}
}
```

```
mapping UML::DataType::primitiveAttributes2columns(in targetType: UML::DataType) :
OrderedSet(RDB::TableColumn) {
	init {
		result := self.attributes->map primitiveAttribute2column(targetType)-
>asOrderedSet()
	}
}


mapping UML::Property::primitiveAttribute2column(in targetType: UML::DataType) :
RDB::TableColumn
	when { self.isPrimitive() }
{
	isPrimaryKey := self.isPrimaryKey();
	name := self.name;
	type := object RDB::datatypes::PrimitiveDataType { name :=
umlPrimitive2rdbPrimitive(self.type.name); };
}


mapping UML::DataType::enumerationAttributes2columns(in targetType: UML::DataType) :
OrderedSet(RDB::TableColumn) {
	init {
		result := self.attributes->map enumerationAttribute2column(targetType)-
>asOrderedSet()
	}
}


mapping UML::Property::enumerationAttribute2column(in targetType: UML::DataType) :
RDB::TableColumn
	when { self.isEnumeration() }
{
	isPrimaryKey := self.isPrimaryKey();
	name := self.name;
	type := object RDB::datatypes::PrimitiveDataType { name := 'int'; };
}


mapping UML::DataType::relationshipAttributes2columns(in targetType: UML::DataType) :
OrderedSet(RDB::TableColumn) {
	init {
		result := self.attributes->map
relationshipAttribute2foreignKey(targetType)->
			collect(includedColumns)->asOrderedSet();
	}
}


mapping UML::Property::relationshipAttribute2foreignKey(in targetType:
UML::DataType) : RDB::constraints::ForeignKey
	when { self.isRelationship() }
{
	name := 'FK' + self.name;
```

```
	includedColumns := self.type.asDataType().map
dataType2primaryKeyColumns(self.name, self.isIdentifying(), targetType);
	referredUC := self.type.late resolveoneIn(UML::Class::class2primaryKey,
RDB::constraints::PrimaryKey);
}


mapping UML::DataType::associationAttributes2columns(targetType : UML::DataType) :
OrderedSet(RDB::TableColumn) {
	init {
		result := self.attributes[isAssociation()]->
			collect(type.asDataType()->map dataType2columns(targetType))-
>asOrderedSet()
	}
}


mapping UML::Class::generalizations2columns(targetClass : UML::Class) :
OrderedSet(RDB::TableColumn) {
	init {
		result := self.generalizations.general->map class2columns(targetClass)-
>flatten()->asOrderedSet();
	}
}


query UML::Package::getSubpackages() : OrderedSet(UML::Package) {
	return self.ownedElements[UML::Package]->asOrderedSet()
}


query UML::Type::asDataType() : UML::DataType {
	return self.oclAsType(UML::DataType)
}


query UML::Property::isPrimaryKey() : Boolean {
	return self.stereotype->includes('primaryKey')
}


query UML::Property::isIdentifying() : Boolean {
	return self.stereotype->includes('identifying')
}


query UML::Property::isPrimitive() : Boolean {
	return self.type.oclIsKindOf(UML::PrimitiveType)
}


query UML::Property::isEnumeration() : Boolean {
	return self.type.oclIsKindOf(UML::Enumeration)
}
```

```
query UML::Property::isRelationship() : Boolean {
	return self.type.oclIsKindOf(UML::DataType) and self.type.isPersistent()
}


query UML::Property::isAssociation() : Boolean {
	return self.type.oclIsKindOf(UML::DataType) and not self.type.isPersistent()
}


query RDB::Table::getPrimaryKeyColumns() : OrderedSet(RDB::TableColumn) {
	return self.columns->select(isPrimaryKey)
}


query UML::ModelElement::isPersistent() : Boolean {
	return self.stereotype->includes('persistent')
}


query UML::Package::hasPersistentClasses() : Boolean {
	return self.ownedElements->exists(
		let c : UML::Class = oclAsType(UML::Class) in
			c.oclIsUndefined() implies c.isPersistent())
}


helper umlPrimitive2rdbPrimitive(in name : String) : String {
	return if name = 'String' then 'varchar' else
		if name = 'Boolean' then 'int' else
			if name = 'Integer' then 'int' else
				name
			endif
		endif
	endif
}
```

Anexo B

Transformação SAMM2PCM_allocation

```
import SAMM2PCM_system;
import SAMM2PCM_resourceenvironment;

modeltype SAMM_SS uses 'http://q-impress.eu/samm/staticstructure';
modeltype SAMM_CORE uses 'http://q-impress.eu/samm/core';
modeltype PCM_CORE uses 'http://sdq.ipd.uka.de/PalladioComponentModel/Core/4.0';
modeltype PCM_REP uses 'http://sdq.ipd.uka.de/PalladioComponentModel/Repository/4.0';
modeltype PCM_SYS uses 'http://sdq.ipd.uka.de/PalladioComponentModel/System/4.0';
modeltype SAMM_AL uses 'http://q-impress.eu/samm/deployment/allocation';
modeltype SAMM_HW uses 'http://q-impress.eu/samm/deployment/hardware';
modeltype SAMM_TE uses 'http://q-impress.eu/samm/deployment/targetenvironment';
modeltype SAMM_SEFF uses 'http://q-impress.eu/seff';
modeltype SAMM_QOS uses 'http://q-
impress.eu/qualityannotationdecorator/seffdecorator';
modeltype PCM_RE uses
'http://sdq.ipd.uka.de/PalladioComponentModel/ResourceEnvironment/4.0';
modeltype PCM_RT uses
'http://sdq.ipd.uka.de/PalladioComponentModel/ResourceType/4.0';
modeltype PCM_AL uses 'http://sdq.ipd.uka.de/PalladioComponentModel/Allocation/4.0';

transformation SAMM2PCM(in rep: SAMM_SS, in sys: SAMM_SS, in qos : SAMM_QOS, in
pcm_res :PCM_REP, in samm_hw : SAMM_HW, in samm_te : SAMM_TE, in pcm_rt : PCM_RT, out
pcm_rep: PCM_REP, out pcm_sys: PCM_SYS, out pcm_re: PCM_RE, out pcm_alloc: PCM_AL)
        extends transformation SAMM2PCM_system(in rep: SAMM_SS, in sys: SAMM_SS, in
seff : SAMM_SEFF, in qos : SAMM_QOS, in pcm_res :PCM_REP, in samm_hw : SAMM_HW, in
samm_te : SAMM_TE, in pcm_rt : PCM_RT, out pcm_rep: PCM_REP, out pcm_sys: PCM_SYS,
out pcm_re: PCM_RE);

configuration property allocationModelName : String;

main() {
        sys.rootObjects()[SAMM_SS::ServiceArchitectureModel]->map toAllocation();
}

mapping SAMM_SS::ServiceArchitectureModel::toAllocation() : PCM_AL::Allocation @
pcm_alloc {
        entityName := allocationModelName;
        system_Allocation := sys.rootObjects()[SAMM_SS::ServiceArchitectureModel]-
>any(true).late resolveone(PCM_SYS::System);
        targetResourceEnvironment_Allocation :=
samm_te.rootObjects()[SAMM_TE::TargetEnvironment]->any(true).late
resolveone(PCM_RE::ResourceEnvironment);
        allocationContexts_Allocation := self.service->map toAllocationContext()-
>asSet();
```

```
}
mapping SAMM_AL::Service::toAllocationContext() : PCM_AL::AllocationContext {
        entityName := self.subcomponentInstance.name;
        resourceContainer_AllocationContext := self.container.late
resolveone(PCM_RE::ResourceContainer);
        assemblyContext_AllocationContext := self.subcomponentInstance.late
resolveone(PCM_CORE::composition::AssemblyContext);

}
```

Anexo C

Transformação Uml2Rdbms_Replication

```
modeltype simpleUml "strict" uses umlMM('umlMM');
modeltype simpleDb "strict" uses rdbmsMM('rdbmsMM');
modeltype ecore "strict" uses ecore('http://www.eclipse.org/emf/2002/Ecore');
modeltype qvt "strict" uses
qvtoperational('http://www.eclipse.org/QVT/1.0.0/Operational');

/**
@nfpmodel[nfps => "Uml2Rdbms.nfps"]
*/

/**
@keydefs{
	simpleUml::Package(#name),
	simpleUml::Classifier(#name, #namespace),
	simpleUml::Class(#name, #namespace),
	simpleUml::Attribute(#name, #owner),
	simpleUml::Association(#name, #owner),
	simpleUml::Generalization(#name, #owner),

	simpleDb::Schema(#name),
	simpleDb::Partition(#name, #schema),
	simpleDb::Table(#name, #schema),
	simpleDb::Column(#name, #owner),
	simpleDb::Key(#name, #owner),
	simpleDb::ForeignKey(#name, #owner)
}
*/

transformation Uml2Rdbms_Replication(in classModel:simpleUml, out dbModel:simpleDb);

main() {
	var packages := classModel.rootObjects()[Package];
	assert fatal (packages->size() = 1) with log ("Input model does not contain
exactly one package.");

	packages->map Package2Schema();
}

mapping Package::Package2Schema() : Schema {
	log("Mapping package " + self.name + " to schema");

	result.name := self.name;
```

```
	-- Map top classes
	result.table += self.classifier[Class]->map TopClassToTable();

	-- Map sub classes
	self.classifier[Class]->map SubclassToTable(result);

	-- Map associations
	self.association->map AssociationToRelationship(result);

	-- Decide for Paritioning
	self.classifier[Class]
		// Select only top classes
		->select(c | not self.generalization->exists(g | g.subType = c))
		->map UsePartitioning_VariationPoint();
}

-- Marker mapping
mapping Class::ClassToTableMarker(inout table:Table) : Table {
	init {
		result := table;
	}
	log("Established marker mapping ClassToTable " + self.name + " ---> " +
result.name);
}

mapping Class::ClassToPartitionMarker(inout partition:Partition) : Partition {
	init {
		result := partition;
	}
	log("Established marker mapping ClassToPartition " + self.name + " ---> " +
result.name);
}

-- Base mapping for all ClassToTable mappings
mapping Class::ClassToTableActual(inout table:Table) : Table {
	init {
		result := table;
	}
	result.name := self.name;

	-- Map all attributes
	result.column += self.attribute->map Attribute2Column();

	-- Map id attributes on autogen cols and primary key
	table.hasKey :=  object Key {
		name := self.name + "_pkey";
		self.attribute->select(a | a.isId = true)->forEach(attr) {
			var mappingCol :=
attr.resolveoneIn(Attribute::Attribute2Column).oclAsType(Column);
			column += mappingCol;
		};
	};
}

mapping Class::TopClassToTable() : Table
```

```
when {
	not self.namespace.generalization->exists(g | g.subType = self);
} {
	log("Mapping top class " + self.name + " to table");

	-- Some basic checks
	assert fatal (self.attribute->exists(a | a.isId)) with log('Class ' +
self.name + ' has no id attributes');

	self.map ClassToTableActual(result);
	self.map ClassToTableMarker(result);
}

mapping Class::SubclassToTable(inout schema:Schema)
when {
	self.namespace.generalization->exists(g | g.subType = self);
} {
	log("Mapping subclass " + self.name + " to table");

	-- Check that class participates to only one generalization as subtype
	var generalizations := self.namespace.generalization->select(g | g.subType =
self);
	assert fatal (generalizations->size() = 1) with log ("Class " + self.name + "
has more than one super class");

	-- Get the table on which the superclass has been mapped
	var superClass := generalizations->any(true).superType;
	var superTable := superClass.resolveoneIn(Class::ClassToTableMarker);
	if (superTable = null) then {
		-- Super class has not been mapped, map it before
		superTable := superClass.map SubclassToTable(schema);
	} endif;

	schema.table += self.map InheritanceMapping_VariationPoint();
}

/**
@varpoint {
	name := InheritanceStrategy,
	description := "Selects the strategy to map a generalization.",
	analyzer := it.polimi.qvtr2.examples.orm.qnanalyzer.QnAnalyzer(),
	impact := {
		nfps::GlobalResponseTime(_),
		nfps::ResponseTime(_, $"self")
	}
}
*/
mapping Class::InheritanceMapping_VariationPoint() : Table
disjuncts Class::AssociationCollapse_Variant, Class::UpwardCollapse_Variant;

/**
@variant {
	name := UpwardCollapse,
	description := "Maps a subclass on the parent table.",
	excludes := AssociationCollapse($"self.namespace.generalization
```

```
            ->select(g | g.superType = self.namespace.generalization
                  ->select(g1 | g1.subType = self)->any(true).superType)
            .subType
            ")
}
*/
mapping Class::UpwardCollapse_Variant() : Table {
      init {
            var generalizations := self.namespace.generalization->select(g |
g.subType = self);
            assert fatal (generalizations->size() = 1) with log ("Class " +
self.name + " has more than one super class");

            -- Get the table on which the superclass has been mapped
            var superClass := generalizations->any(true).superType;
            var superTable := superClass.resolveoneIn(Class::ClassToTableMarker);
            result := superTable.oclAsType(Table);
      }

      log("Mapping (UpwardCollapse) class " + self.name + " to table");

      -- Invoke the marker mapping
      self.map ClassToTableMarker(result);

      -- Map attributes
      result.column += self.attribute->map Attribute2Column();

      -- Create the discriminator column, if not yet created
      if (not result.column->exists(c | c.name = 'discriminator')) then {
            result.column += object Column {
                  name := 'discriminator';
                  type := 'CHAR';
            };
      } endif;
}

/**
@variant {
      name := AssociationCollapse,
      description := "Maps a subclass on its own table and on an association with
the table onto which the parent was mapped.",
      excludes := UpwardCollapse($"self.namespace.generalization
            ->select(g | g.superType = self.namespace.generalization
                  ->select(g1 | g1.subType = self)->any(true).superType)
            .subType
            ")
}
*/
mapping Class::AssociationCollapse_Variant() : Table
{
      log("Mapping (AssociationCollapse) class " + self.name + " to table");

      var generalizations := self.namespace.generalization->select(g | g.subType =
self);
```

```
	assert fatal (generalizations->size() = 1) with log ("Class " + self.name + "
has more than one super class");

	-- Get the table on which the superclass has been mapped
	var superClass := generalizations->any(true).superType;
	var superTable :=
superClass.resolveoneIn(Class::ClassToTableMarker).oclAsType(Table);

	result.name := self.name;

	-- Invoke the marker mapping
	self.map ClassToTableMarker(result);

	-- Create a primary key
	result.hasKey := object Key {
		name := self.name + '_pk';
	};

	-- Clone SuperClass ids for this class
	superTable.hasKey.column->forEach(cl) {
		result.hasKey.column += object Column {
			name := cl.name;
			type := cl.type;
			owner := result;
		};
	};

	-- Create a foreign key referring to the parent class pkey on this table pkey
cols
	result.hasFKey += object ForeignKey {
		name := self.name + '_' + superClass.name + '_Gen';
		column := result.hasKey.column;
		refersTo := superTable.hasKey;
	}
}

mapping Attribute::Attribute2Column() : Column
{
	result.name := self.name;
	result.type := UmlTypeToDbtype(self.type.name);

	-- This should be implemented as a merge
	if (self.isId = true) then {
		self.map IdAttribute2AutogenColumn_VariationPoint();
	} endif;
}

/**
@varpoint {
	name := AutogenerationStrategy,
	description := "Selects the strategy to auto generate values for id
attributes",
	analyzer := it.polimi.qvtr2.examples.orm.qnanalyzer.QnAnalyzer(),
	impact := {
		nfps::GlobalResponseTime(_),
```

```
            nfps::ResponseTime(_, $"self.owner")
        }
}
*/
mapping Attribute::IdAttribute2AutogenColumn_VariationPoint() : Column
disjuncts Attribute::IdAttribute2AutogenSequenceColumn_Variant,
Attribute::IdAttribute2AutogenRandomColumn_Variant;

/**
@variant {
        name := SequentialGeneration,
        description := "A sequential generation strategy."
}
*/
mapping Attribute::IdAttribute2AutogenSequenceColumn_Variant() : Column
{
        init {
                result :=
self.resolveoneIn(Attribute::Attribute2Column).oclAsType(Column);
        }
        assert fatal (self.isId = true) with log('Trying to map non id attribute ' +
self.name + ' to an autogen column');
        result.autoGenerate := 'SEQUENTIAL';
}

/**
@variant {
        name := RandomizedGeneration,
        description := "A randomized generation strategy.",
        excludes := ClassToPartition($"self.owner")
}
*/
mapping Attribute::IdAttribute2AutogenRandomColumn_Variant() : Column
{
        init {
                result :=
self.resolveoneIn(Attribute::Attribute2Column).oclAsType(Column);
        }
        assert fatal (self.isId = true) with log('Trying to map non id attribute ' +
self.name + ' to an autogen column');
        result.autoGenerate := 'RANDOM';
}

mapping Association::AssociationToRelationship(inout schema:Schema)
disjuncts Association::OneToNAssociation, Association::NToOneAssociation,
Association::NToNAssociation;

mapping Association::OneToNAssociation(inout schema:Schema)
when {
        self.sourceCardinality = 1 and self.destCardinality = -1;
}
{
        log("Mapping association " + self.name);
```

```
	var sourceTable :=
self.source.resolveoneIn(Class::ClassToTableMarker).oclAsType(Table);
	var destTable :=
self.destination.resolveoneIn(Class::ClassToTableMarker).oclAsType(Table);

	assert fatal (sourceTable <> null) with log('Trying to map association ' +
self.name + ' but source class has not been mapped');
	assert fatal (destTable <> null) with log('Trying to map association ' +
self.name + ' but dest class has not been mapped');

	destTable.hasFKey += object ForeignKey {
		name := self.name + '_fk';
		refersTo := sourceTable.hasKey;
		sourceTable.hasKey.column->forEach(cl) {
			column += object Column {
				name := cl.name + '_' + self.name + '_fk';
				type := cl.type;
				owner := destTable;
			}
		};
	};
}

mapping Association::NToOneAssociation(inout schema:Schema)
when {
	self.sourceCardinality = -1 and self.destCardinality = 1;
}
{
	log("Mapping association " + self.name);

	var sourceTable :=
self.source.resolveoneIn(Class::ClassToTableMarker).oclAsType(Table);
	var destTable :=
self.destination.resolveoneIn(Class::ClassToTableMarker).oclAsType(Table);

	assert fatal (sourceTable <> null) with log('Trying to map association ' +
self.name + ' but source class has not been mapped');
	assert fatal (destTable <> null) with log('Trying to map association ' +
self.name + ' but dest class has not been mapped');

	sourceTable.hasFKey += object ForeignKey {
		name := self.name + '_fk';
		refersTo := destTable.hasKey;
		destTable.hasKey.column->forEach(cl) {
			column += object Column {
				name := cl.name + '_' + self.name + '_fk';
				type := cl.type;
				owner := sourceTable;
			}
		};
	};
}

mapping Association::NToNAssociation(inout schema:Schema)
when {
```

```
	self.sourceCardinality = -11 and self.destCardinality = -1;
}
{
	log("Mapping association " + self.name);

	var sourceTable :=
self.source.resolveoneIn(Class::ClassToTableMarker).oclAsType(Table);
	var destTable :=
self.destination.resolveoneIn(Class::ClassToTableMarker).oclAsType(Table);

	assert fatal (sourceTable <> null) with log('Trying to map association ' +
self.name + ' but source class has not been mapped');
	assert fatal (destTable <> null) with log('Trying to map association ' +
self.name + ' but dest class has not been mapped');

	var at:Table;
	schema.table += object at:Table {
		name := self.name;
		hasKey := object Key {
			name := self.name + '_pk';
		};

		-- Create a foreign key in the assoc table for the source table
		var sourceTableFKey := object ForeignKey {
			name := sourceTable.name + '_fk';
			refersTo := sourceTable.hasKey;
		};
		at.hasFKey += sourceTableFKey;

		-- Create a column in the assoc table for each source table id col
		sourceTable.hasKey.column->forEach(cl) {
			var newCol := object Column {
				name := sourceTable.name + '_' + cl.name;
				type := cl.type;
			};
			at.column += newCol;
			at.hasKey.column += newCol;
			sourceTableFKey.column += newCol;
		};

		-- Create a foreign key in the assoc table for the dest table
		var destTableFKey := object ForeignKey {
			name := destTable.name + '_fk';
			refersTo := destTable.hasKey;
		};
		at.hasFKey += destTableFKey;

		-- Create a column in the assoc table for each dest table id col
		destTable.hasKey.column->forEach(cl) {
			var newCol := object Column {
				name := destTable.name + '_' + cl.name;
				type := cl.type;
			};
			at.column += newCol;
			at.hasKey.column += newCol;
```

```
                destTableFKey.column += newCol;
            };
        };
}

query UmlTypeToDbtype(umlType:String) : String {
        if (umlType = 'Integer') then {
            return 'NUMBER';
        } endif;

        if (umlType = 'String') then {
            return 'VARCHAR';
        } endif;

        if (umlType = 'Boolean') then {
            return 'BOOLEAN';
        } endif;

        if (umlType = 'Date') then {
            return 'DATETIME';
        } endif;

        assert fatal (false) with log('Unable to translate UML type ' + umlType + ' to
DB type');
        return null;
}

/**
@varpoint {
        name := UsePartitioning,
        description := "Selects whether a class should be mapped onto a single table
or a partition of tables",
        analyzer := it.polimi.qvtr2.examples.orm.qnanalyzer.QnAnalyzer(),
        impact := {
            nfps::GlobalResponseTime(_),
            nfps::ResponseTime(_, $"self")
        }
}
*/
mapping Class::UsePartitioning_VariationPoint()
disjuncts Class::ClassToSingleTable_Variant, Class::ClassToPartition_Variant;

/**
@variant {
        name := ClassToSingleTable,
        description := "Maps a class onto a single table."
}
*/
mapping Class::ClassToSingleTable_Variant() {
        log("Mapping class " + self.name + " to single table");

        var mappingTable :=
self.resolveoneIn(Class::ClassToTableMarker).oclAsType(Table);
        mappingTable.partitionId := 0;
}
```

```
/**
@variant {
	name := ClassToPartition,
	description := "Maps a class onto a partition.",
	excludes := RandomizedGeneration($"self.attribute")
}
*/
mapping Class::ClassToPartition_Variant() {
	log("Mapping class " + self.name + " to single partition");

	-- Retrieve all subtables
	var subClasses := self.getAllSubClasses();
	var subTables := subClasses->resolveIn(Class::ClassToTableMarker)-
>oclAsType(Table);

	-- Create a partition
	var mappingTable :=
self.resolveoneIn(Class::ClassToTableMarker).oclAsType(Table);
	var schema := mappingTable.schema;

	-- Create a new partition
	var partition := object Partition {
		name := self.name;
		mappingTable.partitionId := 0;

		-- Attach table to the partition
		table += mappingTable;

		-- Attach all the subtables if they exist
		table += subTables;
	};
	schema.partition += partition;

	-- Invoke marker relationship
	self.map ClassToPartitionMarker(partition);

	-- At least two slices in the partition
	partition.size := 2;

	-- Check if more slices should be added
	self.map AddAnotherSlice_VariationPoint(partition.size + 1);
}

/**
@varpoint {
	name := SlicesNumber,
	description := "Selects the number of slices to add in a partition",
	subjects := {#self},
	analyzer := it.polimi.qvtr2.examples.orm.qnanalyzer.QnAnalyzer(),
	impact := {
		nfps::GlobalResponseTime(_),
		nfps::ResponseTime(_, $"self")
	}
}
```

```
*/
mapping Class::AddAnotherSlice_VariationPoint(in sliceId:Integer)
disjuncts Class::AnotherSlice_Variant;

/**
@variant {
	name := AnotherSlice,
	description := "Adds another slice to the partition."
}
*/
mapping Class::AnotherSlice_Variant(in sliceId:Integer) {
	log("Adding slice " + sliceId.toString() + " for class " + self.name);

	var mappingPartition :=
self.resolveoneIn(Class::ClassToPartitionMarker).oclAsType(Partition);
	mappingPartition.size := sliceId;

	self.map AddAnotherSlice_VariationPoint(sliceId + 1);
}

query Class::getAllSubClasses() : Set(Class) {
	return self.namespace.generalization->select(g | g.superType = self)
		->collect(g | g.subType)->asSet();
}
```