UNIVERSIDADE FEDERAL FLUMINENSE

ROBERTO WEIDMANN MENEZES

APLICAÇÕES DE CONTRATOS DE TRANSFORMAÇÃO

NITERÓI

2011

UNIVERSIDADE FEDERAL FLUMINENSE

ROBERTO WEIDMANN MENEZES

APLICAÇÕES DE CONTRATOS DE TRANSFORMAÇÃO

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Engenharia de Software

Orientador:
CHRISTIANO DE OLIVEIRA BRAGA

NITERÓI

2011

Roberto Weidmann Menezes

Aplicações de contratos de transformação

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Engenharia de Software

Aprovada em 17 de outubro de 2011.

BANCA EXAMINADORA

Prof. D.Sc. Christiano de Oliveira Braga - Orientador, UFF

Prof. D.Sc. Leonardo Gresta Paulino Murta, UFF

Prof. D.Sc. Franklin Ramalho, UFCG

Niterói

2011

Às minhas avós Traute Weidmann e Rosa Mozaner Barreto Menezes.

Agradecimentos

A Deus, pois sem ele nada seria possível.

À minha família, em especial aos meus pais, por acreditar em meu potencial, sempre me apoiando em todas as dificuldades.

Ao meu Orientador Christiano de Oliveira Braga por acreditar e viabilizar que eu concretizasse um bom mestrado.

Aos membros do grupo de pesquisa λSE, em especial ao Cássio Santos, pela ajuda durante o desenvolvimento de minha dissertação.

Aos membros da república estudantil *APJava*, principalmente ao Heliomar Kann da Rocha Santos, pelo companheirismo ao longo dos últimos 5 anos.

À Mariana Martins, pelos momentos de tranquilidade e por ajudar na revisão deste trabalho.

A todos os docentes que tive o prazer de conhecer no Instituto de Computação da UFF, especialmente à profa. Viviane Torres da Silva, por terem contribuído com minha formação tanto profissionalmente quanto pessoalmente.

Aos meus amigos e amigas pela felicidade e companheirismo em todos os momentos.

Ao CNPq pelo apoio financeiro.

Resumo

Desenvolvimento dirigido a modelo (DDM) é uma abordagem para desenvolvimento de software com enfoque em modelos e nas suas transformações para a geração de outros artefatos de software como código-fonte, documentação ou outros modelos em diferentes níveis de abstração. Contratos de transformação (CT) é uma abordagem para o desenvolvimento rigoroso de transformações de modelo que utiliza técnicas para verificação e validação formal de modelos e transformações de modelo. Baseia-se numa percepção simples, porém expressiva, de que uma transformação de modelos pode ser vista como um modelo e, como tal, técnicas de modelagem, verificação e validação aplicáveis a um modelo são também aplicáveis a uma transformação de modelos. Este trabalho apresenta duas aplicações de CT em contextos diferentes: (i) uma transformação de modelos descritos na *Unified Modeling Language* para *Enterprise Java Beans*; (ii) uma transformação de modelos descritos na *Seismic Domain Modeling Language* (SDML), uma linguagem de modelagem para aplicações sísmicas, para código-fonte paralelo. Estas aplicações exploram o uso de CT com linguagens de propósito geral e específicas de domínio, respectivamente. O CT para o domínio sísmico é resultado de um projeto de pesquisa em parceria com o *Schlumberger Brazil Research & Geoengineering Center*. Nas duas aplicações foram utilizadas duas técnicas de verificação e validação formais. A verificação de consistência é feita através de um mapeamento de modelos às teorias em lógica de descrição (DL, sigla em inglês), permitindo a verificação de consistência dos modelos e da transformação. A validação da semântica estática das linguagens de modelagem e das transformações é feita pela aplicação de invariantes, descritos na *Object Constraint Language* (OCL) sobre cada modelo. As principais contribuições deste trabalho são: (i) uma formalização do processo de DDM com CT utilizando OCL e DL; (ii) o desenvolvimento de uma infraestrutura de suporte, na forma de um biblioteca chamada `TCLib`, à implementação de transformações de modelo seguindo a abordagem de contratos de transformação; (iii) a incorporação de verificação de consistência baseada em lógica de descrição à `TCLib`; (iv) o desenvolvimento de dois experimentos não-triviais à técnica de CT fazendo uso da `TCLib`.

Palavras-chave: Contratos de transformação, transformação de modelos, desenvolvimento dirigido à modelos, métodos formais, engenharia de software.

Abstract

Model-driven development (MDD) is a software development approach that relies on models and its transformations to generate other artifacts such as source-code, documentation and other models at different levels of abstraction. Transformation contracts (TC) is a rigorous approach to the development of model transformations that uses techniques for formal verification and validation of models and model transformations. In the TC approach, a model transformation is interpreted as a model. Hence, model specification, verification and validations techniques are also applicable to a model transformation. This dissertation presents two applications of TC in different contexts: (i) a transformation of models described in the Unified Modeling Language to Enterprise Java Beans; (ii) a transformation of models described in Seismic Domain Modeling Language (SDML), a modeling language for seismic applications, to source code with support to parallelism. These applications exploit the use of TC with general-purpose and domain-specific languages, respectively. The latter application is the result of a research project in collaboration with Schlumberger Brazil Research & Geoengineering Center. Both applications used two techniques of formal verification and validation of models and model transformations. Consistency checking is achieved through a mapping from models to description logic (DL) theories. The validation of the static semantics of modeling languages and transformations is done by applying invariants, described in the Object Constraint Language (OCL). The main contributions of this work are: (i) a formalization of the MDD process with CT using OCL and DL; (ii) the development of a library called `TCLib` to support the implementation of model transformation using the transformation contracts approach; (iii) the inclusion of consistency checking based into `TCLib`; (iv) the development of two non-trivial experiments using TC and `TCLib`.

Keywords: Transformation contract, model transformation, model-driven development, formal methods, software engineering.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Processos de desenvolvimento de software [51] tradicionalmente priorizam a etapa de implementação, porque as tarefas do processo têm foco no código-fonte. Portanto, o código-fonte possui mais valor que uma especificação.

Desenvolvimento dirigido a modelos (DDM) é uma técnica de Engenharia de Software que utiliza modelos como principal artefato do processo de desenvolvimento de software. DDM ambiciona aumentar o nível de abstração para produção de um aplicativo, reforçando a importância da especificação, em vez de priorizar idiossincrasias da plataforma para implementação escolhida. Sendo assim, o código-fonte para uma aplicação é gerado a partir de transformações aplicadas diretamente sobre os modelos, isto é, especificações de uma aplicação.

Tradicionalmente, para garantir que o código-fonte seja válido, i.e., possua o comportamento esperado, utilizam-se diferentes métodos manuais e, muitas vezes, automáticos de verificação da especificação do software como, por exemplo, auditorias de software e diversas técnicas de teste. Em DDM, de forma semelhante, propõem-se diversas abordagens para garantir que uma transformação ocorra como esperado. Uma dessas abordagens é chamada de contratos de transformação.

Contratos de transformação [9, 11, 23] é uma técnica formal para a especificação, verificação e implementação de transformações de modelos. Um contrato de transformação é uma relação entre os domínios que uma transformação relaciona com as propriedades que tal relação deve cumprir. Um domínio é definido como um metamodelo de uma linguagem de modelagem em conjunto com as propriedades que instâncias deste metamodelo devem cumprir.

Uma maneira de entender os benefícios dos contratos de transformação é através

de uma comparação com a técnica de Desenho por Contrato (DbC) [41], definida por Bertrand Meyer para a validação de programas orientados a objeto. Um contrato de transformação é executado para cada aplicação de uma transformação de modelo que o implementa. Por isso, as transformações do modelo podem ser validadas através de assertivas, representando as propriedades desejadas, através de raciocinadores acoplados no transformador. Nesse caso, uma implementação da transformação termina abruptamente quando uma propriedade falhar. Em DbC, os programas são anotados com assertivas e geram exceções quando falham. Ou seja, CT são uma tentativa de mimetizar DbC no nível de modelagem. Portanto, os contratos de transformação têm os mesmos benefícios para o desenvolvedor transformação que DbC fornece para um programador que adotá-lo.

Num contrato de transformação, uma transformação de modelo é definida por um metamodelo. Consequentemente, as mesmas técnicas utilizadas para verificar e validar modelos podem ser utilizadas na verificação e validação de transformadores de modelos. Por fim, esta abordagem não depende de uma determinada tecnologia, como *Query-View-Transformation* (QVT, [42]), permitindo a sua aplicação em qualquer transformação.

Trabalhos anteriores do λSE, grupo de pesquisa em Engenharia de Software orientada a linguagens[1], foram utilizados com sucesso na área de segurança dirigida a modelos [23, 11]. Até o momento, não foram publicados trabalhos com CT envolvendo linguagens de propósito geral, como UML, ou transformações para problemas reais. Além disso, em [23, 11] foram utilizadas apenas a validação por assertivas escritas na *Object Constraint Language* (OCL) [57], uma linguagem para restrições sobre modelos UML, carecendo de outros mecanismos de verificação de outras propriedades também importantes como, por exemplo, a consistência dos modelos através de lógica de descrição [4].

1.1 Objetivos e contribuições

O objetivo deste trabalho é explorar o conceito de contratos de transformação através de seu uso em diferentes contextos. Além disso, formalizar CT com a verificação e validação através da OCL e da lógica de descrição. Assim, pretende-se apresentar a evolução do estado da arte em contratos de transformação, além de propor os possíveis caminhos a serem seguidos.

A primeira contribuição envolve a adição de uma técnica de validação ao processo de transformação. A técnica proposta analisa a consistência de um modelo utilizando lógica

[1] Para mais informações sobre o grupo de pesquisa λSE, acesse `http://lse.ic.uff.br`.

de descrição. Assim, a verificação de consistência garante que um modelo é instanciável enquanto as assertivas em OCL validam uma instância particular de um metamodelo em relação a seus invariantes. Essas técnicas de verificação são complementares. No Capítulo 2 essa complementaridade é discutida. Consequentemente, também é necessário formalizar o processo de transformação com contratos de transformação utilizando os validadores abordados. Diferente de [23], este trabalho apresenta uma generalização do processo de transformação com contratos de transformação, junto com a sua formalização.

Outra contribuição é a apresentação de duas diferentes aplicações não-triviais de contratos de transformação, com transformadores fora do domínio de segurança, diferente de [23]. Assim, é possível atestar a aplicabilidade desta abordagem de especificação de transformações em diferentes contextos, que são:

- Uma abordagem da geração de aplicações distribuídas a partir de um diagrama de classe UML, sem qualquer estereótipo, conforme será apresentada no Capítulo 3. A intenção é gerar uma estrutura para a criação de aplicações orientadas a componentes, utilizando o framework EJB [53], que permite uma infra-estrutura distribuída *multi-tier* e escalável. Para tal, utiliza-se a linguagem induzida por EJB e UML para analisar o impacto do uso de contratos de transformação com linguagens de propósito geral.

- Uma abordagem para a criação de aplicações paralelas para processamento de dados sísmicos, a ser apresentada no Capítulo 4. Para tal, foi especificado um transformador que relaciona duas linguagens específicas de domínio criadas especialmente para este problema. Então, foram utilizados contratos de transformação em um caso relacionado com a indústria petrolífera, em parceria com a *Schlumberger Brazil Research & Geoengineering Center* (BRGC), para simplificar o desenvolvimento de aplicações que, além de complexas, podem causar grandes prejuízos financeiros se erros não forem detectados e corrigidos.

Para o desenvolvimento desses transformadores foi proposto um padrão de projeto para a aplicação de contratos de transformação, assim como a sua implementação em Java como uma biblioteca chamada `TCLib`, conforme apresentados no Capítulo 2. Logo, `TCLib` foi utilizado pelos transformadores abordados neste trabalho.

Por fim, são analisados os resultados obtidos pelo uso de contratos de transformação nas abordagens supracitadas, para atestar os benefícios obtidos com o seu uso.

1.2 Organização

O restante desta dissertação está organizado da seguinte maneira.

O Capítulo 2 aborda todo o conhecimento necessário para a compreensão do trabalho, que envolve discussões sobre: (i) desenvolvimento dirigido a modelos e sua formalização; (ii) contratos de transformações, além de um padrão de projeto para seu uso e as técnicas de validação utilizadas.

A primeira aplicação de contratos de transformação, apresentada no Capítulo 3, consiste em uma abordagem para transformação para sistemas distribuídos. Para tal, é utilizado diagramas de classe UML para a geração de código-fonte em Java, utilizando o framework EJB.

Já no Capítulo 4, é apresentada uma transformação de modelo para aplicações sísmicas, realizada em parceria com a BRGC. Nesse caso, a transformação permite a geração de código-fonte paralelo em C#, usando as especificações de um framework específico, a partir de um diagrama de classe UML, estereotipado de acordo com uma linguagem para processamento de dados sísmicos.

O Capítulo 5 apresenta os trabalhos relacionados aos conceitos apresentados nesta dissertação e suas aplicações. Além disso, nesse capítulo é apresentado uma comparação entre contratos de transformação e QVT, para ressaltar os benefícios obtidos em relação a uma especificação de implementação importante na área.

Finalmente, o Capítulo 6 conclui esta dissertação, sumarizando as contribuições alcançadas, discutindo as limitações observadas e apresentando possíveis trabalhos futuros, referentes tanto a técnica de contratos de transformação quanto aos transformadores discutidos.

Capítulo 2

Transformações de modelo com contratos de transformação

Neste capítulo é apresentado o embasamento teórico necessário para a compreensão deste trabalho. Para tal, é importante apresentar uma introdução sobre desenvolvimento dirigido a modelo, na Seção 2.1, e a sua formalização utilizando conceitos de linguagens e métodos formais, na Seção 2.2. Então, na Seção 2.3 é introduzida a abordagem de contratos de transformação, destacando as propriedades pertencentes ao contrato e um padrão de projeto para a sua implantação. Por fim, são apresentadas, na Seção 2.4, as considerações finais relacionadas com os tópicos abordados.

2.1 Desenvolvimento dirigido a modelo

Desenvolvimento dirigido a modelo (DDM) é uma abordagem para o desenvolvimento de software, onde modelos de um sistema são artefatos vivos, i.e., um modelo não é apenas um documento passivo, mas pode ser usado como entrada para uma ferramenta que, por exemplo, o transforma em código-fonte compilável. Esse tipo de transformação é conhecida como *model-to-code* ou, em português, modelo-para-código [25]. Outros modelos também podem ser produzidos a partir de um modelo, nas chamadas transformações modelo-para-modelo.

A intenção de DDM é aumentar o nível de abstração no processo de desenvolvimento, para que desenvolvedores não sejam programadores, mas sim especialistas no domínio em que atuam e para que se preocupem apenas com a modelagem. A modelagem específica de domínio auxilia a alcançar esse objetivo.

Segundo [40], um modelo é um conjunto coerente de elementos formais, descrevendo

algo, como, por exemplo, um software, construído com alguma finalidade. Um modelo, quando formalmente definido, é passível de análise automatizada (como validação de invariantes ou verificação de consistência) e transformação em uma implementação. Para tanto, um modelo deve respeitar uma estrutura específica, dada por um *metamodelo*.

Um metamodelo é a descrição da sintaxe de uma linguagem de modelagem. Um modelo, que é uma instância de um metamodelo, é dito bem-formado em relação ao metamodelo dado. É interessante notar que através da relação de instanciação pode-se construir uma hierarquia de metamodelagens tão profunda quanto se queira. Informalmente, um metamodelo é uma instância de um modelo que representa o meta-metamodelo, que por sua vez é instância de um modelo que representa o meta-meta-metamodelo, e assim sucessivamente.

Outro conceito importante em DDM é o de transformação de modelo, que pode ser vista como uma relação entre metamodelos.

Neste trabalho, são consideradas transformações de modelos entre dois diferentes metamodelos, chamados de origem e destino, que devem ser aplicadas direcionalmente da origem para o destino. Segundo a classificação definida em [25], isto seria uma transformação de modelo-para-modelo exógena e unidirecional. Essa abordagem é essencialmente uma função que relaciona elementos entre dois metamodelos dados.

Uma transformação opera sobre níveis de abstração, atuando nas seguintes combinações: (i) diminuindo o nível de abstração, utilizado para geração de código-fonte a partir de modelos; (ii) aumentando o nível de abstração, através da engenharia reversa; (iii) mantendo o mesmo nível de abstração, criando outros modelos ou documentação complementar. Este trabalho se concentra na categoria (i).

DDM possui diversos benefícios, conforme apresentado seguir, que foram adaptados de [37, p. 9].

- **Produtividade**: O desenvolvedor da aplicação concentra seus esforços na especificação do problema, já que não precisa implementá-lo diretamente. Além disto, detalhes específicos da plataforma já estão especificados na transformação, evitando a definição de características ligadas a plataforma de destino. Por fim, apesar da complexidade em desenvolver um transformador, esta atividade é realizada uma única vez.

- **Portabilidade**: Um único modelo pode ser transformado para diferentes plataformas. Assim, mudanças de tecnologias são triviais, desde que o modelo seja suficiente

para especificar o sistema e a transformação seja completa, i.e., o artefato final não precisa de customizações.

- **Interoperabilidade**: As chamadas pontes, ou *bridges*, permitem a comunicação entre artefatos gerados por diferentes transformações, a partir de um mesmo modelo de entrada. Uma transformação de modelos relaciona elementos de diferentes domínios. Ao executar, por exemplo, duas transformações distintas a partir de um mesmo modelo, é possível relacionar os elementos transformados de diferentes artefatos, porque foram originados de um mesmo elemento.

- **Manutenção e documentação**: O modelo é o principal artefato utilizado do processo de desenvolvimento. Como o artefato final, e.g., código-fonte, reflete os modelos utilizados, a documentação do projeto está sempre atualizada. Idealmente, mudanças no sistema serão realizadas diretamente nos modelos, que permitirá gerar uma nova versão do artefato final. Assim, a documentação sempre estará atualizada, o que auxilia na manutenção do software.

Nas próximas seções, a visão de modelos como linguagens é explorada, permitindo formalizar e acrescentar novos conceitos amplamente estudados ao desenvolvimento dirigido a modelo.

2.2 Formalizando DDM

Segundo [37, cap. 2], um modelo é sempre descrito em uma linguagem, seja ela UML, uma linguagem natural ou de programação. Para uma transformação automática, deve-se utilizar apenas modelos descritos em linguagens bem-formadas[1].

Um modelo é considerado *sintaticamente bem-formado* (ou simplesmente bem-formado, como foi abordado previamente) em relação a um metamodelo, quando ele segue a estrutura definida por tal. Formalmente, em termos algébricos, um modelo m é bem-formado com relação a um dado metamodelo M, denotado por $m \in M$, quando m é um termo na álgebra induzida por M, cuja assinatura é dada pelos elementos de modelo de M e as equações representam restrições de cardinalidade das associações de M.

Um modelo m é dito *em conformidade* em relação a um dado metamodelo M, denotado por $m \models P_M$ onde P_M representa as propriedades de M, se, e somente se, m é

[1] De acordo com [37], uma linguagem bem-formada possui uma sintaxe bem-definida e semântica que pode ser interpretada automaticamente por um computador.

bem-formado em relação a M e as *propriedades* do metamodelo M se cumprem em m. Um exemplo de propriedade de um metamodelo de diagramas de classe UML é que, qualquer cadeia de herança em um modelo dado não pode ter ciclos. Portanto, para que um modelo de classe m seja considerado em conformidade com o metamodelo de diagrama de classe UML, é necessário (mas não suficiente) que m não possua ciclos em suas hierarquias de herança. Uma maneira de entender tais propriedades é especificá-las como *invariantes* sobre o metamodelo.

Dada uma transformação de modelo $\tau : M \rightarrow M'$, a sua corretude é formalizada por:

$$\forall \hat{m} \in M, \hat{m}' \in M', \hat{m}' = \tau(\hat{m})$$
$$(\hat{m} \models P_M \wedge pre_\tau(\hat{m}) \Rightarrow \hat{m}' \models P_{M'} \wedge post_\tau(\hat{m}, \hat{m}'),$$

onde M e M' são metamodelos, $\hat{m} \in M$ significa que $\hat{m}$ é bem-formado a M; $\hat{m} \models P_M$ simboliza que $\hat{m}$ está em conformidade com a linguagem de modelagem associada a M de acordo com as propriedades P_M; $pre_\tau : M \rightarrow boolean$ representa a pré-condição para a transformação τ em uma instância do metamodelo M; $post_\tau : M \times M' \rightarrow boolean$ denota a pós-condição de τ, que atua sobre instâncias de M e M'.

Sob uma perspectiva de linguagens de programação, um transformador de modelos, assim com um compilador, pode ser entendido como um caso particular de um transformador de programas [24]. Consideramos que uma linguagem de modelagem possui uma descrição de sua sintaxe concreta e de sua sintaxe abstrata. Ao escolher UML como metalinguagem, a sintaxe concreta de uma linguagem de modelagem é descrita como um *profile* e a sintaxe abstrata como um metamodelo. Sob essa perspectiva, o processo de desenvolvimento dirigido a modelos pode ser apresentado de forma semelhante ao processo de transformação de programas, como a seguir:

$$m \in M_{sintaxe\ \ concreta} \xrightarrow{parse} \hat{m} \in M_{sintaxe\ \ abstrata}$$
$$\hat{m}' \in M'_{sintaxe\ \ abstrata} \xrightarrow{pretty\ \ print} m' \in M'_{sintaxe\ \ concreta}$$

onde m, m', $\hat{m}$ e $\hat{m}'$ são modelos; M e M' representam os metamodelos de origem e destino da transformação de modelos τ; $m \in M$ denota que o modelo m é bem-formado em respeito a M; *parse* é o mapeamento no qual um dado modelo m produz uma instância $\hat{m}$ de M, onde $\hat{m}$ é bem-formado em relação ao metamodelo M; *pretty print* é o mapeamento inverso do *parse*. O *parse* e o *pretty print* podem ser interpretados como transformações, repectivamente, de código-para-modelo e modelo-para-código.

A perspectiva de linguagens de programação aplicada ao DDM fornece mecanismos importantes para a validação e verificação de transformadores de modelos, conforme será apresentado na Seção 2.3.

2.3 Contratos de transformação

Uma vez que modelos são a base para o DDM, sua validação e verificação consistem em etapas importantes e fundamentais do processo de desenvolvimento. Erros durante suas especificações podem resultar em problemas durante todo o processo de transformação, gerando artefatos incorretos. Além disso, a própria transformação de modelos também precisa ser validada, pois é uma tarefa complexa e propensa a erros.

Contratos de transformação é uma abordagem para transformação de modelos cujo principal objetivo é acoplar à transformação de modelos mecanismos de validação e verificação, de forma transparente e formal [2]. Ou seja, o contrato é uma especificação *do que* uma transformação de modelo deve fazer, e não *como*. Um contrato de transformação é um modelo de transformação [8, 11] que especifica associações representando regras de transformação que relacionam elementos do metamodelo de entrada com o de saída. Em [11], o contrato de transformação possui as relações entre os metamodelos de entrada e saída. Ele pode ser definido formalmente como a união disjunta do metamodelos de entrada e saída, ou seja, $M_K = M \bowtie_A M'$ dado que M_K, M e M' são, respectivamente, o modelo de transformação e os metamodelos de entrada e saída, e associações $a \in A$ entre elementos de modelo de M e M'.

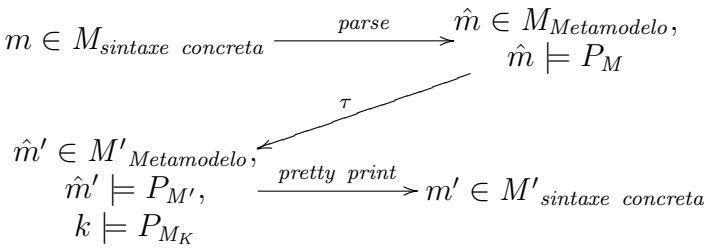
Uma outra interpretação de um contrato de transformação é vê-lo como uma operação de composição de (meta)modelos que realiza a união disjunta dos metamodelos relacionados por uma transformação com um conjunto A de associações entre eles. Não é definido, neste trabalho, uma linguagem para especificar contratos. Eles são descritos na mesma linguagem utilizada para descrever os metamodelos relacionados por uma transformação de modelos, isto é, diagramas de classe UML.

Adicionalmente, as associações em A são representadas por classes, modularizando os relacionamentos entre os elementos do metamodelo de transformação, facilitando, acredita-se, a legibilidade do modelo de transformação. Um exemplo é o diagrama de

[2] Por ser genérico, um contrato de transformação permite a especificação, verificação e implementação de modelos estruturais e comportamentais, apesar deste trabalho utilizar apenas modelos estruturais. Para modelos comportamentais, o contrato especificaria também as transições entre configurações de estados definidos por um modelo.

classes da Figura 3.5 na página 48. As classes no retângulo rotulado *UMLEJB* representam associações do contrato de transformação que relacionam elementos de modelo do metamodelo UML com elementos de modelo do metamodelo EJB.

O processo de transformação de modelos com contratos de transformação pode ser apresentado, formalmente, da seguinte forma:



onde M, M' e M_K simbolizam, respectivamente, os metamodelos de entrada e saída e o modelo de transformação; k é a instância do modelo de transformação, ou seja, a relação entre os elementos dos modelos de entrada $\hat{m}$ e de saída $\hat{m}'$; P_M são as propriedades do metamodelo da linguagem de modelagem M; $m \models P_M$ significa que todas as propriedades em P_M são válidas no modelo $m \in M$.

Dado um contrato de transformação $K = M \bowtie_A M'$, sendo A o conjunto de associações induzidas pelo modelo de transformação, a corretude pode ser definida por:

$$\forall \hat{m} \in M, \hat{m}' \in M', c = \hat{m} \bowtie_l \hat{m}', l \in A$$
$$(\hat{m} \models P_M \Rightarrow \hat{m}' \models P_{M'} \wedge k \models P_K),$$

onde $M \bowtie_A M'$ denota a união disjunta dos metamodelos M e M' juntos com o conjunto de associações A; $\hat{m} \bowtie_l \hat{m}'$ com $l \in A$ significa a união disjunta dos modelos de objeto $\hat{m}$ e $\hat{m}'$ juntos com o conjunto de links l instâncias das associação A.

Então, a corretude é garantida quando a conjunção das propriedades dos metamodelos de origem, destino e de transformação são mantidas, i.e., os modelos envolvidos estão em conformidade com seus metamodelos. Portanto, a conformidade dos modelos implica na corretude da transformação.

A execução de um contrato de transformação pode ser definida algoritmicamente, representada no Algoritmo 1, onde também é definida a validação das instâncias dos metamodelos de origem e destino.

Se algum passo dado pelo Algoritmo 1 não for atendido, o processo de transformação é interrompido. Ou seja, a geração do artefato final ocorre somente se todos os modelos envolvidos estão em conformidade aos seus metamodelos e, consequentemente, corretos.

Algoritmo 1 Algoritmo do processo de transformação de um contrato

1: Constrói uma instância $\hat{m}$ do metamodelo de origem M a partir do modelo de entrada m.
2: Valida $\hat{m}$ de acordo com as propriedades P_M do metamodelo de origem M.
3: Transforma $\hat{m}$ em uma instância $\hat{m}'$ do metamodelo de destino, através do modelo de transformação $K = M \bowtie_A M'$.
4: Valida a instância $\hat{m}'$ do metamodelo de destino M', de acordo com as propriedades $P_{M'}$.
5: Valida a instância k do modelo de transformação K, de acordo com as propriedades P_K.
6: Faz o *pretty-printing* da instância $\hat{m}'$ do metamodelo de destino M'.

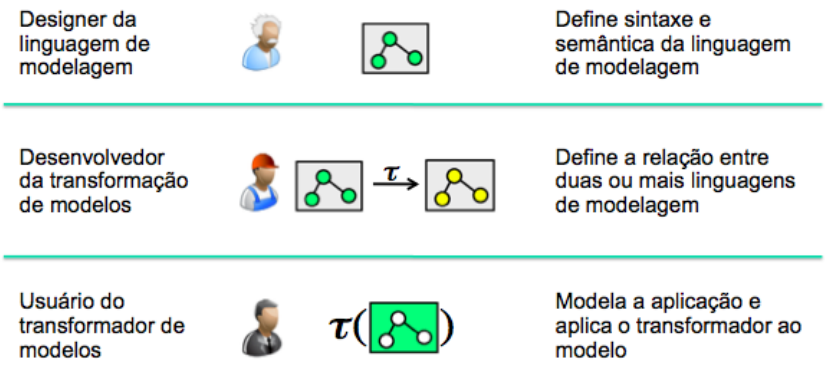


Figura 2.1: Usuários beneficiados pelo uso de contratos de transformação, dado o seu título e função

O processo de transformação apresentado beneficia, basicamente, três tipos de usuários diretamente ligados a qualquer transformador: (i) designer da linguagem de modelagem; (ii) desenvolvedor da transformação de modelos; (iii) usuário do transformador. A Figura 2.1 apresenta mais detalhes sobre os tipos de usuários.

O designer da linguagem de modelagem é o primeiro tipo de usuário afetado pelo uso de contratos de transformação. Ele é responsável pela definição da sintaxe abstrata e a sua relação com a sintaxe concreta de uma linguagem, i.e., define o processo de *parsing* e *pretty-print*. O designer também determina as propriedades do metamodelo, que serão utilizadas para a validação de suas instâncias.

O desenvolvedor da transformação define a relação entre duas diferentes linguagens, i.e., especifica o modelo de transformação e as suas propriedades. Para tal, ele precisa conhecer as linguagens envolvidas, porque idiossincrasias referentes à plataforma de destino serão atendidas durante a transformação.

Por fim, o usuário do transformador utiliza a ferramenta elaborada pelo trabalho conjunto dos outros tipos de usuários abordados. Neste momento, o contrato de transfor-

mação é utilizado em casos reais, garantindo a corretude da transformação e gerando um artefato final confiável.

Esses três tipos de usuários podem identificar problemas nos seguintes casos: (i) nos artefatos em que atuam, como um designer da linguagem de modelagem especificando as propriedades de uma dada linguagem e encontra erros na sua metamodelagem; (ii) nos artefatos definidos por um usuário anterior no processo de desenvolvimento, como um designer da transformação que, durante os testes da transformação em desenvolvimento, descobriu falhas na linguagem de modelagem.

Além disso, cada usuário pode encontrar as seguintes situações relacionadas aos modelos utilizados: (i) um modelo é mal-formado em respeito ao seu metamodelo, ou seja, ele não obedece à estrutura sintática adequada; (ii) um modelo é bem-formado em relação ao seu metamodelo mas não está em conformidade com suas as propriedades; (iii) um modelo é bem-formado e está em conformidade com as propriedades de seu metamodelo. As propriedades pertencentes ao metamodelo, durante as etapas de validação, auxiliam a identificar os modelos que atendem aos itens (i) e (ii), o que inclui a própria transformação de modelo já que ela é vista como um modelo.

2.3.1 Propriedades pertencentes ao contrato de transformação

A semântica de uma linguagem de modelagem é simbolizada pelas suas propriedades, que devem ser respeitadas por suas instâncias, através da relação de conformidade.

As propriedades representam aspectos estáticos ou dinâmicos de uma linguagem de modelagem [10, 36], que podem ser especificadas, por exemplo, em: (i) OCL, para a semântica estática; (ii) lógica de descrição, para verificação da consistência, uma propriedade também relacionada à semântica estática; (iii) lógica temporal, para semântica dinâmica; (iv) lógica de reescrita, com subteorias equacionais para representar aspectos estáticos e a subteorias baseadas em regras para a semântica dinâmica.

Este trabalho utiliza OCL e lógica de descrição para definir propriedades da semântica estática das linguagens de modelagem e são discutidas nas Seção 2.3.1.1 e na Seção 2.3.1.2.

2.3.1.1 Invariantes na *Object Constraint Language*

OCL [57], ou *Object Constraint Language*, é uma linguagem definida pela OMG para especificar restrições em modelos UML. Para tal, é possível navegar pelas associações definidas num modelo. Uma expressão OCL é avaliada sobre um modelo de objetos (ou

snapshots) instância de um dado modelo e não produz efeito colateral sobre o modelo sendo avaliado. OCL é uma linguagem declarativa fortemente tipada.

Em diagramas UML estáticos, pode-se utilizar OCL para a definição de invariantes que restringem os elementos de modelo de um diagrama. A forma geral de um invariante é apresentada a seguir:

context $\mathcal{E}$ **inv** $\mathcal{I}$:
$\mathcal{R}$

onde $\mathcal{E}$ simboliza o contexto, dado por uma classe, sobre o qual o invariante será aplicado, i.e., o tipo dos elementos que serão verificados; $\mathcal{I}$ é o identificador do invariante, que é opcional; a restrição $\mathcal{R}$, que faz uso da palavra *self* para referenciar o elemento sobre o qual $\mathcal{R}$ é aplicado.

Os tipos que OCL suporta são: (i) os tipos básicos *String*, *Integer*, *Boolean* e *Real*; (ii) tipos enumerados; (iii) tipos compostos representando coleções, entre eles *Set*, *OrderedSet*, *Bag* e *Sequence*; (iv) tipos especificados na UML e nos modelos utilizados. É importante notar que, dentre eles, apenas os tipos definidos no item (iv) podem ser utilizados como contexto de um invariante.

Para especificar uma restrição, OCL define diferentes expressões que podem ser aplicadas a coleções. As expressões utilizadas neste trabalho são:

- $select(e : E \mid c)$: Permite selecionar elementos do tipo E, iterando sobre a variável e, que atendem a condição c;
- $forAll(e : E \mid c)$: Verifica se uma condição c é válida em todos os elementos de um conjunto do tipo E, através da variável de iteração e;
- *if c then ct else cf endif*: É uma estrutura de desvio de fluxo de controle. Se a condição c é válida, a expressão ct é avaliada. Caso contrário, avalia-se a expressão cf;
- c *implies* i: Implicação lógica entre as expressões boleanas c e i;
- $collect(e : E \mid c)$: Dado um conjunto de elementos do tipo E, coleta-se elementos informados em c, a partir de e;
- $size()$: Calcula o tamanho de um conjunto de elementos;
- $isEmpty()$ / $notEmpty()$: Verifica se um conjunto de elementos está vazio ou não;

- *includes*(e) / *excludes*(e): Verifica se o elemento e está ou não em um conjunto de elementos;
- *includesAll*(s): Verifica se um conjunto de dados possui todos os elementos do conjunto s;
- *including*(e) / *excluding*(e): Produz um conjunto contendo todos os elementos do conjunto sobre o qual a expressão foi aplicada, incluindo (excluindo) o elemento e;
- *symmetricDifference*(s): Calcula a diferença simétrica entre um conjunto de elementos com o conjunto s;
- *exists*($e : E \mid c$): Verifica se algum elemento de um conjunto do tipo E, iterável pela variável e, atende a condição c;
- *asSet*(): Transforma um conjunto de elementos em um *Set*.

O operador de aplicação de expressões sobre coleções é "$->$". Por exemplo, $C->$ *select*($c|P(c)$) seleciona todos os elementos de C que satisfazem a condição estabelecida por P.

OCL define também operações para verificação de tipos e *casting*. São elas: (i) $e.oclIsTypeOf(E)$, que verifica se o elemento e é do tipo E ; (ii) $e.oclIsKindOf(E)$, semelhante a $oclIsTypeOf(E)$, também considera a hierarquia de herança; (iii) $e.oclAsType(E)$, faz um *type casting* de e para o tipo E. Note que o operador de aplicação destas expressões é "." ao invés de "$->$", como nas expressões sobre coleções.

Neste trabalho as seguintes operações são utilizadas sobre o tipo básico *String*, também aplicadas através do operador "." : (i) $s.size()$, calcula o tamanho da *String* s; (ii) $s.substring(i, f)$, que cria uma nova *String* formada pelos caracteres entre os índices i e f de s.

Em relação a operações básicas de uma linguagem, tem-se: (i) negação de booleano, pela palavra reservada *not*; (ii) igualdade e desigualdade de elementos e valores, por $=$ e $<>$; (iii) comparação entre valores numéricos por $>$, $>=$, $<$ e $<=$; (iv) as operações lógicas da álgebra booleana *and*, *or* e *xor*.

A semântica estática de uma linguagem de modelagem pode ser descrita usando OCL e pode ser utilizada para validar modelos de forma automática, por exemplo executando-a sobre modelos vistos como instâncias de um metamodelo.

A validação baseada em invariantes é uma abordagem similar a *Desenho por Contrato* (DbC), em inglês *design by contract*, apresentado por Meyer [41]. Em DbC, programas são anotados com assertivas e disparam exceções em caso de falha. A validação proposta segue o mesmo conceito, utilizando de invariantes em OCL como assertivas. No contexto de uma aplicação de uma transformação de modelos, a transformação é interrompida caso uma validação falhe, isto é, se falhar a execução de um invariante sobre um modelo visto como instância de um metamodelo. Assim, contratos de transformação possuem os mesmos benefícios para desenvolvedores de transformação que DbC fornece aos programadores.

2.3.1.2 Consistência em lógica de descrição

Lógica de descrição (DL, [4]) é uma família de lógicas eficientemente decidíveis para representação de conhecimento, através da definição dos conceitos relevantes de um domínio para especificar propriedades de objetos e indivíduos, formando uma base de conhecimento. Formalmente, DL é um fragmento da lógica de primeira ordem que permite raciocínio decidível.

Uma base de conhecimento é composta por dois componentes: (i) *TBox*, que introduz a terminologia, i.e., vocabulário do domínio de aplicação; (ii) *ABox*, contendo assertivas sobre indivíduos nomeados de acordo com a terminologia.

Um vocabulário consiste em conceitos (em inglês, *concepts*), que denotam conjuntos de indivíduos, e papéis (em inglês, *roles*), que denotam relações binárias entre conceitos. Subsunção define uma relação de inclusão entre conceitos, denotada pela fórmula $C_1 \sqsubseteq C_2$, significando que o conceito C_1 subsume o conceito C_2, isto é, C_1 está incluído em C_2. A fórmula $C_1 \equiv C_2$ denota equivalência de conceitos e é uma simplificação para $C_1 \sqsubseteq C_2 \sqcap C_2 \sqsubseteq C_1$.

Dois tipos de raciocínio são possíveis em DL. O primeiro consiste em raciocinar sobre a *ABox*, para verificar se os axiomas de uma base de conhecimento, i.e., *TBox*, são satisfeitos num conjunto específico de conceitos e papeis (também chamados de indivíduos) dado pela ABox . A outra análise raciocina sobre a *TBox*, e realiza um processamento geral, simbólico, que verifica se existe uma ABox que satisfaça a TBox.

O poder de expressividade de uma DL é inversamente proporcional a sua eficiência ao raciocinar. A família $\mathcal{ALC}$, que significa em inglês *attributive language with complements* foi introduzida em [49] como uma linguagem mínima para o uso prático. A seguir é apresentada a sintaxe referente a linguagem $\mathcal{ALC}$.

$$
\begin{array}{rcrcl}
C, D & \longrightarrow & A & | & \text{(conceito atômico)} \\
 & & \top & | & \text{(conceito universal ou } top\text{)} \\
 & & \bot & | & \text{(conceito vazio ou } bottom\text{)} \\
 & & \neg C & | & \text{(negação de conceito)} \\
 & & C \sqcap D & | & \text{(conjunção de conceitos)} \\
 & & C \sqcup D & | & \text{(disjunção de conceitos)} \\
 & & \forall R.C & | & \text{(restrição de valor)} \\
 & & \exists R.C & | & \text{(quantificação existencial),}
\end{array}
$$

onde C e D são conceitos; A é um conceito atômico, i.e., não é composto por outros conceitos; R é um papel. A partir desta linguagem, uma família de lógicas de descrição surgiu, conhecida como família de $\mathcal{AL} - languages$.

Raciocínio por consistência é um procedimento comumente associado ao raciocínio em lógica de descrição. Em [7] é apresentado que esta inferência pode ser feita em diagramas de classe UML, quando devidamente mapeado para DL, provando-os instanciáveis.

A codificação de diagramas de classe em lógica de descrição essencialmente relaciona classes à conceitos DL e associações à papéis. Para tal, deve-se utilizar uma lógica com expressividade suficiente para a representação adequada dos elementos, conhecida como lógica $\mathcal{ALCQI}$, uma extensão da lógica $\mathcal{ALC}$ com a adição das seguintes características: (i) quantificação existencial de conceitos, através de $\exists^{\geq n} R.C$ e $\exists^{\leq n} R.C$; (ii) definição de inversão de papeis, por P^-. A seguir é apresentada a sintaxe da lógica $\mathcal{ALCQI}$.

$$
\begin{array}{rcrcl}
C, D & \longrightarrow & A & | & \text{(conceito atômico)} \\
 & & \top & | & \text{(conceito universal ou } top\text{)} \\
 & & \bot & | & \text{(conceito vazio ou } bottom\text{)} \\
 & & \neg C & | & \text{(negação de conceito)} \\
 & & C \sqcap D & | & \text{(conjunção de conceitos)} \\
 & & C \sqcup D & | & \text{(disjunção de conceitos)} \\
 & & \forall R.C & | & \text{(restrição de valor)} \\
 & & \exists R.C & | & \text{(quantificação existencial de conceitos)} \\
 & & \exists^{\geq n} R.C & | & \text{(quantificação existencial de conceitos)} \\
 & & \exists^{\leq n} R.C & | & \text{(quantificação existencial de conceitos)}
\end{array}
$$

$$
\begin{array}{rcrcl}
P & \longrightarrow & P & | & \text{(papel atômico)} \\
 & & P^- & | & \text{(inversão de papeis)}
\end{array}
$$

Em [7, Seção 7.1] é detalhado um mapeamento de diagramas de classes a TBoxes em $\mathcal{ALCQI}$ e que é utilizado neste trabalho. A seguir é apresentado um subconjunto de tais regras, que englobam classes, herança e associações binárias:

- Uma *classe* C é representada pelo conceito atômico C;
- Uma *generalização* entre uma classe C e sua classe filha C_1 pode ser representada usando a assertiva de inclusão $C_1 \sqsubseteq C$. A hierarquia de uma classe pode ser representada pelas assertivas $C_1 \sqsubseteq C, \ldots, C_n \sqsubseteq C$ quando C_i herda de C. Uma disjunção[3] entre classes $C_1, \ldots, C_n$ pode ser modelada como $C_i \sqsubseteq \bigsqcap_{j=i+1}^{n} \neg C_j$, com $1 \leq i \leq n-1$, enquanto a restrição de cobertura[4] pode ser expressada por $C \sqsubseteq \bigsqcup_{i=1}^{n} C_i$;
- Cada associação binária (ou agregação) A entre as classe C_1 e C_2, com multiplicidades $m_l..m_u$ e $n_l..n_u$ em cada ponta, respectivamente, é representada pelo papel atômico A, junto com a assertiva de inclusão $\top \sqsubseteq \forall A.C_2 \sqcap \forall A^-.C_1$. As multiplicidades são formalizadas pelas assertivas $C_1 \sqsubseteq (\exists^{\geq n_l} A.\top) \sqcap (\exists^{\leq n_u} A.\top)$ e $C_2 \sqsubseteq (\exists^{\geq m_l} A^-.\top) \sqcap (\exists^{\leq m_u} A^-.\top)$.

Por fim, o conceito de inconsistência é denotado por $C \sqsubseteq \bot$, onde C é um conceito e $\bot$ é o conceito vazio.

Tal técnica para raciocínio por consistência foi incorporada neste trabalho como um validador. A intenção é garantir a consistência de modelos antes da validação por invariantes pois é desnecessário validar um modelo, em relação ao seu metamodelo, se ele não for instanciável.

Para verificar a consistência de um modelo $m \in M$, é necessário defini-lo em lógica de descrição, conforme proposto em [7] para representar diagramas de classe UML em $\mathcal{ALCQI}$, este foi entendido, neste trabalho, para suportar a verificação de instâncias de um metamodelo. Esta técnica pode ser aplicada a qualquer metamodelo, ou seja, independe do metamodelo UML apresentado na Seção 3.1.1 (Na verdade, o mapeamento proposto em [7] basea-se na perspectiva conceitual de um diagrama de classe UML [28] e não possui qualquer relação com o metamodelo UML utilizado neste trabalho.). O mapeamento depende do tipo de raciocínio a ser usado, que são:

[3] Uma disjunção determina quais subtipos de um elemento são disjuntos entre si, i.e., o conjunto de elementos são mutualmente exclusivos.

[4] A restrição de cobertura impõe que um tipo é a união de todos os seus subtipos e nada mais.

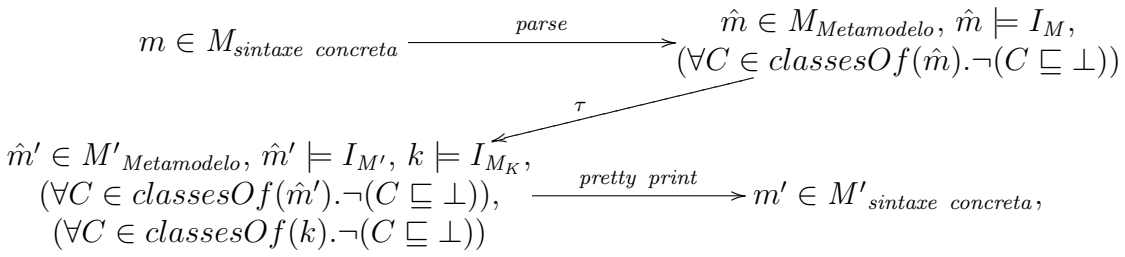
- Análise de *ABox*: Permite raciocinar sobre um conjunto específico de indivíduos de conceitos e papeis, i.e., verificar se m respeita as propriedades de M, dado que m e M são, respectivamente, representados como *ABox* e *TBox*.

- Análise de *TBox*: Realiza o processamento sobre uma *TBox* para verificar se os seus axiomas geralmente são satisfatíveis, e não somente para um conjunto particular de indivíduos. Para tal, um modelo m será uma extensão da *TBox* [14] que representa o metamodelo M. Assim, os axiomas representando m são definidos da seguinte forma, considerando que m é bem-formado em relação a M: (i) conceitos que representam as metaclasses em M são subsumidos por conceitos que representam objetos, da classe apropriada, em m; (ii) papéis representando associações em M são subsumidos por papéis que representam links, da associação apropriada, entre objetos de m; (iii) axiomas para restringir os papeis que representam links em m.

O verificador de consistência utilizado realiza a análise de *TBox*, por ser mais abrangente e atesta a consistência do metamodelo e do modelo validado, independentemente dos objetos que serão instanciados. Assim, garante-se que a linguagem de modelagem admite modelos e que o modelo apresentado é instanciável, i.e., admite diferentes cenários.

2.3.2 Formalizando contratos de transformação e suas propriedades

Então, um contrato de transformação com as técnicas de validação descritas pode ser apresentado formalmente da seguinte forma:



onde $m \models I_M$ significa que todos os invariantes OCL em I_M são válidos no modelo $m \in M$; $C \in classesOf(m)$ sinaliza se o conceito C pertence ao modelo m; $C \sqsubseteq \bot$ simboliza que o conceito C é inconsistente, i.e., está contido no conjunto de conceitos inconsistentes; $(\forall C \in classesOf(m).\neg(C \sqsubseteq \bot))$ determina que todo conceito C pertencente ao modelo m não pode ser inconsistente.

Validação por OCL executa os invariantes para um cenário específico ou instância do metamodelo, mas não permite qualquer raciocínio no nível da linguagem de modelagem. Já o uso de DL permite raciocinar sobre um metamodelo em geral, e não somente sobre um cenário particular.

Apesar de ambas as técnicas permitirem o raciocínio de modelos, elas são complementares, como apontado em [12]. A validação por invariantes OCL pode encontrar erros não detectáveis pelo raciocínio utilizando lógica de descrição.

A verificação de consistência DL trabalha com *semântica de mundo aberto*, o que significa dizer, informalmente, que a ausência de uma informação num modelo não é interpretada como erro. A execução de invariantes OCL, por outro lado, identifica informação ausente num modelo por trabalhar com *semântica de mundo fechado*, como em bancos de dados relacionais.

Assim, a execução dos invariantes deverá acusar um problema que não foi detectado pela verificação de consistência, já que ela considera a falta de informação. Assim, a ordem de execução das técnicas deve ser respeitada.

2.3.3 Contratos de transformação como um padrão de projeto

Com base nos conceitos de contratos de transformação, foi especificado um padrão de projeto para facilitar o seu uso, sendo uma contribuição importante deste trabalho. A Figura 2.2 apresenta o diagrama de classe referente ao padrão, apenas com as classes e suas relações.

A classe *Domain* possui a especificação do metamodelo, o processo de *parsing* (carregar o modelo como uma instância do metamodelo, preservando a relação de boa formação entre o modelo e seu metamodelo) e o *pretty-print* (produzir um artefato com base na instância do metamodelo), i.e., contém a definição da sintaxe abstrata e a sua relação com a sintaxe concreta. Exemplos de processos de *parsing* são: (i) carregar arquivos XMI específicos para uma linguagem; (ii) utilizar mapeamentos de UML para um metamodelo, porém, em algum momento, interpretou um arquivo de entrada de forma semelhante ao item (i). Já para o *pretty-print*, pode-se gerar o código-fonte com base em instâncias de um metamodelo ou, semelhante ao *parsing*, realizar um mapeamento para UML, onde outra ferramenta pode criar o código-fonte.

O processo de *parsing* é auxiliado pelo *IXMIParser*, uma interface para um interpretador de arquivos *XML Metadata Interchange* (XMI, [34]). As informações referente

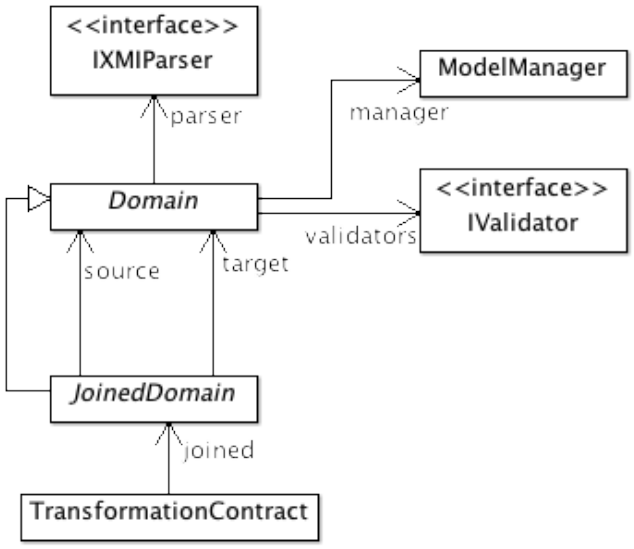


Figura 2.2: Diagrama de classe UML simplificado do padrão de projeto referente a contratos de transformação

aos metamodelos e suas instâncias são armazenadas no *ModelManager*, o que permite a execução de buscas em OCL sobre os dados armazenados.

A classe *JoinedDomain* representa o modelo de transformação e a relações com os metamodelos de entrada e saída são representadas pelas associações com *Domain* chamadas, respectivamente, de *source* e *target*.

IValidator é uma interface que define os validadores envolvidos no processo. As propriedades de um metamodelo, tal como os invariantes em OCL, são vistas, nessa arquitetura, como validadores.

Por fim, *TransformationContract* é responsável pela execução do contrato sobre um *JoinedDomain*. A Figura 2.3 apresenta o diagrama de sequência com detalhes sobre a execução da transformação, que segue os mesmos passos apresentados no Algoritmo 1.

Com isto, o designer da linguagem de modelagem é responsável por implementar extensões das classes *Domain* e *IValidator*, para representar, respectivamente, a linguagem abordada e suas propriedades. O desenvolvedor da transformação de modelo deve selecionar duas classes herdadas de *Domain* e construir uma extensão de *JoinedDomain*, para relacionar os metamodelos envolvidos.

Uma implementação em Java do padrão de projeto descrito, chamada de *TCLib*[5], foi feita para utilizar na implementação dos transformadores deste trabalho. Além da es-

[5] *TCLib* está disponível em `http://lse.ic.uff.br`.

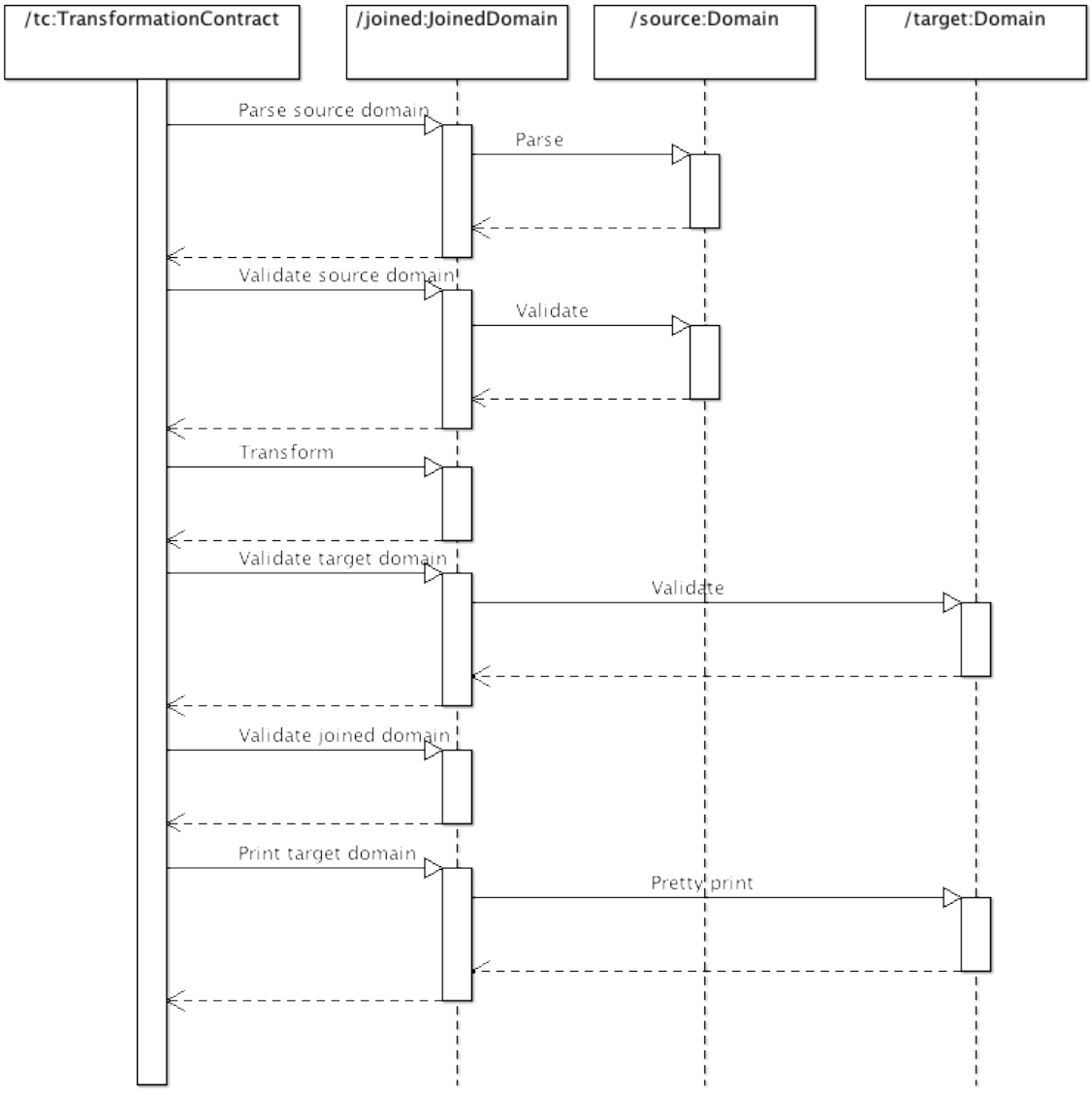


Figura 2.3: Diagrama de seqüência UML do padrão de projeto referente a execução da transformação

trutura já apresentada, ela fornece mecanismos para validação de propriedades em OCL, junto com um mecanismo de rastreabilidade baseado em invariantes para modelos inválidos.

Quando um invariante falha num dos três possíveis passos de validação, como descrito no Algoritmo 1, a arquitetura sinaliza uma exceção, que contém o resultado de uma consulta pelos objetos que não satisfazem a propriedade. A busca realizada verifica a diferença simétrica dos objetos que atendem ao invariante com o conjunto de todos os elementos. Este mecanismo se mostrou bastante efetivo e simplificou a correção dos erros, conforme será abordado nos Capítulos 3 e 4.

Para a implementação do *TCLib*, foi utilizado o *Eye OCL Software* (EOS, [22]), um componente em Java que avalia eficientemente OCL, para armazenar as informações referentes aos metamodelos e suas instâncias e executar expressões OCL sobre os modelos envolvidos. Ele foi utilizada por ser conhecida pelo grupo de pesquisa responsável por este trabalho, conforme apresentado em [23], e pela simplicidade em seu uso, que consiste em informar os modelos e as expressões na forma de *Strings*. Para representar os metamodelos e suas instâncias, o EOS se baseia em uma abstração de diagramas de classes e de objetos, com classes, generalizações, associações e atributos, assim como objetos e links representando as instâncias de classes e associações. Porém o EOS não implementa totalmente a especificação OCL, o que resultou em adaptações nas expressões utilizadas, além de erros para a execução da expressão *let* e a inexistência de mecanismos para a representação de invariantes, pré e pós condições.

Foi utilizada a ferramenta de modelagem *ArgoUML* para a geração do modelo de entrada porque já havia uma implementação para o carregamento de arquivos XMI em objetos Java, utilizada em [23], simplificando o processo de *parsing*. Porém essa ferramenta utiliza uma versão antiga da UML, impedindo o uso de certos modelos pertencentes a versão atual da sua especificação.

Para a verificação de consistência, foi utilizada a ferramenta chamada *Consistency Checker*[6], que utiliza as mesmas especificações apresentadas na Seção 2.3.1. Para realizar a verificação, foi utilizado o raciocinador *Pellet*[7], que representa uma base de conhecimento em DL com *OWL*[8].

[6] *Consistency Checker*, desenvolvido por Cássio Santos, membro do λSE, dentro do qual este trabalho foi desenvolvido, está disponível em `http://lse.ic.uff.br`.

[7] *Pellet* é um raciocinar de ontologias escritas em OWL, disponível em `http://clarkparsia.com/pellet/`.

[8] *OWL*, ou *Web Ontology Language*, é uma linguagem para definição de ontologias mantida pela W3. A sua especificação está disponível em `http://www.w3.org/TR/owl-features/`.

O *Consistency Checker* funciona como uma *caixa preta*, não necessitando entender de lógica de descrição nem como é calculada a consistência de um modelo. Para realizar a verificação, é necessário informar os elementos do metamodelo e carregar o modelo utilizando uma estrutura própria, descrita na documentação do verificador. Assim, tal verificador foi utilizado em todos os modelos presentes nos transformadores apresentados nos Capítulos 3 e 4.

2.4 Considerações finais

Desenvolvimento dirigido a modelo é uma abordagem cujos modelos de um sistema são elementos importantes no processo, aumentando o nível de abstração do processo de desenvolvimento e simplificando a criação de softwares. A formalização de DDM fornece novos mecanismos para garantir que a transformação está funcionando corretamente, além de aumentar a confiança sobre o artefato final gerado.

Contratos de transformação é uma abordagem para a especificação formal de transformações de modelo, com enfoque na validação e verificação do processo e dos modelos envolvidos. Aplicando técnicas de validação já existentes, é possível garantir que a transformação está correta, uma vez que a própria transformação é interpretada como um modelo. Além disto, esta abordagem também foi modelada como um padrão de projeto, sendo aplicado com sucesso nos transformadores a serem apresentados na Seção 3 e Seção 4.

Apesar das técnicas de validação apresentadas serem suficientes para os transformadores desenvolvidos até o momento, os modelos abordados apenas representam aspectos estáticos de uma aplicação. Apesar de, conceitualmente, não existir uma limitação que impeça a especificação de propriedades dinâmicas num contrato de transformação, novos estudos sobre validadores e verificadores para aspectos dinâmicos, como o uso de lógica temporal, são necessários para atestar a aplicabilidade de contratos de transformação neste contexto.

Novas comparações de CT com outras abordagens, e.g., ATL, são importantes para destacar os benefícios obtidos pela formalização e uso de validadores.

Capítulo 3

Uma abordagem baseada em contratos de transformação para sistemas baseados em componentes

Este capítulo apresenta uma transformação de modelos para sistemas distribuídos utilizando contratos de transformação, conforme publicado em [13]. Esta transformação considera diagramas de classe UML descritos em arquivos XMI como entrada de dados e gera código-fonte em Java com *Enterprise JavaBeans* (EJB). Este transformador foi nomeado de `UMLtoEJB`, cuja intenção é ilustrar a aplicabilidade dos contratos de transformação num contexto complexo.

`UMLtoEJB` é baseada na transformação de UML para EJB apresentada em [37]. As definições dos metamodelos envolvidos (UML e EJB) foram estendidas para atender herança de classes e suporte a diferentes tipos de dados. Os invariantes dos metamodelos foram baseados nas especificações referentes às linguagens envolvidas. A escolha de [37] foi feita por se tratar de um trabalho clássico de DDM, sendo bastante referenciado, e por apresentar uma transformação de modelos completa, descrevendo todos os passos necessários para o processo de transformação. As consequências desta escolha são: (i) o metamodelo UML utilizado não afeta aspectos comportamentais, ou seja, ele representa apenas um subconjunto da especificação da UML (No entanto, a exemplificação da técnica num contexto não-trivial é válida.); (ii) o metamodelo EJB aborda apenas uma parte da sua especificação, além de se tratar de uma versão antiga. Mais detalhes sobre os metamodelos utilizados e seus invariantes serão apresentados na Seção 3.1.

Devido ao uso de contratos de transformação, é necessário definir o modelo de transformação. O modelo de transformação é o resultado da união disjunta dos metamodelos

UML e EJB, além das metaclasses representando as relações entre os metamodelos. Os seus invariantes garantem a instanciação correta das suas relações. No caso de `UMLtoEJB`, o conjunto de invariantes é dado pelo conjunto de implicações entre as pré e pós condições de cada regra de transformação descrita em [37], unido a outros invariantes necessários para a garantia das relações entre os metamodelos UML e EJB. A Seção 3.2 discute o contrato de transformação de `UMLtoEJB`.

Considerando o padrão de projeto para desenvolvimento de transformadores de modelo utilizando contratos de transformação apresentado na Seção 2.3.3, o transformador `UMLtoEJB` implementa três domínios: UML, EJB e UMLEJB. Somente o domínio UML implementa um *parser* XMI, que processa modelos construídos na ferramenta ArgoUML. O EOS é utilizado tanto como gerenciador de modelos quanto interpretador OCL de todos os domínios. Além da verificação de invariantes em OCL, cada domínio (UML, EJB e UMLEJB) também possui um verificador de consistência em lógica de descrição, conforme apresentado na Seção 2.3.3.

A transformação, implementada como um método na classe *UMLEJBDomain*, que estende a classe *JoinedDomain*, é definida por indução estrutural sobre o metamodelo UML. Essencialmente, para cada elemento do metamodelo de origem existe um método que realiza "*queries*" no gerenciador de modelos, buscando por suas instâncias a serem transformadas e inserindo novos objetos no gerenciador, que são instâncias de elementos do metamodelo de destino relacionadas aos elemento de metamodelo de origem pelo modelo de transformação. Além dos novos objetos relativos ao metamodelo de destino, links e objetos que instanciam as relações entre as linguagens de modelagem relacionadas pela transformação são também inseridos (É importante notar que as relações induzidas pelo transformador entre as linguagens de modelagem são representadas no modelo de transformação por classes, representando as transformações, e associações entre as classes, representando relações entre os elementos de modelo dos metamodelos das linguagens de modelagem.). A Seção 3.3 detalha a implementação do transformador.

Para testar e analisar o transformador, foram utilizados três diferentes exemplos: (i) a arquitetura de um *blog*; (ii) um sistema de agendamento de reuniões; e (iii) um sistema para uma empresa de cestas de café da manhã, apresentado em [37]. Esses exemplos exercitam todos os elementos de modelo definidos no metamodelo UML implementado em `UMLtoEJB`. A Seção 3.3.1 detalha esses exemplos e a Seção 3.3.2 discute os resultados desse experimento.

Por fim, na Seção 3.4 são apresentadas as considerações finais sobre o `UMLtoEJB`, i.e.,

uma reflexão sobre os resultados obtidos e o uso de contratos de transformação.

3.1 Domínios envolvidos

Neste trabalho, uma linguagem de modelagem tem sua sintaxe representada por um metamodelo e modelos descritos nesta linguagem de modelagem (que representam diagramas na linguagem de modelagem) são instâncias deste metamodelo. Numa transformação de modelos, que implementa o padrão de projeto descrito na Seção 2.3.3, uma linguagem de modelagem é definida através de um domínio que possui um metamodelo, um método *parse*, que cria uma instância do metamodelo a partir de uma descrição XMI, e um método *pretty-print*, que cria uma representação numa sintaxe concreta com base numa instância do metamodelo.

Note que o método de *parse* implementa as regras de *boa-formação* (Ver Capítulo 2) associadas à linguagem de modelagem. A relação de conformidade é implementada pelos validadores de um domínio. Adiante serão abordados os domínios UML e EJB, que são, respectivamente, os domínios de entrada e de saída do transformador `UMLtoEJB`.

3.1.1 Domínio UML

Unified Modeling Language (UML) é uma linguagem de modelagem de propósito geral da engenharia de software orientada a objetos. Ela consiste na especificação de diversos diagramas para determinar as características estáticas e dinâmicas de um sistema. Segundo [33, p. 4], em termos de visões de um modelo, a UML define os seguintes diagramas gráficos:

- Diagrama de caso de uso: Exibe as relações entre os casos de uso, dentro de um sistema ou outras entidades semânticas, e seus atores.
- Diagrama de classe: Exibe a estrutura estática do sistema, apresentando seus elementos, tais como classes e tipos, suas estruturas internas e as relações entre si.
- *Diagramas de comportamento:*
  - Diagrama de estado: Descreve o comportamento das instâncias de um elemento do modelo, como um objeto ou uma interação. Ele também pode ser visto como um grafo que representa uma máquina de estado.

- Diagrama de atividade: Descreve o comportamento interno de processamento, que significa o conjunto de ações possíveis geradas internamente por um elemento do modelo.
- *Diagramas de interação:*
  - Diagrama de sequência: Exibe a sequência explícita de comunicação entre atores e objetos em um cenário.
  - Diagrama de colaboração: Exibe as relações e comunicações de atores e objetos em um cenário.
- *Diagramas de implementação:*
  - Diagrama de componente: Descreve a estrutura de componentes, incluindo quem os especificam e implementam.
  - Diagrama de implantação: Descreve a estrutura no qual os componentes serão implantados.

Em [37] é apresentado um metamodelo para diagramas de classe UML simplificado, definidos por classes, classes associativas, interfaces, atributos, operações e as relações entre seus elementos. A Figura 3.1 apresenta uma extensão desse metamodelo UML (A não ser quando dito o contrário, quando utilizado o termo "metamodelo UML" refere-se ao metamodelo para modelos representando diagramas de classe UML.).

Todo elemento de um modelo que representa um diagrama de classes UML, i.e., instância do metamodelo de UML, é uma instância da metaclasse *ModelElement*, que precisa de um nome, representado pelo atributo *name*. Essencialmente, todo elemento de um modelo é uma instância de *Classifier*, *Typed*, *Association* ou *EnumerationLiteral*. A metaclasse abstrata *Classifier* é uma generalização de *Class*, *Interface* e *DataType*. *Typed*, como *Classifier*, é uma metaclasse abstrata e uma generalização de *Feature* e *Parameter*. Cada *Typed* tem seu tipo definido por um *Classifier*.

A metaclasse *Interface* representa uma interface em UML. *AssociationClass* representa uma classe associativa, que é tanto uma classe quanto uma associação. Instâncias de *Class* podem implementar instâncias de *Interface* e herdar de outras instâncias de *Class*.

Association define como classes são relacionadas. Para representar uma associação, são necessárias duas ou mais pontas de associação (metaclasse *AssociationEnd*), que definem a cardinalidade (pelos atributos *lower* e *upper*), se é uma composição (pelo atributo *composition*) e se é navegável (pelo atributo *navigable*).

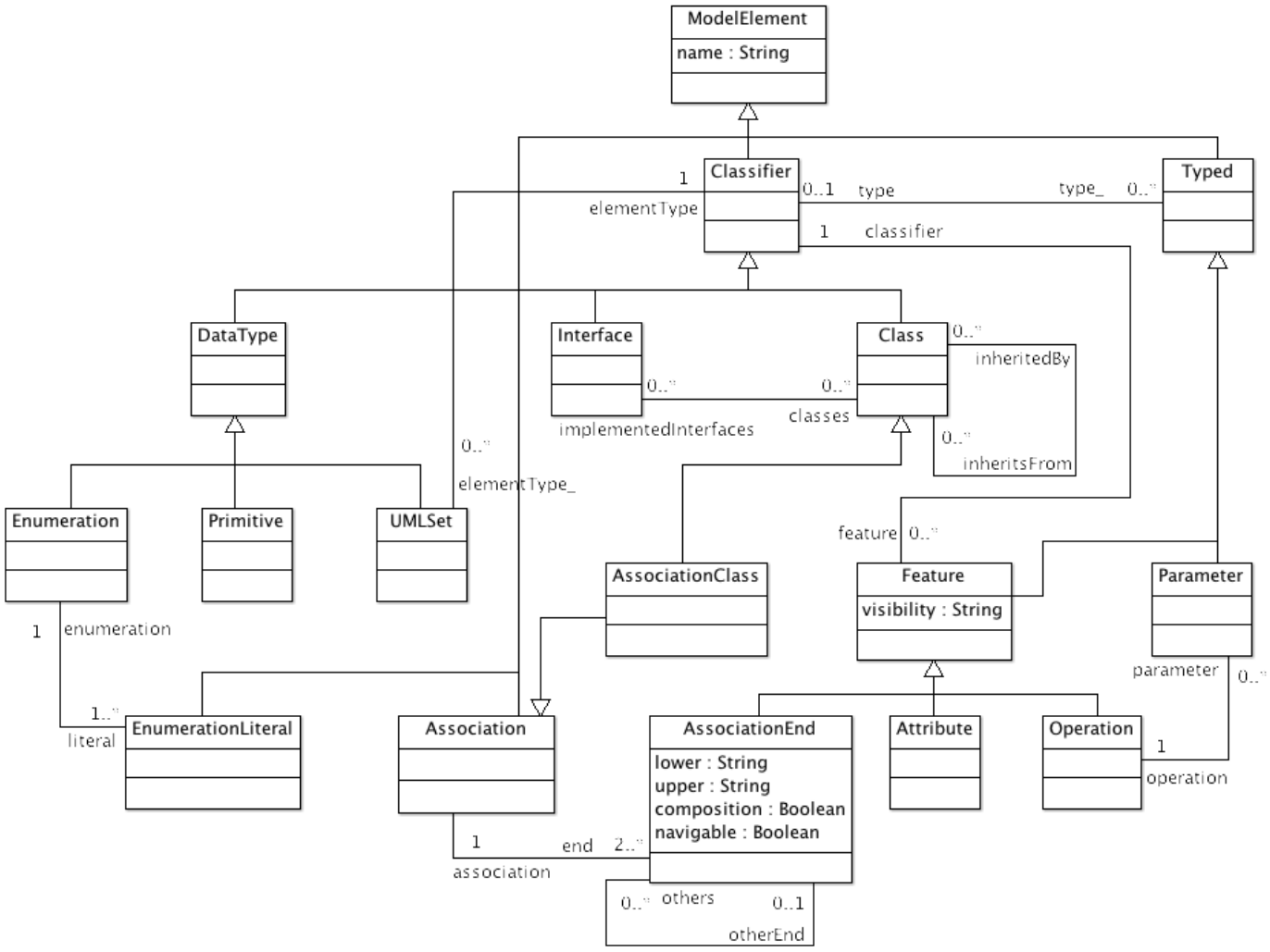


Figura 3.1: Metamodelo para digramas de classe UML utilizado neste trabalho

Feature simboliza as características de um elemento. Toda instância de *Feature* pode ser um *AssociationEnd*, *Attribute* ou *Operation*. As metaclasses *Attribute* e *Operation* representam, respectivamente, atributos e métodos de uma instância de *Classifier*. Uma instância de *Operation* pode ter nenhuma ou várias instâncias de *Parameter*, que representam os parâmetros de um dado método.

Instâncias de *DataType* são responsáveis por representar os tipos de dados que podem ser usados nos modelos. Os dados são divididos em três grupos: (i) *Primitive*; (ii) *UMLSet*; (iii) *Enumeration*. *Primitive* simboliza um tipo primitivo do modelo. *UMLSet* configura um conjunto de elementos cujo tipo é definido pela relação com *Classifier*. *Enumeration* desempenha uma enumeração de literais, que são representados por instâncias de *EnumerationLiteral*.

Esse metamodelo UML possui as seguintes diferenças, em relação ao metamodelo apresentado em [37]:

- Capacidade para representar herança de classes, através da associação, cujas pontas de associação (*AssociationEnd*) são *inheritedBy* e *inheritsFrom*;
- Tratamento para diversos tipos de dados, representados pela adição das metaclasses *Primitive* (para tratamento específico de tipos primitivos) e *Enumeration* (para tratamento de enumerações, em conjunto com a metaclasse *EnumerationLiteral*);
- Adição do atributo *navigable* em *AssociationEnd*, para configurar associações navegáveis;
- Mudança da relação de *Class* e *Feature* para *Classifier* e *Feature*, com o objetivo de que tipos de dados possam relacionar-se com instâncias de *Feature*;
- Substituição dos tipos dos atributos *visibility*, *lower* e *upper* para *String*, com o intuito de simplificar o tratamento dos seus valores;
- Adição de nomes e cardinalidades para as pontas de associação que não os possuiam;

As propriedades em OCL associadas a esse metamodelo foram baseadas na especificação de diagramas de classe UML [43], conforme apresentadas a seguir[1].

- Classes associativas não podem ter vinculado a ela um *AssociationEnd* que possua o atributo *composition* com valor verdadeiro, já que uma *AssociationClass* também é um *Association*. Em outras palavras, significa que uma classe associativa não pode ter pontas de associação com composição, quando analisada como uma associação. Essa propriedade existe pois neste metamodelo é considerado que toda classe associativa depende das pontas de associação relacionadas a ela, descartando qualquer tratamento especial relacionado as pontas de associação. O invariante referente a está propriedade é:

```
context AssociationClass inv notExistsCompositionAssociationEndAssignedToAssociationClass:
self.end->forAll(ae : AssociationEnd | ae.composition = false)
```

[1] Apesar de existir formas mais simples de especificar os invariantes do metamodelo UML sem mudar seu sentido, eles são executáveis na ferramenta *EOS*, que possui algumas idiossincrasias em relação a interpretação das expressões OCL.

- Nenhuma ponta de associação pode pertencer a uma classe associativa, i.e., nenhum *AssociationEnd* pode pertencer a um *Classifier* do tipo *AssociationClass*. A única forma de relacionar um *AssociationClass* ocorre pela relação *association−end*, herdada da metaclasse *Association*. Com isso, garantimos que não existirá associações com classes associativas, exceto com as classes que as definem.

```
context AssociationEnd inv associationClassCannotBeTypedByAssociationEnd:
self.classifier->forAll(c : Classifier | not c.oclIsKindOf(AssociationClass))
```

- Toda instância de *ModelElement* deve ter o atributo *name* preenchido, com exceção das instâncias das metaclasses *Association* e *AssociationEnd*.

```
context ModelElement inv restrictionRequiredFieldNameToModelElement:
self.name <> '' or self.oclIsKindOf(Association) or self.oclIsKindOf(AssociationEnd)
```

- Não se permite ter ciclos na hierarquia de herança de classes, i.e., uma instância de *Class* não pode se relacionar pela associação representada pelas pontas de associação chamadas *inheritsFrom* e *inheritedBy*, seja diretamente ou pelo caminho induzido pela associação entre várias classes. A associação *inheritsFrom−inheritedBy* modela a hierarquia de herança suportada pelo metamodelo UML utilizado.

```
context Class inv noCyclesinClassHierarchy:
self.inheritsFrom->forAll(c : Class | c.superPlus()->excludes(self))

context Class::superPlus():Set(Class) def:
self.superPlusOnSet(self.emptySet())

context Class::superPlusOnSet(rs:Set(Class)):Set(Class) det:
if self.inheritsFrom->notEmpty() and rs->excludes(self) then
  self.inheritsFrom->collect(c : Class | c.superPlusOnSet(rs->including(self)))->flatten()->asSet()
else
  rs->including(self)
endif

context Class::emptySet():Set(Class) def:
Class.allInstances()->select(dc | false)
```

- Instâncias da metaclasse *AssociationEnd* devem ter valores únicos do atributo *name*, referente às outras instâncias de mesmo tipo, relacionadas com uma mesma instância de *Association*, pela associação chamada *association−end*.

```
context Association inv associationEndsNamesAreUniqueInAnAssociation:
self.end->forAll(ae1, ae2 : AssociationEnd | ae1.name = ae2.name implies ae1 = ae2)
```

- Instâncias da metaclasse *Operation* devem ter as assinaturas de suas operações diferentes de outras instâncias de mesmo tipo, relacionadas com uma mesma instância

de *Classifier*, pela associação chamada *feature−classifier*. A assinatura do método foi definida pelo nome da operação e de seus parâmetros, com os nomes de seus respectivos tipos.

```
context Classifier inv onlyOneOperationNameperClassifier:
self.feature−>select(f : Feature | f.oclIsTypeOf(Operation))−>collect(f : Feature | f.oclAsType(Operation))−>forAll(op1, op2
    | op1.signature() = op2.signature() implies op1 = op2)

context Operation::signature():String def:
if self.parameter−>size() = 0 then
  self.name
else
  self.name.concat(' ').concat(self.signatureParams(self.parameter))
endif

context Operation::signatureParams(params : Set(Parameter)):String def:
if params−>size() = 1 then
  params−>asOrderedSet()−>first().name.concat('_').concat(params−>asOrderedSet()−>first().type.name)
else
  params−>asOrderedSet()−>first().name.concat('_').concat(params−>asOrderedSet()−>first().type.name).concat(' ').
      concat(self.signatureParams(params−>excluding(params−>asOrderedSet()−>first()))) endif
```

- Instâncias das metaclasse *Attribute* e *AssociationEnd* devem ter valores únicos do atributo *name* referente às outras instâncias de mesmos tipos relacionadas com uma mesma instância de *Classifier*, pela associação chamada *feature−classifier*.

```
context Classifier inv onlyOneAttrOrAsscEndNameperClassifier:
self.feature−>select(f : Feature | f.oclIsTypeOf(Attribute) or f.oclIsTypeOf(AssociationEnd))−>forAll(f1, f2 | f1.name = f2.
    name implies f1 = f2)
```

- Apenas associações binárias podem ser composições. De acordo com o metamodelo UML, apenas instâncias de *Association* que se relacionam com duas instâncias de *AssociationEnd*, pelas pontas de associação chamadas de *association* e *end*, podem possuir suas instâncias de *AssociationEnd* com o atributo *composition* verdadeiro.

```
context Association inv limitCompositeAssociationSize:
self.end−>exists(ae : AssociationEnd | ae.composition = true) implies self.end−>size() = 2
```

- A multiplicidade de uma relação de composição não deve ter seu limite superior maior que 1. De acordo com a especificação da UML, toda instância de *AssociationEnd* que possui o atributo *composition* verdadeiro, deve possuir o atributo *upper* igual ao valor *1* ou vazio.

```
context AssociationEnd inv compositeMultiplicityRestriction:
self.composition = true implies self.upper = '1' or self.upper.size() = 0
```

- Instâncias de *DataType* somente se relacionam com instâncias de *Feature* que sejam do tipo *Attribute*.

```
context DataType inv restrictionDataTypeAttribute:
self.feature->forAll(f : Feature | f.oclIsTypeOf(Attribute))
```

Como dito anteriormente, o método *parse* é responsável pela criação de uma instância de um metamodelo de uma linguagem de modelagem, representando um diagrama nesta linguagem. No caso do desse domínio UML, o método *parse* cria, no gerenciador de modelos, uma instância do metamodelo de UML, representando um diagrama de classes.

De acordo com o processo de transformação com contratos de transformação, apresentado no Capítulo 2, ocorre a validação da instância do metamodelo logo após o *parsing*. Esta etapa de validação auxilia na detecção de problemas de conformidade da instância do metamodelo UML. O *parser* é, conceitualmente, responsável pela verificação de boa-formação em relação a sintaxe concreta da linguagem, porém isto não é suportado na implementação atual. As únicas validações realizadas neste transformador após o *parsing* são a validação por invariantes em OCL e a verificação de consistência, ambos referentes à sintaxe abstrata de UML.

O arquivo de entrada, por se tratar de um tipo particular de arquivo XML, pode ser gerado pelo *ArgoUML* ou manualmente, desde que utilize a mesma estrutura usada pela ferramenta de modelagem. Em ambos os casos, podem ocorrer erros durante a especificação do diagrama de classe, resultando em um modelo mal-formado, o que é detectado pelo *parser*.

O erro de conformidade ocorre porque a instância, apesar de correta, não está de acordo com as propriedades do metamodelo. Um exemplo, é o ciclo de herança entre classes, que apesar de correto em relação à sintaxe, viola uma das propriedades apresentadas anteriormente.

O protótipo criado não definiu um método de *pretty-printing* para UML, que, atuando como processo inverso do *parsing*, geraria um arquivo XMI contendo a instância do metamodelo UML ou, ainda, poderia também gerar uma representação em *Eclipse Ecore* de um modelo construído utilizando o *ArgoUML*. No entanto, este método não é necessário no `UMLtoEJB`, uma vez que o domínio UML é utilizado como entrada da transformação. No caso geral, no entanto, para a *reutilização* do domínio UML em outra transformação, eventualmente como domínio de saída, esse método deve ser implementado.

Por isso, o domínio UML apresentado possui as seguintes características: (i) um metamodelo capaz de representar um subconjunto do diagrama de classe UML especificado em [33]; (ii) propriedades para validação da instância desse metamodelo; (iii) processo de *parsing* para carregar um diagrama de classe UML como instância do metamodelo; e (iv) um verificador de consistência em lógica de descrição, utilizando o validador apresentado na Seção 2.3.3.

3.1.2 Domínio EJB

Enterprise JavaBeans (EJB, [53]) é uma arquitetura de componentes *server-side* (i.e., cujas aplicações são executadas no servidor da aplicação) que permite o desenvolvimento de aplicações distribuídas. De acordo com [53], os principais objetivos do EJB são: (i) ser a arquitetura baseada em componentes padrão para construção de aplicações distribuídas orientadas a objeto em Java; (ii) suportar o desenvolvimento, implantação e uso de serviços web; (iii) facilitar o desenvolvimento provendo diversas abstrações que incluem o gerenciamento dos estados dos componentes, assim como uma forma simplificada para o seu acesso remoto e seu processamento em paralelo; (iv) fornecer interoperabilidade entre componentes EJB, assim como entre aplicações que não estão desenvolvidas em Java; (v) ser compatível com o protocolo de comunicação *CORBA*. Este trabalho é baseado na versão 2 do EJB, porque utiliza as mesmas definições apresentadas em [37].

A arquitetura EJB define três tipos de componentes:

- *Entity Bean* (componente de entidade): Responsável pela representação dos dados de um banco de dados, provê um objeto referente a um dado persistente, e.g., um registro de uma tabela. Esse componente é identificável por uma chave primária. Se algum problema ocorrer com o servidor, o componente de entidade, sua chave primária e qualquer referência remota serão válidas, i.e., serão restauradas de forma transparente. Este tipo de componente possui acesso compartilhado por vários usuários, de forma a manter os dados consistentes ao sofrer qualquer alteração. Por fim, ele é mantido ativo conforme a sua existência no banco de dados, sendo destruído no caso dos dados serem apagados.

- *Session Bean* (componente de seção): Responsável pela execução das regras de negócio, realiza operações tais como cálculos e acesso a dados. Esse componente executa o comportamento de um único cliente, que pode atualizar dados representados por componentes de entidade. No caso de algum problema no servidor, esse

tipo de componente não é recuperado, requerendo a criação de novas instâncias para atender aos clientes. Existem dois tipos de componente de seção: (i) *Stateless Session Bean*, um componente de seção que não mantém qualquer estado de conversação (*conversational state*), i.e., não garante a preservação de informações entre as chamadas de método. Neste caso, todas as instâncias deste componente são consideradas idênticas pelo cliente; (ii) *Statefull Session Bean*, um componente de seção que mantém o estado da conversação com um cliente. Este tipo de componente é acessado unicamente por um cliente, que, por sua vez, controla o seu ciclo de vida.

- *Message-Driven Bean* (componente dirigido por mensagem): Responsável pelo processamento assíncrono de tarefas. Ele pode ser invocado assincronamente e executa uma única mensagem de um cliente, podendo atualizar dados representados por componentes de entidade. Ele não possui estado de conversação. Se ocorrer algum problema no servidor, as instâncias desse componente são removidas.

A arquitetura EJB permite criar aplicações *multi-tier*(ou seja, com mais de um camada de servidores, onde cada camada é dada por um ou mais servidores executando os mesmos serviços), que são acessadas por clientes de outras camadas. Um cliente pode ser de três tipos: (i) cliente remoto, que acessa os componentes independentemente de sua localização através de uma chamada de método remota (RMI, sigla em inglês), não sendo obrigatório o uso de Java; (ii) cliente local, que está localizado na mesma máquina virtual Java (JVM, sigla em inglês) com os componentes EJB que provêem acesso local; (iii) cliente de serviço web, que acessa um *stateless session bean* através de serviços WEB descritos em documentos na linguagem *WEB Service Description Language* (WSDL), um formato de XML para descrever um serviço web.

Além disto, em EJB existem duas formas de tratar persistência de dados: (i) *contained-managed persistence* (CMP), onde a persistência é controlada pelo *container* EJB; (ii) *bean-managed persistence* (BMP), na qual a própria aplicação controla a persistência dos dados.

Neste trabalho é utilizado apenas o componente de entidade, devido ao foco em representar aspectos estruturais, o que limita a definição dos outros componentes, e porque [37] detalha apenas a especificação desse componente. Para a sua definição, três interfaces em EJB são importantes: (i) *Remote Interface*; (ii) *Home Interface*; (iii) *Entity Bean.*

Remote Interface é uma interface Java que provê os métodos de negócio executáveis pelo cliente, i.e., representa a interface do componente EJB visível ao cliente. Sua im-

plementação pode ser feita de duas formas: (i) declarar uma interface Java que estende *javax.ejb.EJBObject* ou (ii) declarar uma interface Java que estende *javax.ejb.EJBLocalObject*. As instâncias de uma *Remove Interface* são chamadas de objetos EJB.

Ao estender *javax.ejb.EJBObject* considera-se que o componente EJB está em um ambiente distribuído, i.e., em diferentes JVMs, e, consequentemente, os métodos serão invocados remotamente. Os métodos visíveis ao cliente podem disparar exceções do tipo *javax.rmi.RemoteException*, além daquelas declaradas pelos métodos na interface.

Ao estender *javax.ejb.EJBLocalObject*, o componente EJB está na mesma JVM que o cliente, permitindo a chamada local dos métodos. Os métodos visíveis ao cliente que podem disparar exceções do tipo *javax.ejb.EJBException*, além das declaradas pelos métodos.

Por padrão, ambas as interfaces permitem recuperar a chave primária do objeto EJB, verificar se ele é idêntico a outro objeto EJB, destruí-lo e acessar a *Home Interface* que o recuperou. Já a interface *javax.ejb.EJBObject* permite recuperar uma referência para o próprio *Remove Interface* (pela classe *javax.ejb.Handle*), utilizado para obter novamente a referência do objeto EJB, possivelmente em uma JVM diferente.

Home Interface é uma interface Java que provê aos clientes operações para criar, remover e encontrar objetos EJB, assim como métodos que não utilizam dados de uma instância específica. Essa interface é utilizada para que o cliente tenha acesso ao componente de entidade desejado. Sua implementação pode ser feita de duas formas: (i) declarar uma interface Java que estende *javax.ejb.EJBHome* ou (ii) declarar uma interface Java que estende *javax.ejb.EJBLocalHome*.

Interfaces do tipo *javax.ejb.EJBHome* atuam sobre componentes EJB distribuídos e seus métodos serão invocados remotamente podendo disparar exceções do tipo *javax.rmi.RemoteException*, além das declaradas pelo método. Interfaces referentes ao tipo *javax.ejb.EJBLocalHome* atuam sobre componentes EJB locais, i.e., na mesma JVM, permitindo acesso direto aos objetos EJB sem a necessidade de invocações remotas e seus métodos podem disparar exceções do tipo *javax.ejb.EJBException*, além das declaradas pelo método.

Por padrão, ambos os tipos permitem remover um objeto EJB pela sua chave primária. Já a interface *javax.ejb.EJBHome* permite visualizar metadados sobre o componente EJB, recuperar um referência para o próprio *Home Interface* (pela classe *javax.ejb.HomeHandle*) e remover um objeto EJB com uma referência para o mesmo (pela

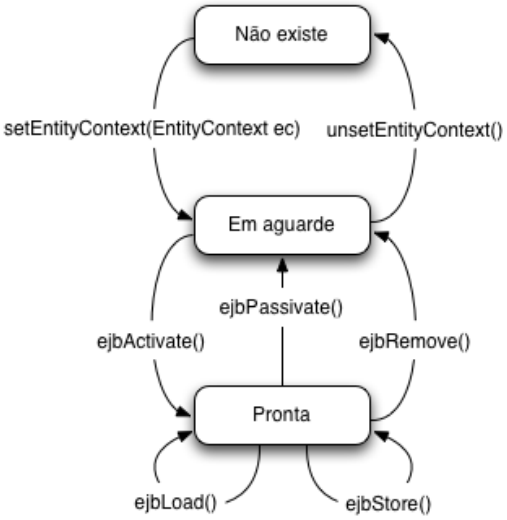


Figura 3.2: Ciclo de vida simplificado de um *Entity Bean*

classe *javax.ejb.Handle*).

Entity Bean é uma interface que representa a implementação de um componente de entidade. Ela define o comportamento do componente, especificado pelos métodos existentes na *Remote Interface* e *Home Interface* relacionadas, e o seu ciclo de vida. O ciclo de vida (ver Figura 3.2) de uma instância *Entity Bean* é definido pelos seguintes estados: (i) a instância ainda não existe; (ii) *em aguarde* (*pooled state*, em inglês), no qual a instância está em um *pool* de instâncias de um *Entity Bean* sem uma particular identidade, i.e., sem um registro de uma tabela; (iii) *pronta* (*ready*, em inglês), a instância está pronta para uso com uma particular identidade associada.

A mudança dos estados ocorrem automaticamente pelos seguintes métodos, exemplificados pela Figura 3.2, que é uma simplificação do ciclo de vida existente em [53]: (i) *ejbActivate()*, chamado quando não existem instâncias disponíveis (no estado *pronta*) para atender uma requisição do cliente, modifica o estado da instância de *em aguarde* para *pronta* e a atribui uma identidade; (ii) *ejbLoad()*, necessário quando o container deseja sincronizar a identidade da instância com um registro do banco da dados, carregando os dados persistentes; (iii) *ejbPassivate()*, invocado quando uma instância é desassociada de uma identidade, mudando o estado de *pronta* para *em aguarde*; (iv) *ejbRemove()*, executado para remover a identidade de uma instância, muda o estado de *pronta* para *em aguarde*; (v) *ejbStore()*, chamado quando o *container* EJB deseja sincronizar o banco de dados com a identidade da instância, persistindo os dados do objeto; (vi) *setEntityContext(EntityContext ctx)*, invocado ao criar uma nova instância (que estará no estado *em aguarde* ao termino da instanciação), define o contexto do componente, informando os

serviços disponíveis pelo *container* EJB e informações sobre o cliente que o invocou; (vii) *unsetEntityContext()*, executado para remover uma instância no estado *em aguarde*. Estes métodos devem ser implementados pelo desenvolvedor do componente caso seja utilizado a persistência de dados BMP. Outros métodos podem mudar o estado de um *Entity Bean* mas não são obrigados de serem implementados. Em [53, p. 269] o ciclo de vida completo de *Entity Bean* e a transição de estados são apresentados com mais detalhes.

Por fim, segundo [55, p. 856], deve-se utilizar EJB quando uma aplicação tem um dos seguintes requisitos: (i) a aplicação precisa ser escalável; (ii) é necessário garantir a integridade dos dados compartilhados; (iii) a aplicação tem vários clientes, necessitando de mecanismos para localizar clientes remotos.

A especificação sobre EJB [53] é mantida pela *Java Community Process* (JCP), responsável por especificações referentes a linguagem Java, entre elas a *Java Platform, Enterprise Edition* (J2EE, [54]). J2EE é uma especificação que agrupa várias especificações, inclusive EJB, para o desenvolvimento de softwares empresariais distribuídos em Java.

Em [37] é apresentado um metamodelo EJB capaz de representar componentes de entidade e de seção, utilizando componentes de granularidade grossa. A descrição do modelo de componentes em [37] é muito sucinta, não apresentando detalhes importantes sobre o tratamento da granularidade, representadas pelas metaclasses *EJBEntityComponent* e *EJBDataClass*. Para tal, descobriu-se qual padrão de projeto foi utilizada em [37], com base nas explicações fornecidas, que se chama *Composite Entity* [2]. Este padrão se propõe a modelar, representar e gerenciar um conjunto de objetos persistentes interrelacionados, em vez de representa-los individualmente como componentes de entidade com baixa granularidade.

O padrão de projeto *Composite Entity* surgiu para combater os problemas de desempenho associados à comunicação com *Entity Beans* por meio da interface remota. Para implementa-lo, é sugerido um novo tipo de entidade chamado *objeto dependente*, que terá seu ciclo de vida gerenciado por um *Entity Bean*. Assim, um *Entity Bean* fornece todas as informações referentes aos seus objetos dependentes, que são transferidos somente uma única vez pela rede.

Na Figura 3.3 é apresentado o metamodelo EJB utilizado nesse trabalho, que consiste em uma extensão do metamodelo EJB presente em [37, p. 115], cujas diferenças são apresentadas na Tabela 3.1.

Todo elemento do metamodelo EJB é uma instância de *EJBClassifier* ou *EJBTyped*.

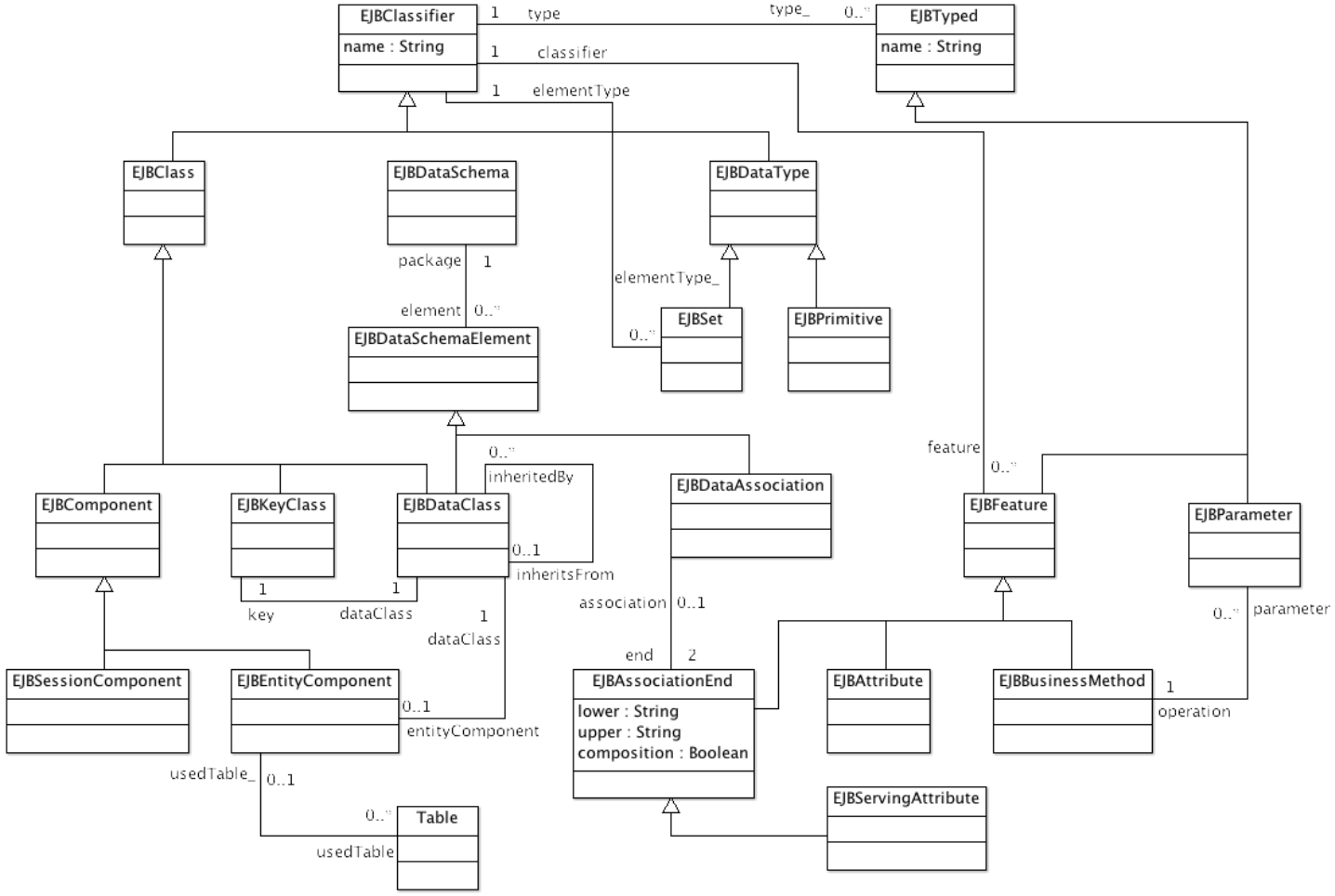


Figura 3.3: Metamodelo EJB utilizado

A metaclasse *EJBClassifier* representa elementos classificadores, que são *EJBClass* ou *EJBDataType*. A metaclasse *EJBTyped* representa elementos tipáveis, que são *EJBFeature* ou *EJBParameter*.

A metaclasse *EJBDataType* simboliza os tipos de dados suportados pelo metamodelo, que são conjuntos de dados que possuem um tipo (metaclasse *EJBSet*) ou dados primitivos (metaclasse *EJBPrimitive*). Instâncias dessas metaclasses devem possuir um nome. Os tipos de dados são representados da mesma maneira que no metamodelo UML, com exceção da ausência de enumeradores.

A metaclasse *EJBClass* representa os tipos de classes existentes em EJB, a saber: (i) *EJBComponent*, que representa os componentes suportados; (ii) *EJBKeyClass*, para representar chaves primárias de componentes de entidade; (iii) *EJBDataClass*, que representa os dados persistentes da aplicação e serão considerados como os objetos dependentes pelo padrão de projeto *Composite Entity*. Toda instância de *EJBDataClass* possui uma chave primária (metaclasse *EJBKeyClass*), igual ao registro da tabela que representa.

Por fim, *EJBDataClass* pode pertencer a uma hierarquia de herança, representada pela associação cujas pontas de associação são *inheritedBy* e *inheritsFrom*.

A metaclasse *EJBComponent* é uma generalização dos componentes EJB de entidade (metaclasse *EJBEntityComponent*) e de seção (metaclasse *EJBSessionComponent*). A metaclasse *EJBEntityComponent* pode se comunicar com várias tabelas do banco de dados (metaclasse *Table*). Componentes de entidade estão interligados a uma instância de *EJBDataClass*, que se relaciona com outras instâncias de *EJBDataClass* consideradas como *objetos dependentes*, definindo então um componente com granularidade grossa. A chave primária necessária para identificar um componente de entidade é, na verdade, uma instância de *EJBKeyClass* associada ao *EJBDataClass* relacionado.

A metaclasse *EJBFeature*, de forma idêntica à metaclasse *Feature* do metamodelo UML, representa as características de uma instância de *EJBClassifier*, a saber: (i) atributos (metaclasse *EJBAttribute*); (ii) métodos referentes às regras de negócio (metaclasse *EJBBusinessMethod*), que pode possuir parâmetros (metaclasse *EJBParameter*); (iii) pontas de associação (metaclasse *EJBAssociationEnd*), que possui o limite inferior (atributo *lower*), limite superior (atributo *upper*) e se é uma composição (atribute *composition*).

As instâncias de *EJBAssociationEnd* serão relacionadas a uma *EJBDataAssociation* somente se as pontas pertencerem a instâncias de *EJBDataClass*. A metaclasse *EJBServingAttribute* é um tipo especifico de ponta de associação utilizado quando se relaciona um *EJBEntityComponent* com um *EJBDataClass*, representando a classe de dados de um componente de entidade.

A metaclasse *EJBDataSchema* descreve um conjunto de instâncias de *EJBDataClass* inter-relacionadas, através de instâncias de *EJBDataAssociation*, que serão usadas seguindo o padrão de projeto *Composite Entity*. Este conjunto é constituído por instâncias de *EJBDataSchemaElement*. Com essas informações, um componente de entidade conhece quais objetos de dados devem ser transferidos juntamente com o seu objeto persistente.

As propriedades a seguir foram formuladas seguindo a especificação [53][2].

- *EJBServingAttribute* é um tipo particular de ponta de associação para representar qual instância de *EJBDataClass* pertence a um *EJBEntityComponent*. Para preservar tal propriedade, foram criadas duas restrições para garantir que: (i) toda

[2]Apesar de existir versões mais simples de dos invariantes do metamodelo EJB, eles são executáveis na ferramenta *EOS*.

Mudança no metamodelo	Motivo
Criação da associação entre *EJBEntityComponent* e *EJBDataClass*	Explicitar a relação entre componentes de entidade e os seus objetos de dados persistente
Criação da associação entre *EJBKeyClass* e *EJBDataClass*	Explicitar a relação entre os objetos de dados persistente e suas chaves primárias
Criação da associação cujas pontas de associação são *inheritedBy* e *inheritsFrom*	Representar herança de classes de dados (metaclasse *EJBDataClass*)
Adição das metaclasses *EJBPrimitive* e *EJBSet*	Tratar, respectivamente, tipos primitivos e conjunto de dados
Mudança da relação de *EJBClass* e *EJBFeature* para *EJBClassifier* e *EJBFeature*	Permitir que tipos de dados possam se relacionar com instâncias de *EJBFeature*;
Substituição dos tipos de atributos *lower* e *upper* para *String*	Simplificar o tratamento dos seus valores
Desconsiderar a criação da classe *Manager* no método *pretty-print*	Não tem uma semântica clara em [37] ou, até onde foi capaz de pesquisar [53, 55, 52, 46, 39, 2], em nenhuma outra documentação sobre EJB.

Tabela 3.1: Diferenças entre o metamodelo EJB em [37] e o utilizado neste trabalho

instância de *EJBServingAttribute* pertence a uma instância de *EJBEntityComponent*; (ii) toda instância de *EJBServingAttribute* é do tipo *EJBDataClass*.

```
context EJBServingAttribute inv everyEJBServingAttributebelongstoEJBEntityComponent:
self.classifier->forAll(c : EJBClassifier | c.oclIsKindOf(EJBEntityComponent))

context EJBServingAttribute inv everyEJBServingAttributehasEJBDataClass:
self.type->forAll(c : EJBClassifier | c.oclIsKindOf(EJBDataClass))
```

- Toda instância de *EJBClassifier* deve possuir o atributo *name* preenchido.

```
context EJBClassifier inv restrictionRequiredFieldNameFromClassifier:
self.name <> ''
```

- Toda instância de *EJBTyped* deve ter o atributo *name* preenchido, com exceção de instâncias da metaclasse *EJBAssociationEnd*.

```
context EJBTyped inv restrictionRequiredFieldNameFromTyped:
self.name <> '' or self.oclIsKindOf(EJBAssociationEnd)
```

- Toda instância de *EJBBusinessMethod*, cujo valor do atributo *name* comece com *remove*, deve ser um *Remove Method*. *Remove Method* é um tipo de método em EJB, que tem como objetivos atender aos seguintes critérios: (i) o valor do atributo

name precisa começar com *remove*; (ii) o retorno do método necessita ser do tipo *void*, i.e., não ter retorno; (iii) tem que possuir um ou mais parâmetros.

```
context EJBBusinessMethod inv prefixRemovemustbeaRemoveMethod:
self.name.substring(1, 6) = 'remove' implies self.isRemoveMethod()

context EJBBusinessMethod::isRemoveMethod():Boolean def:
if self.name.size() >= 6 then
  self.name.substring(1, 6) = 'remove' and self.type.name = 'void' and self.parameter->size() >= 1
else
  false
endif
```

- Toda instância de *EJBBusinessMethod*, cujo valor do atributo *name* comece com *create*, é referenciado como um *Create Method*. *Create Method* é um tipo de método em EJB, que necessita atender aos seguintes critérios: (i) o valor do atributo *name* precisa começar com *create*; (ii) o retorno do método deve ser do tipo *void*, uma instância de *EJBDataClass*, uma instância de *EJBSet* do tipo *EJBDataClass* ou uma instância de *EJBSet* do tipo *void*, representando que não foi definido um tipo.

```
context EJBBusinessMethod inv prefixCreatemustbeaCreateMethod:
self.name.substring(1, 6) = 'create' implies self.isCreateMethod()

context EJBBusinessMethod::isCreateMethod():Boolean def:
if self.name.size() >= 6 then
  self.name.substring(1, 6) = 'create' and
  if self.type.oclIsTypeOf(EJBDataClass) or self.type.name = 'void' then
    true
  else
    (if self.type.oclIsTypeOf(EJBSet) then
      (self.type.oclAsType(EJBSet).elementType.oclIsTypeOf(EJBDataClass) or self.type.oclAsType(EJBSet).elementType.
          name = 'void')
    else
      false
    endif)
  endif
else
  false
endif
```

- Toda instância de *EJBBusinessMethod*, cujo valor do atributo *name* comece com *find*, é conhecida como um *Finder Method*. *Finder Method* é um tipo de método em EJB que precisa atender aos seguintes critérios: (i) o valor do atributo *name* deve ser igual a *findAll* ou começar com *findBy*, *findOne* ou *findMany*; (ii) o retorno do método deve ter como tipo uma instância de *EJBDataClass*, uma instância de *EJB-Set* do tipo *EJBDataClass* ou uma instância de *EJBSet* do tipo *void*, representando que não foi definido um tipo.

```
context EJBBusinessMethod inv prefixFindmustbeaFinderMethod:
self.name.substring(1, 4) = 'find' implies self.isFinderMethod()

context EJBBusinessMethod::isFinderMethod():Boolean def:
self.validateFinderName() and self.validateFinderReturnType()

context EJBBusinessMethod::validateFinderName():Boolean def:
if self.name.size() >= 4 then
  if self.name.substring(1, 4) = 'find' then
    if self.name.size() >= 6 then self.name.substring(1, 6) = 'findBy' else false endif
    or if self.name.size() = 7 then self.name = 'findAll' else false endif
    or if self.name.size() >= 7 then self.name.substring(1, 7) = 'findOne' else false endif
    or if self.name.size() >= 8 then self.name.substring(1, 8) = 'findMany' else false endif
  else false endif
else false endif

context EJBBusinessMethod::validateFinderReturnType():Boolean def:
if self.type.oclIsTypeOf(EJBDataClass) then
  true
else
  (if self.type.oclIsTypeOf(EJBSet) then
    (self.type.oclAsType(EJBSet).elementType.oclIsTypeOf(EJBDataClass)
    or self.type.oclAsType(EJBSet).elementType.name = 'void')
  else
    false
  endif)
endif
```

- Métodos com o prefixo *ejb* possuem tratamento especifico pelo EJB, sendo utilizados para implementar algum método abstrato do próprio framework ou métodos de busca (*Finder Method*), remoção (*Remove Method*) e criação (*Create Method*) do componente de entidade. O invariante a seguir proíbe que sejam declarados *business methods*, isto é, relacionados a funcionalidade de uma aplicação em particular, que tenham o prefixo *ejb*.

```
context EJBBusinessMethod inv prefixEjbisforbidden:
self.substring(1, 3) <> 'ejb'
```

- Instâncias da metaclasse *EJBBusinessMethod* precisam ter as assinaturas de suas operações diferentes de outras instâncias de mesmo tipo relacionadas com uma mesma instância de *EJBClassifier*, pela associação chamada *feature−classifier*. A assinatura do método foi definida pelo nome da operação e de seus parâmetros, com os nomes de seus respectivos tipos.

```
context EJBClassifier inv onlyOneEJBBusinessMethodNameperEJBClassifier:
self.feature−>select(f | f.oclIsTypeOf(EJBBusinessMethod))−>collect(f | f.oclAsType(EJBBusinessMethod))−>forAll(bm1,
      bm2 | bm1.ejbSignature() = bm2.ejbSignature() implies bm1 = bm2))
```

```
context EJBBusinessMethod::ejbSignature():String def:
if self.parameter->size() = 0 then
  self.name
else
  self.name.concat(' ').concat(self.ejbSignatureParams(self.parameter))
endif

context EJBBusinessMethod::ejbSignatureParams(params : Set(EJBParameter)):String def:
if params->size() = 1 then
  params->asOrderedSet()->first().name.concat('_').concat(params->asOrderedSet()->first().type.name)
else
  params->asOrderedSet()->first().name.concat('_').concat(params->asOrderedSet()->first().type.name).concat(' ').
      concat(self.ejbSignatureParams(params->excluding(params->asOrderedSet()->first()))) endif
```

- Instâncias das metaclasse *EJBAttribute* e *EJBAssociationEnd* devem possuir valores únicos do atributo *name*, referente às outras instâncias de mesmos tipos, relacionadas com uma mesma instância de *EJBClassifier*, pela associação chamada *feature−classifier*.

```
context EJBClassifier inv onlyOneAttrOrAsscEndNameperEJBClassifier:
self.feature->select(f | f.oclIsTypeOf(EJBAttribute) or f.oclIsKindOf(EJBAssociationEnd))->forAll(f1, f2 | f1.name = f2.
    name implies f1 = f2)
```

- Instâncias da metaclasse *EJBAssociationEnd* devem ter valores únicos do atributo *name*, referente às outras instâncias de mesmo tipo, relacionadas com uma mesma instância de *EJBDataAssociation*, pela associação chamada *association−end*.

```
context EJBDataAssociation inv EJBAssociationEndsNamesAreUniqueInEJBDataAssociation:
self.end->forAll(ae1, ae2 : EJBAssociationEnd | ae1.name = a2.name implies ae1 = ae2)
```

- Não são permitidos ciclos na hierarquia de herança de classes de dados, i.e., uma instância de *EJBDataClass* não pode pertencer ao fecho transitivo das relações especificadas pelas associações *inheritsFrom* e *inheritedBy*.

```
context EJBDataClass inv noCyclesinEJBDataClassHierarchy:
self.inheritsFrom->forAll(dc : EJBDataClass | dc.superPlusEjb()->excludes(dc))

context EJBDataClass::superPlusEjb():Set(EJBDataClass) def:
self.superPlusEjbOnSet(self.emptyDataClassSet())

context EJBDataClass::superPlusEjbOnSet(rs : Set(EJBDataClass)) def:
if self.inheritsFrom->notEmpty() and rs->excludes(self) then
  self.inheritsFrom.superPlusEjbOnSet(rs->including(self))->asSet()
else
  rs->including(self)
endif

context EJBDataClass::emptyDataClassSet():Set(EJBDataClass) def:
EJBDataClass.allInstances()->select(dc | false)
```

A instância do metamodelo EJB será gerada como resultado de `UMLtoEJB`, a transformação de modelo proposta neste capítulo. No protótipo apresentado, esse domínio não implementa o método de *parsing*. Conceitualmente, para que este domínio pudesse ser reutilizado em outras transformações, seu método de *parsing* deveria ser definido.

A geração de código-fonte foi baseada em [37, p. 64], sendo adaptada para atender as mudanças do metamodelo descritas na Tabela 3.1. O *pretty-printer* do domínio EJB navega pelo grafo resultante do processo de transformação a partir de três tipos de objetos: (i) *EJBEntityComponent*; (ii) *EJBDataClass*; (iii) *EJBKeyClass*.

Cada instância de *EJBEntityComponent* equivale a um componente de entidade EJB, que é composto pela implementação de uma *Remote Interface*, uma *Home Interface* e uma *Entity Bean*.

A geração de código de uma instância de *EJBEntityComponent* cria as seguintes classes:

- Uma *Remote Interface* que estende *javax.ejb.EJBObject*. Ela possui a declaração dos métodos referentes a regras de negócio, que foram definidos como métodos que não são para remoção, criação ou recuperação de objetos EJB, e métodos para leitura (*getter*[3]) e escrita (*setter*[4]) da instância de *EJBDataClass* referente ao componente em questão.

- Uma *Home Interface* que estende *javax.ejb.EJBHome* e possui as seguintes características: (i) declaração de um método chamado *create*, com um parâmetro do tipo da instância de *EJBDataClass*, referente ao *EJBEntityComponent* em questão; (ii) declaração de um método chamado *findByPrimaryKey*, com um parâmetro do tipo da instância de *EJBKeyClass*, referente ao *EJBEntityComponent* em questão; (iii) a declaração de métodos de todas as instâncias de *BusinessMethod* do tipo *Finder Method*, referente ao *EJBEntityComponent* em questão.

- Um *Entity Bean*, implementando os métodos presentes nas *Home Interface* e *Remote Interface* referentes ao objeto, além dos métodos referentes ao ciclo de vida do componente, apresentados na Figura 3.2. Também é gerado a declaração de seu contexto e chave primária (com *getter* e *setter*), além de um construtor padrão, i.e., sem parâmetros.

[3] *Getter* significa uma operação, cujo objetivo, é retornar o valor de um atributo e possui como nome o prefixo *get* junto com o nome do atributo em questão.

[4] *Setter* significa uma operação, cujo objetivo, é atribuir um valor a um atributo e possui como nome o prefixo *set* junto com o nome do atributo em questão.

A declaração do componente EJB, que inclui a relação entre a *Home Interface*, *Remote Interface* e *Entity Bean*, é feita em um arquivo XML de configuração chamado *ejb-jar.xml*. Durante a definição do componente EJB, também é necessário explicitar o tipo de persistência utilizado e o tipo da chave primária (uma instância de *EJBKeyClass*). A geração das configurações não é feita pelo transformador de modelos, que foca apenas na geração de classes em Java.

Na implementação atual do domínio EJB fizemos uma simplificação no seu *pretty-printer*, pelo uso particular deste domínio no transformador `UMLtoEJB`. A metaclasse *EJBSessionComponent* não é tratada, pois o transformador, tal como descrito em [37], não gera *session beans*.

Devido à granularidade grossa dos componentes, consequente da adoção do padrão de projeto *Composite Entity*, a persistência precisa ser controlada pela aplicação, i.e., precisa ser do tipo BMP. Consequentemente, cada instância de *EJBDataClass* dará origem a uma classe Java que representará os dados existentes no banco de dados e implementará manualmente a persistência de acordo com os requisitos da aplicação em particular.

A classe gerada a partir de uma instância de *EJBDataClass* deve atender as seguintes características: (i) implementar a interface *java.io.Serializable*; (ii) extender as classes da sua hierarquia de herança; (iii) possuir a visibilidade dos atributos de acordo com a existência de uma hierarquia de herança. Se existir, a visibilidade deve ser *protected*. No caso de sua inexistência, a visibilidade precisa ser *private*; (iv) possuir um atributo para representar a chave primária dos dados persistidos no banco de dados, que é, na verdade, uma instância de *EJBKeyClass*; (v) possuir um construtor sem parâmetros; (vi) possuir um construtor que tenha um parâmetro, que é uma instância de *EJBKeyClass* para representar a chave primária dos dados; (vii) possuir, para cada instância de *EJBAttribute* relacionada, a declaração de um atributo, um *getter* e um *setter*; (viii) possuir, para cada instância de *EJBAssociationEnd* relacionada, cujo atributo *upper* é igual a 1, a declaração de um atributo, um *getter* e um *setter*; (ix) possuir, para cada instância de *EJBAssociationEnd* relacionada cujo atributo *upper* é diferente de 1, a declaração de um atributo do tipo *java.util.List*, um *getter* para a lista e métodos para adição e remoção de um elemento.

Estas características seguem a especificação presente em [37, p. 69], com exceção dos itens (ii), (iii) e (v). Os itens (ii) e (iii) foram criados porque uma das contribuições feitas em relação ao metamodelo EJB, a representação de herança em instâncias de *EJBDataClass*, requer que a implementação tenha um tratamento específico para as classes (item

(ii)) e a visibilidade de seus atributos (item (iii)). O item (v), referente à criação do construtor vazio, foi introduzido apenas para simplificar a criação de objetos.

Cada instância de *EJBKeyClass* equivale à uma classe Java, representando a chave primária em uma instância de *EJBDataClass* e de *EJBEntityComponent*. Segundo [37], esta classe deveria implementar *javax.ejb.Handle* e ter, para cada atributo da instância de *EJBKeyClass*, um construtor e um *getter* associado, porém estas definições não são suficientes.

Adicionalmente, a classe gerada deve atender as seguintes características: (i) implementar a interface *java.oi.Serializable* [46, p. 184]; (ii) implementar os métodos *public boolean equals(Object obj)* e *public int hashCode()* [53, p. 216]; (iii) sua visibilidade precisa ser pública [53, p. 222]; (iv) possuir um construtor público sem parâmetros [53, p. 222]; (v) todos os seus atributos precisam ser públicos [53, p. 222]; (iv) possuir um construtor com parâmetros para inicializar os seus atributos [52, p. 53].

Com isso, o domínio EJB apresentado possui as seguintes características: (i) um metamodelo capaz de representar um subconjunto dos componentes e recursos oferecidos pelo EJB, conforme especificado em [53]; (ii) propriedades para validação da instância desse metamodelo; (iii) processo de *pretty-printing* para geração de código-fonte em Java utilizando EJB; e (iv) um verificador de consistência em lógica de descrição, utilizando o validador apresentado na Seção 2.3.3. A existência destas características são essenciais para a transformação de modelo proposta neste capítulo.

3.2 Contrato de transformação

Conforme apresentado no Capítulo 2, um contrato de transformação é composto pelo modelo de transformação e suas propriedades. Seguindo essa especificação, foi construído um contrato para a transformação de UML para EJB.

Inicialmente, é apresentado o modelo de transformação dessa transformação, chamado de modelo *UMLEJB*. Nele, as relações entre as linguagens de modelagem UML e EJB, induzidas pelas regras de transformação descritas em [37], dão origem à associações entre os metamodelos UML e EJB.

Em [9, 23] as relações entre as linguagens de modelagem são representadas como associações entre os elementos de modelo dos metamodelos de origem e destino. Nesta dissertação, as relações entre as linguagens de modelagem são representadas por metaclasses

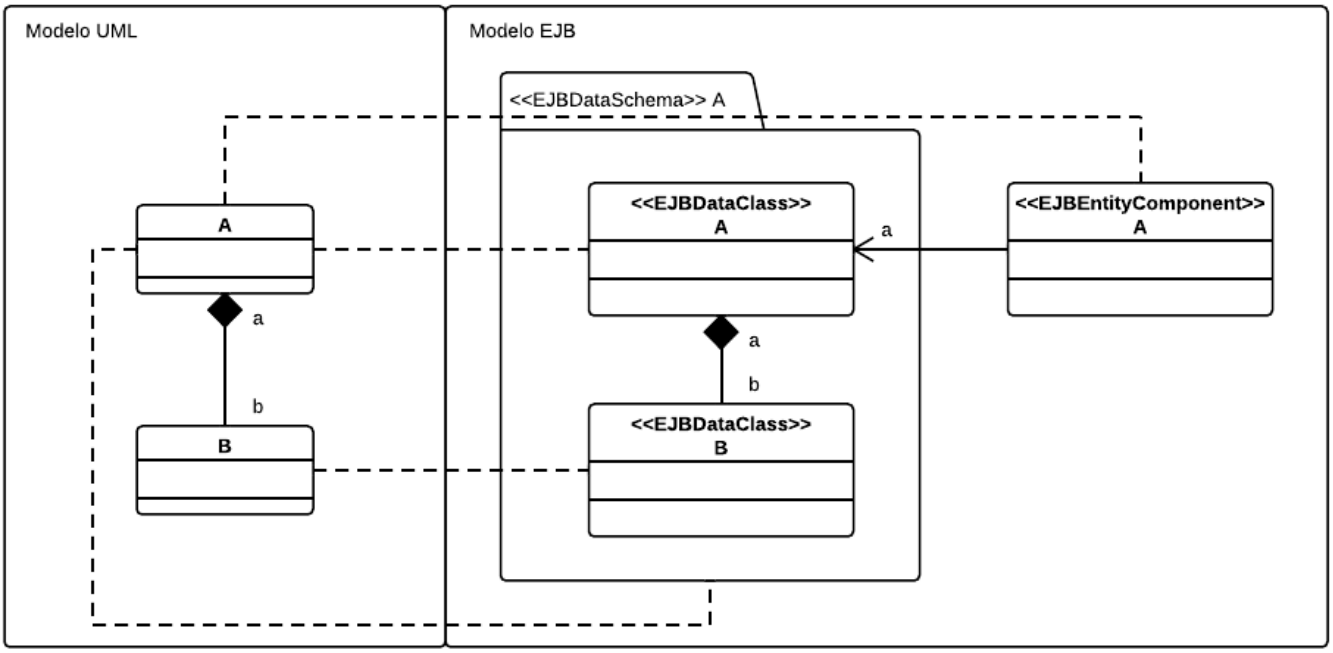


Figura 3.4: Exemplo de transformação de classes UML em componentes de entidades e classes de dados

associadas aos elementos de origem e destino. Desta forma, cada regra de transformação é representada como uma metaclasse, o que permite uma modularização da implementação das transformações. Adicionalmente, esta abordagem facilita a compreensão da relação entre [37] e o contrato de transformação descrito a seguir.

O modelo *UMLEJB* representa a transformação de classes UML em componentes de entidade EJB de acordo com o padrão de projeto *Composite Entity*, juntamente com a estrutura necessária para a sua execução. A Figura 3.4 apresenta um exemplo de transformação, onde um modelo UML (classes com relação de composição) é transformado em um modelo EJB (componentes de entidade e classes de dados) e as relações entre os elementos dos modelos são definidas através de linhas tracejadas.

As Figuras 3.5, 3.6 e 3.7 descrevem o modelo *UMLEJB*, que define as relações entre os metamodelos UML e EJB através de classes representando as regras de transformação, conforme apresentadas a seguir:

- *UMLClassToEJBKeyClass*: Representa a transformação de uma classe UML para uma chave primária no modelo EJB. Para cada instância de *Class*, uma instância de *EJBKeyClass* é criada com um atributo para a verificação de sua unicidade.
- *UMLAssociationClassToEJBKeyClass*: Semelhante à transformação referente à classe *UMLClassToEJBKeyClass*, representa a relação entre uma classe associativa com

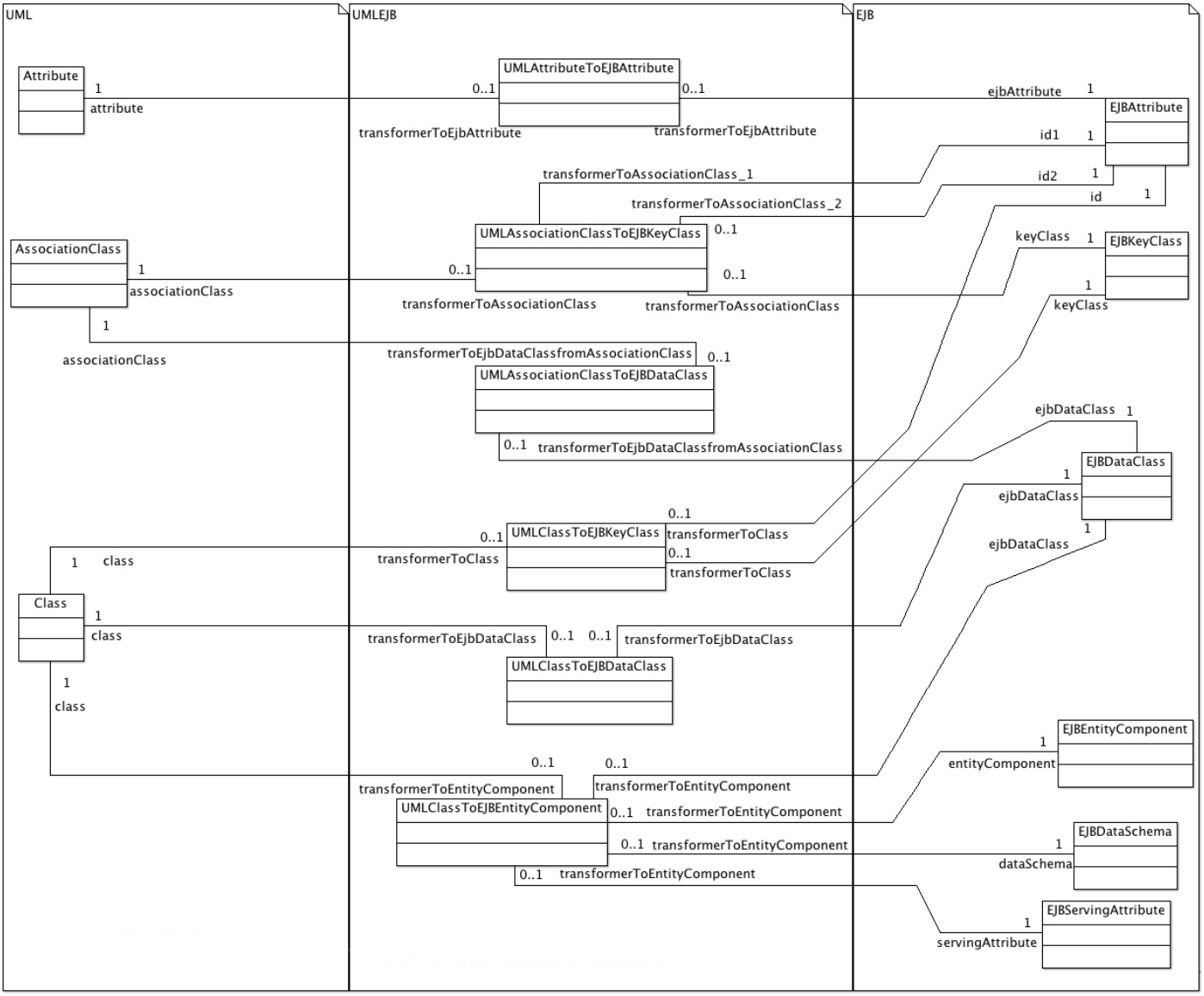


Figura 3.5: Modelo UMLEJB (Parte 1 de 3)

uma chave primária em EJB. Seu objetivo é gerar uma instância de *EJBKeyClass* para cada instância de *AssociationClass* existente no modelo de entrada.

- *UMLClassToEJBDataClass*: Define o mapeamento de uma classe UML, que depende de outra por uma relação de composição, para uma classe de dados EJB (classe *EJBDataClass*).

- *UMLAssociationClassToEJBDataClass*: Análoga à regra de transformação imposta a classe *UMLClassToEJBDataClass*, gera instâncias de *EJBDataClass* a partir de instâncias de *AssociationClass*.

- *UMLClassToEJBEntityComponent*: Simboliza a transformação de uma classe UML em um componente de entidade EJB. Devido ao uso de componentes com granula-

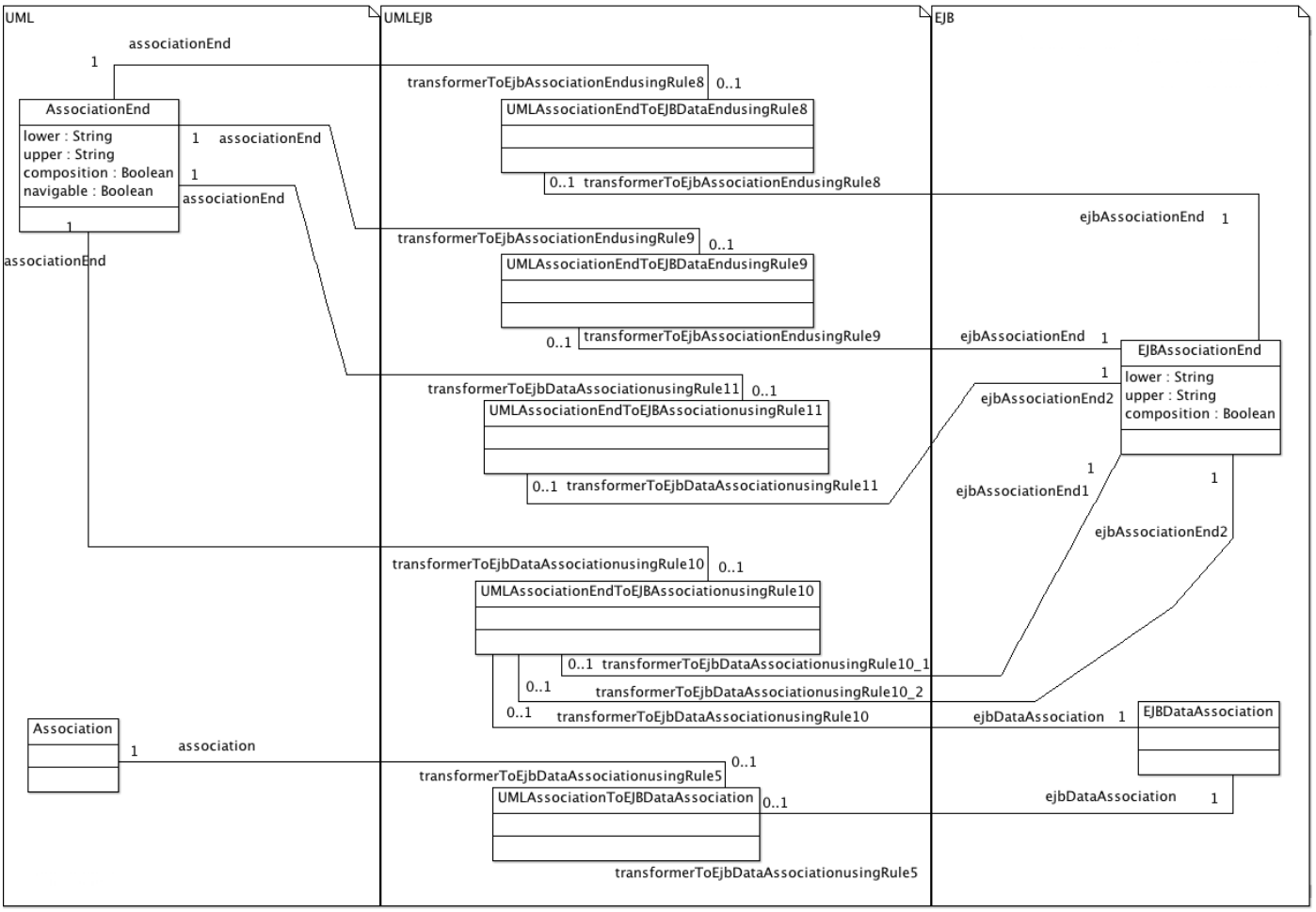


Figura 3.6: Modelo UMLEJB (Parte 2 de 3)

riade grossa, pelo padrão de projeto *Composite Entity*, *somente* instâncias de *Class* que não possuem relação de composição serão transformadas. Em [37] estas classes são chamadas *outermost classes*. Durante a transformação do componente, outras relações também serão criadas: (i) uma classe de dados *EJBDataClass* para agrupar os atributos da classe UML; (ii) um conjunto de elementos para determinar quais classes de dados o componente poderá atuar[5]; (iii) uma instância de *EJBServingAttribute* para determinar, através das características do componente, qual *EJBDataClass* representa os dados do componente.

As operações OCL a seguir calculam quais classes são *outermost*, que serão transformadas em componentes de entidade[6].

[5]O uso do padrão de projeto *Composite Entity* define que os dados de um componente devem ser transferidos em sua totalidade a cada acesso remoto. Para tal, em [37], a existência dos componentes está relacionada a composição das classes UML, já que apenas *outermost classes* serão transformados em instâncias de *EJBEntityComponent*. Logo, um conjunto de instâncias de *EJBDataClass* representam as classes UML que relacionam com a *outermost class* por uma relação de composição.

[6]Apesar de existir formas mais simples de definir as operações OCL relacionadas às classes *outermost*,

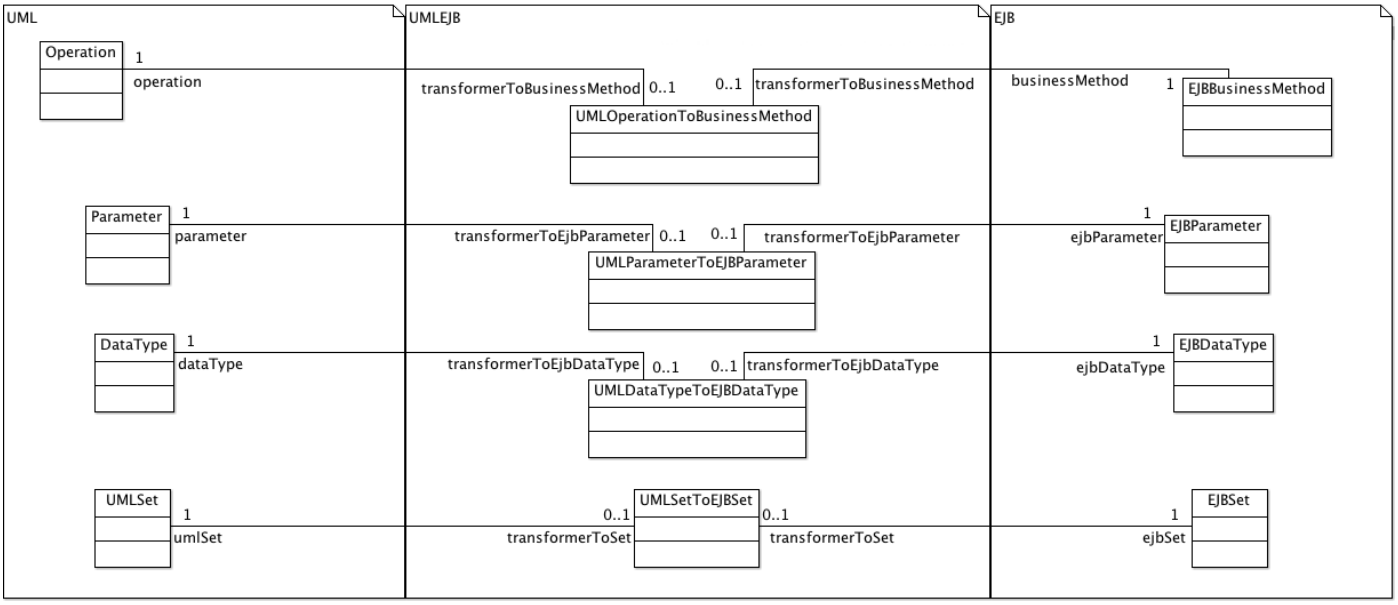


Figura 3.7: Modelo UMLEJB (Parte 3 de 3)

```
context Classifier::getOuterMostContainer():Class body
if self.oclIsTypeOf(Class) then
  self.oclAsType(Class).getOuterMostContainerFromClass()
else
  self.oclAsType(AssociationClass).getOuterMostContainerFromAssociationClass()
endif

context Class::getOuterMostContainerFromClass():Class def:
if self.feature->select(f | f.oclIsKindOf(AssociationEnd))->collect(f | f.oclAsType(AssociationEnd))->exists(ae | ae.
      composition = true) then
  self.feature->select(f | f.oclIsKindOf(AssociationEnd))->collect(f | f.oclAsType(AssociationEnd))->select(ae | ae.
        composition = true).type->select(c | c.oclIsKindOf(Class))->collect(c | c.oclAsType(Class))->asOrderedSet()->
        first().getOuterMostContainer()
else
  self
endif

context Association::getOuterMostContainerFromAssociation():Class def:
if self.oclIsTypeOf(Association) then
  if self.end->select(ae | ae.otherEnd->exists(oe | oe.composition = true))->isEmpty() then
    self.end->asOrderedSet()->first().type->select(c | c.oclIsKindOf(Class))->collect(c | c.oclAsType(Class))->
          asOrderedSet()->first().getOuterMostContainer()
  else
    self.end->select(ae | ae.composition = true)->asOrderedSet()->first().type->select(c | c.oclIsKindOf(Class))->
          collect(c | c.oclAsType(Class))->asOrderedSet()->first().getOuterMostContainer()
  endif
else
  self.oclAsType(AssociationClass).getOuterMostContainerFromAssociationClass()
endif

context AssociationClass::getOuterMostContainedFromAssociationClass():Class def:
```

elas são executáveis na ferramenta *EOS*.

```
if self.end->select(ae | ae.navigable = true)->size() = 1 then
  self.end->select(ae | ae.navigable = true)->asOrderedSet()->first().classifier.oclAsType(Class).getOuterMostContainer()
else
  self
endif
```

A operação *getOuterMostContainer* recupera a lista de classes que fazem parte do fecho transitivo da associação de composição[7].

- *UMLAttributeToEJBAttribute*: Toda instância de *Attribute* deverá ter uma contrapartida em EJB, na forma de uma instância de *EJBAttribute*. O objeto gerado pertencerá a uma instância de *EJBClassifier*, resultante da transformação do elemento classificador da qual o atributo pertence no modelo UML.

- *UMLAssociationEndToEJBDataEndusingRule8*[8]: Dada uma ponta de associação, ela é transformada por essa regra se atender as seguintes condições: (i) pertencer a uma instância de *Association*; (ii) seu tipo e classe a qual pertence estão situados na mesma instância de *EJBDataSchema*, i.e., considerando uma ponta de associação chamada *assocEnd*, a restrição $assocEnd.classifier.getOuterMostContainer() = assocEnd.type.getOuterMostContainer()$ deve ser mantida. O resultado da transformação são instâncias de *EJBAssociationEnd* cujos tipos são instâncias de *EJBDataClass*.

- *UMLAssociationEndToEJBDataEndusingRule9*[8]: Semelhante à transformação representada pela classe *UMLAssociationEndToEJBDataEndusingRule8*, ela transforma quando a classe da qual uma ponta de associação pertence e seu tipo estão em diferentes instâncias de *EJBDataSchema*. Neste caso, as instâncias de *EJBAssociationEnd* criadas possuem instâncias de *EJBKeyClass* como tipo.

- *UMLAssociationEndToEJBAssociationusingRule10*[8]: Representa a transformação de pontas de associação, pertencentes à classes associativas, cuja cardinalidade é maior que *1* e estão situadas na mesma instância de *EJBDataSchema*. Então, é gerado, para cada elemento a ser transformado, uma instância de *EJBDataAssociation* e duas de *EJBAssociationEnd*. Isto é feito para tratar classes associativas, algo inexistente em EJB. Por fim, os tipo das pontas de associação geradas são *EJBDataClass*, por acessarem dados referente ao *EJBDataSchema* em questão.

[7] Em [37, p. 121] possui mais informações sobre a operação *getOuterMostContainer*.

[8] Certas regras de transformação extraídas de [37] possuíam os mesmos nomes, apesar de serem diferentes. Como as relações de UML e EJB foram baseadas em [37], durante a modelagem do modelo de transformação foram listadas as regras de transformação e a posição das regras foram atribuídas às que possuíam nomes repetidos, apenas para diferenciá-las.

- *UMLAssociationEndToEJBAssociationusingRule11*[8]: Similar a transformação representada por *UMLAssociationEndToEJBAssociationusingRule10*, esta classe analisa quando a classe detentora da ponta de associação e seu tipo estão em diferentes instâncias de *EJBDataSchema*. Segundo [37, p. 120, item 11], basta criar uma instância de *EJBAssociationEnd* que referencie uma *EJBKeyClass* fora do conjunto de classes de dados.

- *UMLAssociationToEJBDataAssociation*: Toda associação no modelo UML, que possui composição em alguma de suas pontas, será transformada em uma instância de *EJBDataAssociation*. Classes que possuem relação de composição estarão agrupadas na mesma instância de *EJBDataSchema*. Uma associação em EJB representa a relação entre classes de dados agrupadas para serem utilizadas por um determinado componente. Este componente está associado com a classe de dados da qual as outras classes do agrupamento possuem relação de composição, direta ou indiretamente.

- *UMLOperationToBusinessMethod*: Operações em UML serão transformadas em métodos de negócio em EJB. A relação *operação-classificador* deve ser respeitada no modelo EJB, relacionando o método de negócio com uma instância de *EJBClassifier* adequada.

- *UMLParameterToEJBParameter*: Define o mapeamento de um parâmetro UML para EJB. Neste caso, a sua representação em UML e EJB são idênticas, com exceção de quem a possui, que em um modelo é uma instância de *Operation* e no outro é de *BusinessMethod*. A instância que se relacionará com o parâmetro gerado deve ser equivalente a instância do modelo de origem.

- *UMLDataTypeToEJBDataType*: Representa a transformação dos tipos de dados. Segundo [37], esta transformação deve ser uma série de transformações pré-definidas de baixo nível de UML para EJB. O metamodelo apresentado na Seção 3.1.1 possui mais representações de tipos de dados que [37], que são simbolizadas pelas metaclasses *Primitive* e *Enumeration*. Por isso, precisa-se aprimorar a transformação para também contemplar as novas representações. Instâncias de *Primitive*, que possuem alguma instância de *Feature* relacionada, são transformadas em instâncias de *EJBPrimitive*. Já a transformação das instâncias de *UMLSet* é representada pela classe *UMLSetToEJBSet*.

- *UMLSetToEJBSet*: Toda instância de *UMLSet* deve ser transformada em uma ins-

tância de *EJBSet*, com um tipo de dados equivalente, que no caso é uma instância de *EJBClassifier*. Apesar do metamodelo UML apresentado em [37] possuir as metaclasses referentes a esta regra de transformação, ela não é apresentada, ou seja, não existe qualquer citação ou especificação formal.

As propriedades do modelo de transformação garantem a conformidade de um modelo resultante da transformação e que as propriedades de metamodelo de entrada se preservam na saída. Portanto, as transformações descritas no protótipo devem garantir que estas propriedades são cumpridas. A seguir serão apresentados exemplos de propriedades importantes para a transformação[9].

O primeiro exemplo se refere à transformação de classes UML em componentes de entidade em EJB. Uma restrição da transformação é que apenas *outermost classes* podem ser transformadas. O invariante *everyoutermostClassbecamesEntityComponent* garante que todo elemento transformado atenda à esta condição. Porém, esta propriedade não garante que todos as *outermost classes* foram transformadas. Portanto, foi criado o invariante *verifyUMLClassthatmustbetransformedtoEJBEntityComponent*. Assim, ambos os invariantes garantem que todas as *outermost classes*, e somente elas, foram transformadas em instâncias de *EJBEntityComponent*.

```
context UMLClassToEJBEntityComponent inv everyoutermostClassbecamesEntityComponent:
self.class.isOuterMostContainer()

context UMLClassToEJBEntityComponent inv verifyUMLClassthatmustbetransformedtoEJBEntityComponent:
Class.allInstances()->forAll(c | c.oclIsTypeOf(Class) and c.isOuterMostContainer() implies c.transformerToEntityComponent
    ->notEmpty())
```

O segundo exemplo, complementar ao primeiro, refere-se a transformação de classes UML para instâncias de *EJBDataClass*, sendo que as classes não podem ser *outermost*. Conforme visto no primeiro exemplo, são necessários dois invariantes, que são *UMLClassfromUMLClassToEJBDataClasscannotbeanOutermostclass* e *verifyUMLClassthatmustbetransformedtoEJBDataClass*. Porém nenhum dos invariantes citados garantem que a estrutura presente no modelo UML, pela relação de composição, é preservada no modelo EJB. Portanto, o invariante *verifyClassIntegrityInEJBDataClasses* foi definido, verificando que a relação entre instâncias de *EJBDataClass* preservam a mesma estrutura das classes UML, uma vez que *outermost classes* são transformadas em instâncias de *EJBEntityComponent* com uma *EJBDataClass* específica.

[9] Apesar de existir versões mais simples de especificar as propriedades do modelo de transformação, eles são executáveis na ferramenta *EOS*.

```
context UMLClassToEJBDataClass inv UMLClassfromUMLClassToEJBDataClasscannotbeanOutermostclass:
not self.class.isOuterMostContainer()

context UMLClassToEJBDataClass inv verifyUMLClassthatmustbetransformedtoEJBDataClass:
Class.allInstances()->forAll(c | c.oclIsTypeOf(Class) and not c.isOuterMostContainer() implies c.transformerToEjbDataClass
    ->notEmpty())

context UMLClassToEJBDataClass inv verifyClassIntegrityInEJBDataClasses:
self.class.getOuterMostContainer().transformToEntityComponent.ejbDataClass.feature->select(f | f.oclIsTypeOf(
    EJBAssociationEnd))->collect(f | f.oclAsType(EJBAssociationEnd))->exists(ae | ae.type = self.ejbDataClass)
```

Todos as propriedades, assim como as suas explicações, estão disponíveis no Apêndice A.

Então, com o modelo de transformação e as propriedades, além de um verificador de consistência, é obtido a definição do contrato de transformação utilizado no transformador de UML para EJB.

3.3 Protótipo do transformador

Para auxiliar a implementação do transformador, foi utilizada a arquitetura apresentada no Capítulo 2. A classe *Domain* foi estendida para representar os metamodelos UML (*UMLDomain*) e EJB (*EJBDomain*). No caso do modelo de transformação, foi criada a classe *UMLEJBDomain*, que estende *JoinedDomain* e relaciona *UMLDomain* como domínio de origem e *EJBDomain* como de destino. As propriedades em OCL de cada metamodelo foram representadas por extensões da classe *InvariantValidator*, sendo *UMLInvariantValidator*, *EJBInvariantValidator* e *UMLEJBInvariantValidador* para, respectivamente, os metamodelos UML e EJB e o modelo de transformação. De maneira similar, verificadores de consistência em DL foram definidos para cada domínio, através de extensões da classe *ConsistencyValidator*. Portanto, as classes *InvariantValidator* e *ConsistencyValidator* representam as propriedades desejáveis para este transformador, sendo então extensões necessárias da interface *IValidator* e, consequentemente, do `TCLib`. A Figura 3.8 apresenta um diagrama de classe da implementação do transformador como extensão da arquitetura da Figura 2.2.

Em *UMLDomain*, foi implementado o método de *parsing*, para mapear um arquivo XMI, contendo um diagrama de classe UML modelado na ferramenta *ArgoUML*, numa instância do metamodelo UML adotado neste trabalho. Em *EJBDomain* foi implementado o método de *pretty-printing*, ou seja, a produção de um artefato com base na instância do metamodelo EJB, que, neste caso, é código-fonte em Java usando EJB, conforme

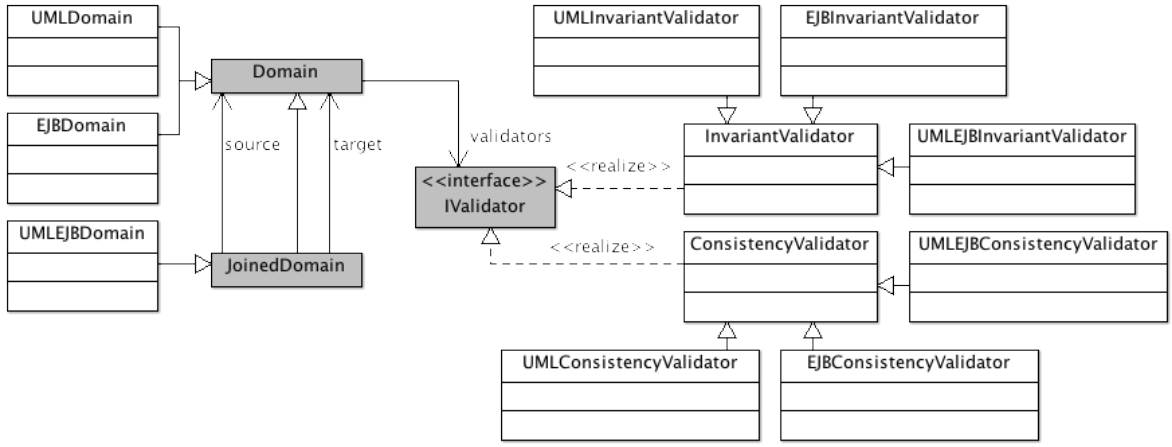


Figura 3.8: Diagrama de classe UML do transformador `UMLtoEJB`, estendendo a arquitetura genérica

apresentado na Seção 3.1.2.

Já a classe *UMLEJBDomain* possui a implementação das regras de transformação, realizadas em Java. A transformação descrita em [37] não se preocupa com a ordem das transformações. É necessário, no entanto, que primeiro os objetos sejam criados para depois serem interconectados.

A abordagem utilizada para tratar o problema de ordenação consiste em separar a transformação em dois passos. No primeiro passo, aplica-se todas as regras de transformação que criam os elementos das instâncias do modelo de transformação e do metamodelo de destino. No próximo passo, criam todas as relações entre os elementos da instância do metamodelo de destino. Assim, todos os elementos já estão criados no momento de relacioná-los.

O protótipo criado possui uma interface gráfica que permite aplicar cada passo do processo de transformação (*parsing*, transformação, verificação e *pretty-printing*) de forma interativa. Adicionalmente, é possível interagir, via OCL, com cada um dos modelos produzidos (origem, de transformação e destino) numa dada aplicação da transformação. A interface gráfica utilizada é instância de uma interface genérica para o auxílio ao ensino e depuração de transformadores de modelos chamada *GenericPad*[10], desenvolvida por Cássio Santos, membro do λSE, grupo de pesquisa em Engenharia de Software orientada a linguagens, dentro do qual este trabalho foi desenvolvido. A Figura 3.9 apresenta uma execução do transformador `UMLtoEJB` exibindo o resultado de uma expressão OCL e uma instância do metamodelo UML.

[10] *Generic Pad* está disponível em `http://www.lse.ic.uff.br`.

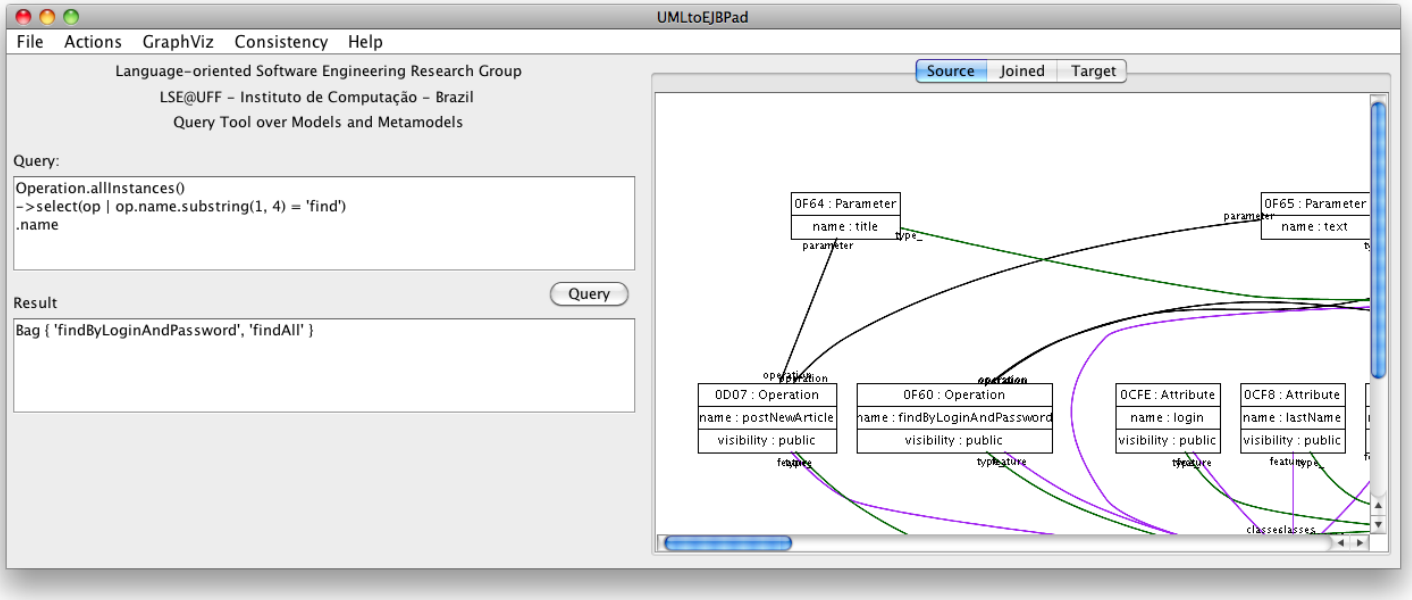


Figura 3.9: Protótipo do transformador `UMLtoEJB`

3.3.1 Exemplos utilizados

Durante a experimentação do protótipo, foram utilizados três diferentes diagramas de classe UML, objetivando verificar a utilidade do contrato de transformação na prática.

O primeiro diagrama, mais simples que os outros, é a modelagem de um blog[11], conforme apresentado na Figura 3.10.

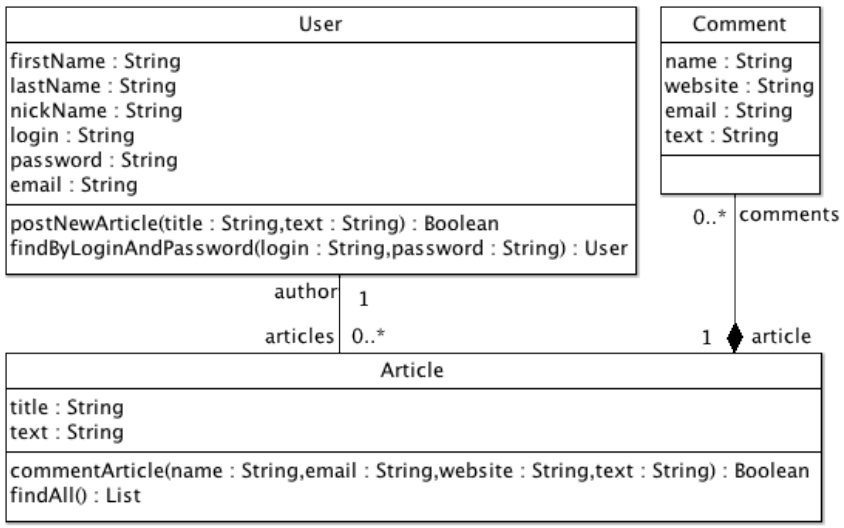


Figura 3.10: Diagrama de classe UML do exemplo do blog

[11] Contração do termo inglês *web log*, um blog é um site cuja estrutura permite criar e disponibilizar conteúdo na forma de artigos, sendo utilizado como um diário online ou para exibir notícias e comentários.

O modelo apresentado determina que usuários (classe *User*) criem artigos (classe *Article*) para visualização por visitantes, que podem escrever comentários (classe *Comment*). Como restrição da estrutura, todo artigo precisa de um autor, representado pela relação $author - articles$, e todo comentário pertence a um artigo, através da relação de composição $comments - article$. Essa visão, apesar de simples, modela a estrutura esperada para um blog.

O segundo diagrama de classe utilizado representa um sistema para gerenciamento de reuniões. Foi desenvolvido para complementar o modelo utilizado em [9], onde é apresentada a modelagem referente a política de segurança da aplicação para gestão de reuniões e visa permitir a integração entre o resultado do transformador implementado em [23] ao `UMLtoEJB`. (Esta integração ainda não foi realizada.) A Figura 3.11 apresenta o diagrama de classe UML referente ao aspecto funcional deste sistema, que chamado, para simplificar a sua referenciação, de *meeting*.

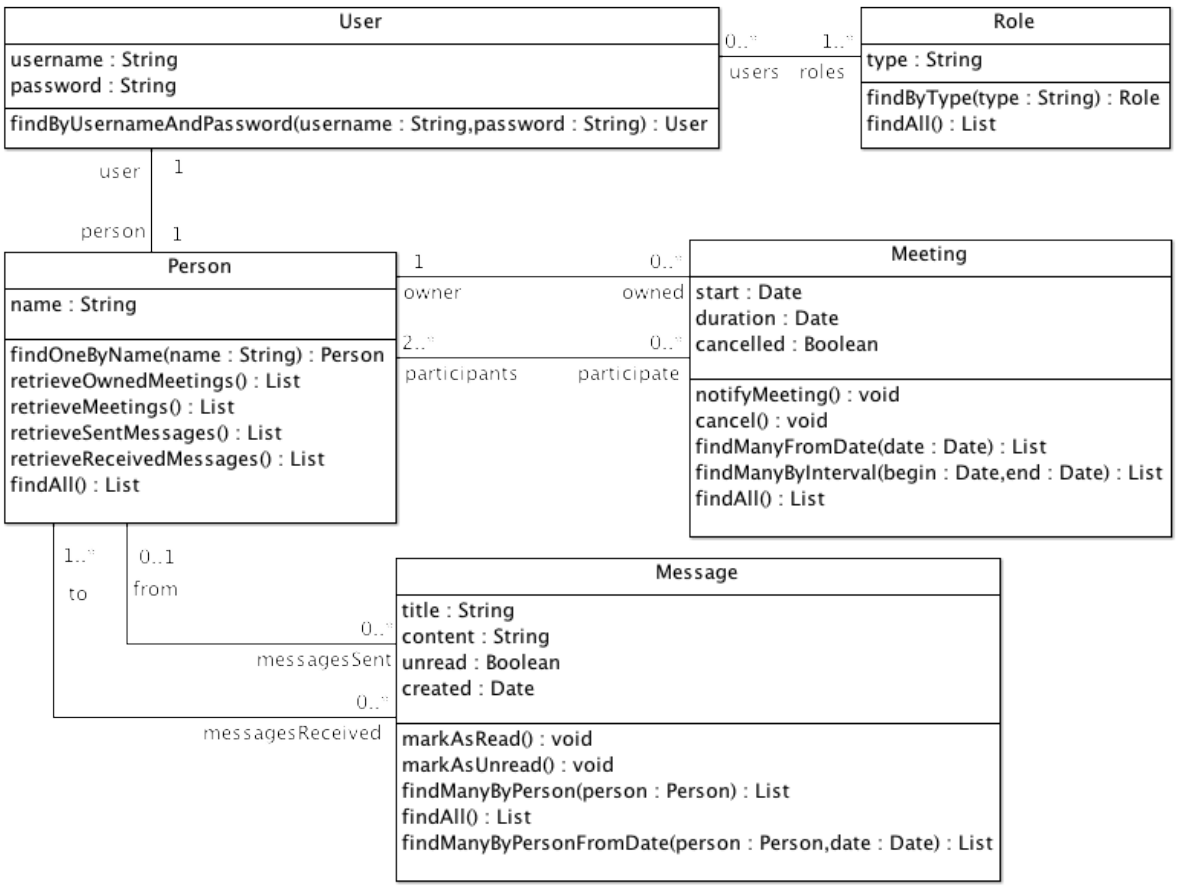


Figura 3.11: Diagrama de classe UML do exemplo chamado *meeting*

O diagrama de classe proposto possui duas partes importantes. A primeira, composta pelas classes *User* e *Role*, fornece informações sobre acesso de um usuário ao sistema e quais papeis ele pode exercer. Já a segunda parte, fornece mecanismos para o agendamento de reuniões (classe *Meeting*), composta por pessoas (classe Person), e para troca de mensagens (classe *Message*), que serve para notificar os participantes de uma reunião. A decisão de definir um usuário com duas classes, *User* e *Person*, foi baseada em duas observações: (i) transferir os dados de um usuário sem comprometer a segurança do sistema, como, e.g., fornecer o login e senha de um usuário para um cliente EJB, ao solicitar dados sobre uma reunião; (ii) facilitar o uso de frameworks de segurança, já que somente a parte referente a autorização será adaptada.

Por fim, o último exemplo foi extraído de [37]. Conhecido como *serviço de café da manhã da Rosa*[12], trata-se da modelagem de um sistema web para pedido de cestas de café da manhã. Este modelo será referenciado neste trabalho como *breakfast*.

A Figura 3.12 exibe o diagrama de classe utilizado, que possui modificações em relação a versão original.

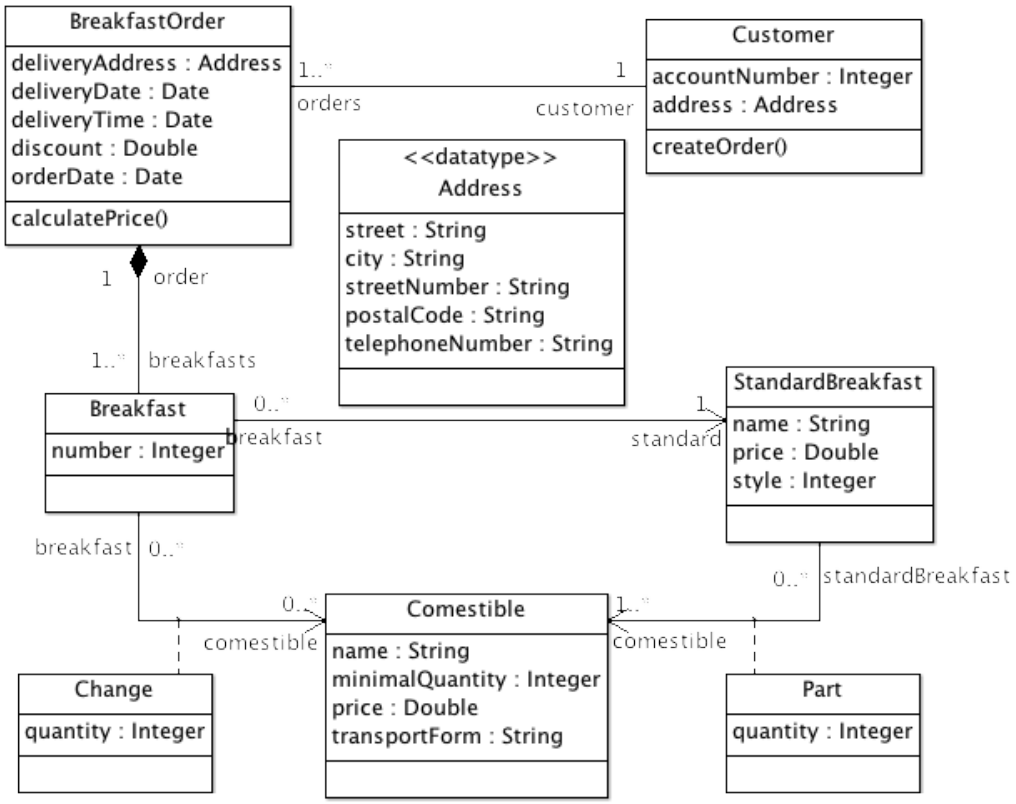


Figura 3.12: Diagrama de classe UML do exemplo chamado *breakfast*

[12]Em inglês, *Rosa's Breakfast Service*.

A modelagem *breakfast* permite que um freguês (classe *Customer*) faça um pedido (classe *BreakfastOrder*) de um ou mais cafés da manhã. Cada café da manhã deve seguir um dos tipos de cestas disponíveis (classe *StandardBreakFast*), que possui uma quantidade específica de itens comestíveis (classes Part e *Comestible*). Quando um freguês escolhe uma determinada cesta, pode acrescentar, retirar ou modificar a quantidade de itens comestíveis (classe Change).

As modificações realizadas, em relação a [37], são: (i) modificação do tipo do atributo *style*, pertencente a classe *StandardBreakfast*, para *Integer*, porque as informações sobre o tipo *Style*, utilizado originalmente, não são apresentadas; (ii) substituição de todos os tipos de dados *Real* para *Double*, porque a ferramenta de modelagem *ArgoUML* não reconhece *Real* como um tipo de dados, e sim como uma classe; (iii) adição da classe *Address*, referente aos dados de um endereço, que, apesar de utilizada na transformação apresentada em [37], não é especificada no modelo.

3.3.2 Benefícios do uso do contrato na construção e utilização do protótipo

O uso de contratos de transformação no `UMLtoEJB` auxiliou a detectar erros em diversas partes do transformador. Como apresentado no Capítulo 2, foi identificado três tipos de usuário: (i) designer da linguagem de modelagem; (ii) desenvolvedor da transformação de modelos; (iii) usuário do transformador de modelos. Nesta Seção, serão apresentados casos reais no qual o contrato de transformação auxiliou para cada tipo de usuário identificado.

Quando tratamos de modelos, pode existir três tipos diferentes envolvidos: (i) modelo mal-formado; (ii) modelo bem-formado e inválido; e(iii) modelo bem-formado e em conformidade com o seu metamodelo. Os dois primeiros tipos apresentam problemas que influenciam negativamente no processo de transformação.

No caso de um modelo mal-formado, a própria especificação do modelo apresenta erros em relação ao metamodelo. Problemas dessa natureza são resultado de um processo de *parsing* com problemas ou de transformação de elementos de forma incompleta e/ou errada.

Sob o ponto de vista de um designer da linguagem de modelagem, foi vivenciado o carregamento errado de classes associativas no metamodelo UML, que não a instanciava corretamente com suas pontas de associação. A propriedade chamada *restrictionMinimunOneAssociationEndPerAssociation*, apresentada na Seção 3.1.1, detectou esse caso,

já que as pontas de associação não estavam relacionadas com a classe associativa, que também é uma associação e precisa ter pelo menos duas ponta de associação.

Para o desenvolvedor do transformador, a transformação de uma classe em um componente de entidade EJB é complexa. Neste caso, o invariante *everyoutermostClassbecamesEntityComponent*, apresentado na Seção *cap4:contrato-transformacao*, detectou que todas as classes UML eram transformadas em componentes, o que está incorreto. A regra de transformação especifica que apenas classes que não possuem relações de composição com outras podem ser componentes de entidade.

Já para um modelo bem-formado porém inválido, os invariantes permitiram detectar erros referentes a modelos de entrada inválido e regras de transformação gerando instâncias inválidas. Por exemplo, para o usuário da transformação, foram detectados ciclos de herança, através do invariante *noCyclesinClassHierarchy*, e atributos com nomes iguais aos de fim de associação, através do invariante *onlyOneAttrOrAsscEndNameperClass*. Estes invariantes são apresentados na Seção 3.1.1.

A seguir é apresentado, de forma geral, em que pontos o uso do contrato de transformação auxiliou na construção do transformador `UMLtoEJB`.

- O uso de invariantes do metamodelo UML auxiliou a detecção de erros no processo de *parsing*, na leitura correta dos elementos do arquivo XMI e na representação do modelo como instância do metamodelo;
- O uso de invariantes para EJB permitiu validar se o resultado da transformação atendia as especificações feitas no metamodelo EJB;
- O uso de invariantes do modelo de transformação permitiu encontrar erros durante a implementação das regras de transformação;
- A verificação de consistência garantiu que os modelos utilizados e o resultado da transformação são instanciáveis.

A complexidade da transformação dificulta a detecção de que ponto do transformador está com erro. Os validadores permitiram a detecção de erros antes da geração do artefato final, que, juntamente com o mecanismo de rastreabilidade, apresentado na Seção 2.3.3, auxiliou a encontrar e corrigir qualquer anomalia.

Os exemplos apresentados na seção 3.3.1 foram utilizados para atestar a utilidade da ferramenta e analisar o impacto do uso de contratos de transformação na geração do artefato final.

O exemplo referente ao blog foi o primeiro a ser utilizado. O seu uso permitiu a correção de diversos problemas detectados pelos invariantes, referentes a geração da instância do metamodelo EJB. Após a correção da transformação, foram necessárias diversas alterações no método *pretty-print*, já que as especificações fornecidas por [37] não são suficientes para a execução correta da aplicação gerada. Para verificar a utilidade do transformador, foi criado um sistema WEB a partir do código-fonte gerado pela ferramenta `UMLtoEJB`[13].

Para confirmar os resultados obtidos até o dado momento, foi utilizada a modelagem do *meeting*, que foi concebido com características estruturais semelhante ao blog. Para este exemplo, a transformação foi executada com sucesso, sem a detecção de problemas. Então, foi construído um sistema WEB com base no código-fonte gerado pela transformação[14]. Com isso, o protótipo demostrou ser apto em transformar adequadamente diagramas de classe UML em código-fonte.

Até o momento, os exemplos não possuíam classes associativas. Com o modelo *breakfast*, foi possível detectar problemas relacionados a classes associativas, através de invariantes, no processo de *parsing* e na transformação. Após as correções, o código-fonte gerado foi analisado e constatou-se que atende as especificações EJB, podendo ser utilizado no desenvolvimento de um sistema.

3.4 Considerações finais

Contratos de transformação já foram utilizados com sucesso em linguagens específicas de domínio, especificamente na área de segurança [23]. `UMLtoEJB` foi capaz de usar os mesmos conceitos com linguagens de propósito geral, obtendo os benefícios já abordados.

Este trabalho também intencionou criticar a especificação da transformação de UML para EJB apresentada em [37], que possui pontos falhos referentes à definição de regras de transformação, dos metamodelos e do modelo *breakfast*, que são: (i) falta de documentação para entender elementos específicos do EJB como o conceito de *Manager*; (ii) documentação errada e incompleta referente ao processo de *pretty-printing* do metamodelo EJB; (iii) ausência do metamodelo refinado do EJB, que foi definido vagamente, assim como a sua transformação, resultando no aumento da complexidade no método *pretty-print* implementado; (iv) explicação superficial da transformação de UML para EJB; (v) ausência de informações no modelo *breakfast* em relação as classes *Address* e

[13]Para visualizá-lo, acesse `http://lse.ic.uff.br:8080/ExampleBlog-web/`
[14]Para visualizá-lo, acesse `http://lse.ic.uff.br:8080/ExampleMeeting-web/`

Style; (vi) inconsistência na definição de associações navegáveis, que violava restrições de cardinalidade no metamodelo UML; (vii) inconsistência na definição de composição de associações, dificultando a compreensão das *outermost classes*. Os itens (i-v) foram encontrados durante a análise da transformação e foram solucionados através do estudo sobre EJB e discussões envolvendo a especificação. Já os itens (vi) e (vii) foram encontrados durante o desenvolvimento, o que ocasionou em problemas referente a transformação, principalmente da metaclasse *AssociationClass*.

A separação em domínios auxiliou a isolar os problemas conforme foram descobertos e a especificação do contrato ajudou a detectar os pontos exatos que estavam as inconsistências após a sua descoberta. Com isto, foi visto que [37] não possui informações suficientes para descrever um transformador executável.

Durante o desenvolvimento foi constatado que a ferramenta utilizada para interpretar OCL não verifica as restrições de cardinalidade nas instâncias dos metamodelo. Portanto, foi necessário implementar invariantes para suprir tal deficiência. Como não são uma contribuição ao trabalho, eles não foram listados junto com as propriedades de cada metamodelo. Além disto, por causa das limitações do interpretador, os metamodelos precisaram atender as seguintes restrições: (i) todas as pontas de associação devem ter nome e cardinalidade; (ii) não pode existir associações direcionadas; (iii) não pode existir associações de composição.

Após analisar os experimentos realizados, foi constatado que o contrato de transformação especificado detectou erros no processo de *parsing* e na transformação. Além de evitar a geração de código-fonte defeituoso, auxiliou a determinar quais partes do transformador apresentavam os problemas de implementação. O recurso de rastreabilidade, por sua vez, ajudou na investigação pela causa da falha do invariante, permitindo encontrar a causa das falhas diretamente sobre os elementos fornecidos. A verificação de consistência, além de ajudar a detectar erros, aumentou a confiança sobre a transformação executada, por garantir a instanciabilidade dos modelos gerados.

Devido a expressividade das linguagens envolvidas, o código-fonte gerado pelo transformador `UMLtoEJB` não é suficiente para a sua execução imediata, precisando da intervenção de desenvolvedores para a adição dos comportamentos associados aos chamados *business methods*. O transformador gera toda a infraestrutura EJB necessária. O uso de metamodelos mais ricos e transformações mais completas, com a interação de mais metamodelos de entrada e saída, podem reduzir a necessidade de customizações. Em [37], além de transformar um diagrama de classe UML para código-fonte com EJB, define a

transformação para SQL e *JavaServer Pages* (JSP[15]).

Por fim, os exemplos utilizados foram cruciais para a correção do transformador, detectando instâncias dos metamodelos que eram mal-formadas ou inválidas devido a erros na implementação ou nos modelos de entrada. A combinação de contratos de transformação com mecanismos de geração de modelos teste, isto é, a aplicação de técnicas para geração automática de modelos de entrada, baseando-se na estrutura do metamodelo de entrada, podem auxiliar no teste da transformação.

[15]JSP é uma tecnologia para desenvolvimento de aplicações WEB em Java. Assim como EJB, ela faz parte da especificação J2EE.

Capítulo 4

Uma abordagem baseada em contratos de transformação para aplicações sísmicas

Este capítulo trata da aplicação do processo de DDM com contratos de transformação ao desenvolvimento de aplicações sísmicas. Ele é resultado do projeto piloto "Modelagem e validação de aplicações sísmicas" com a *Schlumberger Brazil Research and Geoengineering Center* (BRGC)[1]. O trabalho descrito neste capítulo foi publicado em [15].

O processo de extração de óleo envolve a análise, por um geofísico, de dados coletados de um reservatório[2] através de sensores posicionados na superfície do oceano. Dados coletados em alto mar são geralmente obtidos por embarcações, que emitem ondas em direção a potenciais áreas de exploração e carregam sensores para capturar o retorno das ondas, que refletem no fundo do oceano[3]. A decisão pela exploração de uma certa área é fortemente baseada na *interpretação* dos dados coletados num processo como este. Erros nesta etapa podem resultar em prejuízos na escala de bilhões de dólares, sem mencionar os efeitos negativos causados ao meio-ambiente.

No contexto de aplicações sísmicas, DDM com contratos de transformação pode ajudar desenvolvedores de software a criar aplicativos para auxiliar geofísicos na interpretação de informações geofísicas.

Além disso, o uso de técnicas formais com DDM podem ajudar tanto o geofísico quanto o desenvolvedor de software a ganhar confiança de que um dado modelo está correto em

[1] Para mais detalhes sobre a BRGC, acessar http://www.slb.com/about/rd/research/sbr.aspx. Em relação ao projeto realizado, ver em http://www.lse.ic.uff.br.

[2] Segundo o *Schlumberger Oilfield Glossary*, disponível em http://www.glossary.oilfield.slb.com/, um reservatório (em inglês, *reservoir*) é um corpo subterrâneo de pedra, tendo porosidade e permeabilidade suficientes para armazenar e transmitir fluidos.

[3] Maiores detalhes disponíveis em [48, cap. 4]

relação às propriedades do domínio sísmico e à plataforma de desenvolvimento. A aplicação resultante da transformação com a verificação dessas propriedades e a formalização da transformação garantem um maior nível de confiança do que uma aplicação desenvolvida de forma *ad-hoc*.

Este capítulo discute a aplicação da técnica de contratos de transformação no desenvolvimento de aplicações sísmicas, que é um resultado do projeto de pesquisa realizado em conjunto com a BRGC. Nesse projeto, foram criadas duas linguagens de modelagem: (i) *Seismic Domain Modeling Language* (SDML), uma linguagem de modelagem específica de domínio para a especificação de modelos sísmicos, que permite modelar estruturas ou organizações para processamento de dados sísmicos; (ii) *Parallel Process Modeling Language* (PPML), uma linguagem de modelagem de processos paralelos sobre estruturas de dados. Uma visão detalhada sobre tais metamodelos e suas propriedades são apresentados na Seção 4.1.

Já na Seção 4.2, é apresentado o contrato de transformação especificado, que compreende o modelo de transformação, que relaciona SDML à PPML, e suas propriedades.

O contrato definido, na Seção 4.2, foi implementado no transformador `SDMLtoPPML`, de forma semelhante ao apresentado na Seção 3.3. Com relação aos experimentos, um exemplo foi extraído de [48]. Com isto, pretende-se validar a expressividade de SDML e PPML no contexto das aplicações desenvolvidas no BRGC. A Seção 4.3 detalha a implementação do transformador e o exemplo utilizado.

Na Seção 4.4 são apresentadas considerações finais sobre o experimento e uma reflexão sobre os resultados obtidos.

4.1 Domínios envolvidos

Desenvolvimento dirigido a modelo e contratos de transformação utilizam metamodelos e suas instâncias para representar um modelo. Um domínio, conforme apresentado na Seção 2, é definido com o metamodelo referente ao conhecimento que ele representa, pelo método de *parsing*, de *pretty-printing* e seus validadores. Logo, um domínio representa uma linguagem de modelagem com sua sintaxe concreta definida pela função de *parsing*, sua sintaxe abstrata dada pelo metamodelo e sua semântica descrita pelas propriedades a serem verificadas pelos seus validadores.

A seguir serão abordados os domínios SDML e PPML, que são, respectivamente, os

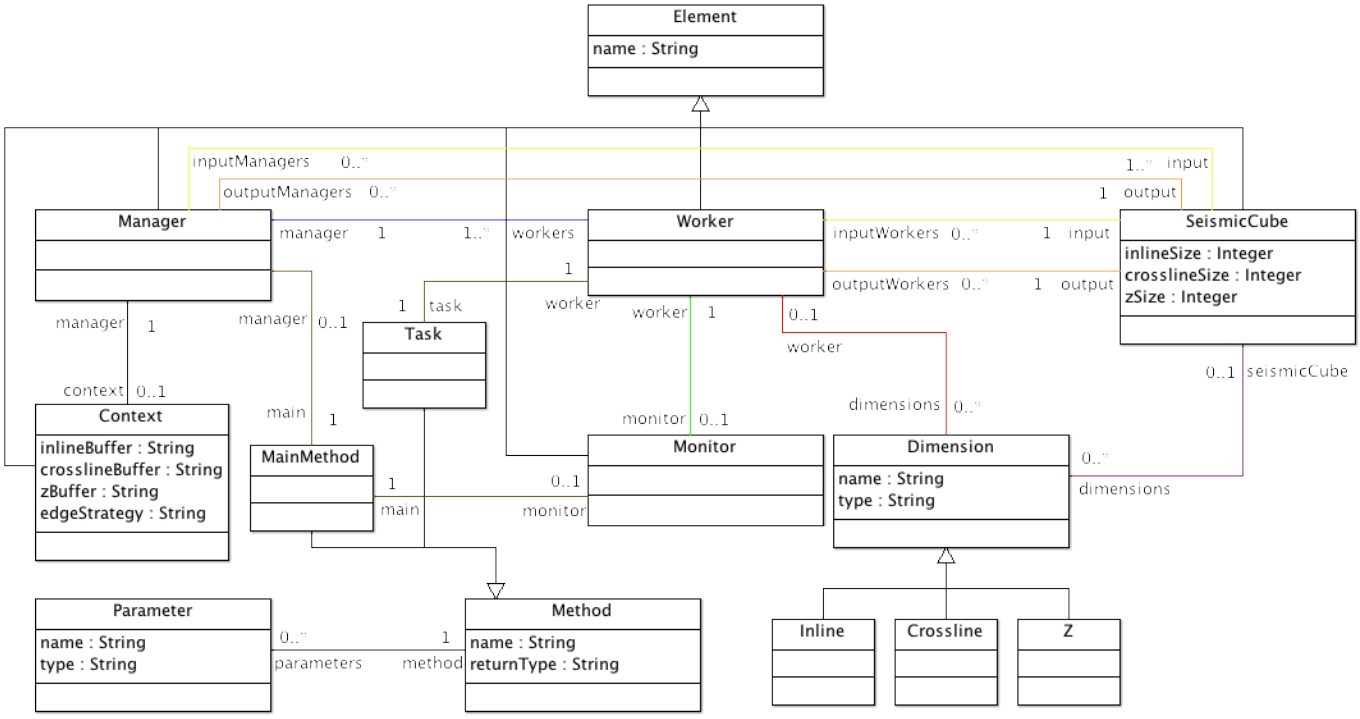


Figura 4.1: Metamodelo SDML utilizado

domínios de entrada e saída do transformador `SDMLtoPPML`.

4.1.1 Domínio SDML

Segundo [48], a principal fonte de informações do domínio neste trabalho, dados sísmicos representam a amplitude medida por refletores sísmicos em determinados pontos no espaço. SDML, abreviação de *Seismic Domain Modeling Language*, é uma linguagem específica de domínio para modelar o processamento de dados sísmicos. Esta linguagem foi definida com base em [48], em uma parceria com a BRGC. A Figura 4.1 apresenta o diagrama de classe que representa o metamodelo de SDML.

A metaclasse *SeismicCube* é a estrutura de dados que representa dados sísmicos modelados como uma estrutura com três dimensões, conhecida como cubo sísmico. Dimensões são definidas por instâncias da metaclasse *Dimension*, que possuem um nome (atributo *name*) e tipo (atributo *type*).

Cubos sísmicos são representados por três dimensões, que são: (i) *inline* (metaclasse *Inline*), simbolizando a direção na qual os dados foram coletados; (ii) *crossline* (metaclasse *Crossline*), que representa a direção ortogonal a *inline*; (iii) *z* (metaclasse *Z*), que significa

a passagem do tempo na coleta dos dados.

Certos cubos sísmicos podem ser muitos grandes para que sejam carregados ou processados de forma adequada. A metaclasse *Manager* é responsável por gerenciar como os cubos sísmicos serão processados, que podem ou não serem divididos de diversas formas.

Instâncias de *Worker* representam elementos responsáveis pelo processamento dos dados sísmicos, seja ele um cubo sísmico inteiro ou apenas uma parte. Um exemplo de processamento de cubos sísmicos é a aplicação de um filtro para visualização de uma determinada faixa de amplitude da onda. Utiliza-se instâncias de *Dimension* para definir sobre quais dimensões do cubo sísmico de entrada um dado trabalhador atuará e outras características referentes ao processamento dos dados do cubo sísmico. Como SDML não trata de aspectos comportamentais, apenas as definições da tarefa são modeladas, por instâncias da classe *Task*.

Já a metaclasse *Monitor* representa entidades que monitoram o comportamento de instâncias de *Worker*, para registrar eventuais problemas e informações sobre o processamento realizado. Semelhante a *Task*, *MainMethod* define a estrutura de execução para monitores ou gerentes (metaclasse *Manager*). Tanto *Task* e *MainMethod* herdam de *Method*, representando um método, que pode possuir parâmetros (metaclasse *Parameter*).

Cada instância de *Worker* processa pequenas partes de um cubo sísmico, definido por uma instância de *Manager*. O processamento desses dados geralmente é complexo, principalmente ao analisar informações nos limites das dimensões de um sub-cubo, chamadas aqui de borda. Instâncias de *Context* especificam o tratamento da borda de um sub-cubo, para auxiliar as instâncias de *Manager* a organizar um *SeismicCube* adequadamente. Para tal, é possível informar o tamanho extra fornecido para processamento, pelos atributos *inlineBuffer*, *crosslineBuffer* e *zBuffer*, e a estratégia de tratamento da borda, que representa o comportamento a ser adotado no processamento dos dados limites de um cubo. Apesar do metamodelo não restringir os tipos de estratégias, uma vez que o modelo trata dos aspectos estruturais de um modelo, as mais comuns são: (i) espelhamento dos dados do sub-cubo; (ii) uso do valor mais externo da dimensão em questão; ou (iii) um valor constante definido previamente.

Por fim, todos os elementos do metamodelo, com exceção de *Method*, *Parameter* e *Dimension*, além das suas hierarquias de herança, também são do tipo *Element*. Esta metaclasse representa os elementos que são definidos como classes estereotipadas em um diagrama de classe.

A seguir são apresentadas as propriedades em OCL pertencentes a SDML[4].

- Instâncias de *Worker*, agrupadas por um dado *Manager*, devem possuir o mesmo cubo sísmico de destino, i.e, a mesma instância de *SeismicCube* referente ao cubo sísmico de destino de seus gerentes.

```
context Manager inv EveryWorkerHaveTheSameOutput:
self.workers->forAll(w | w.output = self.output)
```

- Os cubos sísmicos relacionados com uma instância de *Worker* devem possuir o mesmo tamanho, i.e., os mesmos valores em relação às suas dimensões.

```
context Worker inv CubesMustHaveSameSize:
if (self.input->size() > 0 and self.output->size() > 0) then
  self.input->forAll(input | input.inlineSize = self.output.inlineSize and input.crosslineSize = self.output.crosslineSize and
      input.zSize = self.output.zSize)
else
  true
endif
```

- Toda instância de *Manager* precisa de somente um cubo sísmico de destino e ele não pode ser um dos cubos de entrada.

```
context Manager inv ManagerMustHaveAOutputcube:
self.output->size() = 1 and not self.input->includes(self.output)
```

- O tamanho definido para um contexto sempre será menor que o tamanho dos cubos sísmicos de entrada da instância de *Manager* associada. Para isto, restringe-se o tamanho das instâncias de *Context* para que não sejam fornecidos todos os dados do cubo a uma instância de *Worker*.

```
context Context inv ContextSmallerThanItsCubes:
self.manager.input->forAll(sc : SeismicCube | self.inlineBuffer < sc.inlineSize and self.crosslineBuffer < sc.crosslineSize and
    self.zBuffer < sc.zSize)
```

- Todo trabalhador, i.e., instância de *Worker*, precisa ter pelo menos uma das seguintes dimensões vinculadas: (i) *Inline*; (ii) *Crossline*; (iii) *Z*.

```
context Worker inv WorkerMustHaveAtLeastAneSpecialDimension:
self.dimensions->exists(c | c.oclIsTypeOf(Z) or c.oclIsTypeOf(Inline) or c.oclIsTypeOf(Crossline))
```

- Cada trabalhador deve possuir diferentes cubos sísmicos de entrada e saída.

[4]Apesar de existir formas mais simples de especificar os invariantes do metamodelo SDML sem mudar seu sentido, eles são executáveis na ferramenta *EOS*, que possui algumas idiossincrasias em relação a interpretação das expressões OCL.

```
context Worker inv WorkerMustHaveDifferentInputAndOutput:
self.input <> self.output
```

- Cada cubo sísmico deve possuir três dimensões, referentes às metaclasses *Inline*, *Crossline* e *Z*.

```
context SeismicCube inv MustHaveInlineCrosslineZ:
self.dimensions->exists(inline, crossline, z | inline.oclIsTypeOf(Inline) and crossline.oclIsTypeOf(Crossline) and z.oclIsTypeOf
    (Z))
```

- Toda dimensão deve possuir um nome e um tipo declarado, representados pelos atributos *name* e *type*.

```
context Dimension inv DimensionMustHaveNameAndType:
self.name.size() > 0 and self.type.size() > 0
```

- Toda instância de *Method* deve possuir um nome e um tipo de retorno declarado.

```
context Method inv MethodMustHaveNameAndType:
self.name.size() > 0 and self.returnType.size() > 0
```

- Cada instância de *Parameter* precisa de um nome e tipo declarado, semelhante a instâncias de *Dimension*.

```
context Parameter inv ParameterMustHaveNameAndType:
self.name.size() > 0 and self.type.size() > 0
```

- Toda instância de *Element* precisa do seu atributo *name* preenchido.

```
context Element inv ElementsMustHaveName:
self.name.size() > 0
```

- Cada instância de *MainMethod* deve pertencer ou a um *Monitor* ou a um *Manager*.

```
context MainMethod inv MainMethodMustBelongToMonitorOrManager:
self.manager->size() = 1 xor self.monitor->size() = 1
```

- Instâncias de *Context* devem possuir valores maiores ou iguais a zero para os atributos *inlineBuffer*, *crosslineBuffer* e *zBuffer*.

```
context Context inv FillAllContextInformation:
self.inlineBuffer >= 0 and self.crosslineBuffer >= 0 and self.zBuffer >= 0
```

- Instâncias de *SeismicCube* devem possuir os valores dos atributos *inlineSize*, *crosslineSize* e *zSize* maiores que zero.

```
context SeismicCube inv FillAllCubeInformation:
self.inlineSize > 0 and self.crosslineSize > 0 and self.zSize > 0
```

- O cubo sísmico de entrada de um *Manager* deve ser idêntico aos das suas instâncias de *Worker*.

```
context Manager inv InputCubesInManagerAndItsWorkers:
self.workers.input->asSet() = self.input
```

Diagramas SDML são descritos em XMI e a construção de um modelo como instância do metamodelo de SDML se dá de forma semelhante ao apresentado na Seção 3.1.1.

O modelo de entrada é descrito utilizando-se um *profile* UML definido para SDML. Para as classes que herdam de *Element* (*Manager*, *Context*, *Worker*, *Monitor* e *SeismicCube*), foram criados estereótipos com mesmo nome para simbolizá-las, aplicáveis somente em classes UML. Já para as metaclasses *Task* e *MainMethod*, foram criadas estereótipos, aplicáveis a métodos, com, respectivamente, os nomes *«Task»* e *«Executor»*.

É necessário definir explicitamente quais atributos de *Worker* e *SeismicCube* serão instâncias de *Dimension*. Os tipos *Inline*, *Crossline* e *Z*, que definem um *Dimension* são utilizados através dos seguintes estereótipos: (i) *«inline»*; (ii) *«crossline»*; (iii) *«z»*.

Por fim, são utilizados comentários estruturados nas classes *Context* e *SeismicCube* para definir suas estratégias de borda e dimensões, respectivamente, como descrito a seguir.

- Comentários em *Context* permitem a declaração dos valores dos seguintes atributos: (i) *inlineBuffer*; (ii) *crosslineBuffer*; (iii) *zBuffer*; (iv) *edgeStrategy*.
- Comentários em *SeismicCube*, fornecem as informações necessárias para a definição dos valores dos atributos *inlineSize*, *crosslineSize* e *zSize*.

A Figura 4.2 apresenta um modelo em SDML válido, cuja modelagem será explicada em detalhes na Seção 4.3.1. Ele é utilizado nesta seção para auxiliar na compreensão da sintaxe de SDML.

No protótipo atual, SDML é utilizada como linguagem de modelagem de origem na transformação, e conseqüentemente não definimos o método *pretty-print* para este domínio.

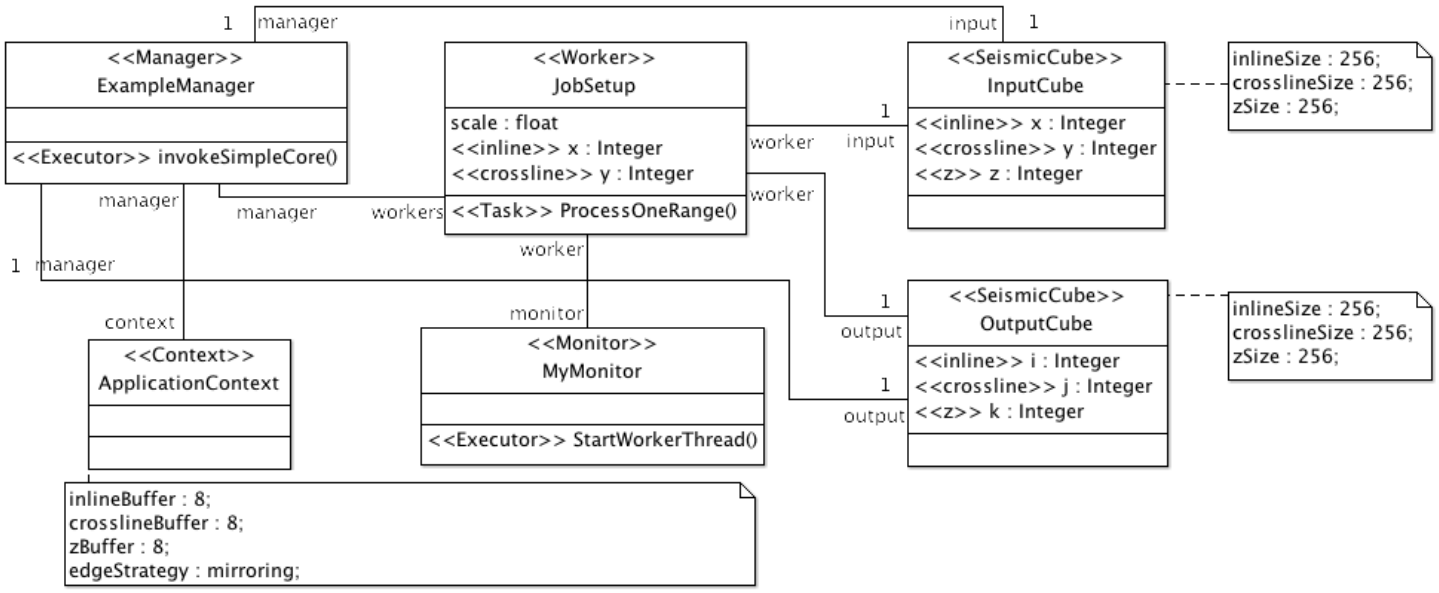


Figura 4.2: Modelo em SDML válido

Por fim, o domínio SDML apresentado possui as seguintes características: (i) um metamodelo capaz de representar modelos para processamento de dados sísmicos; (ii) propriedades para validação da instância desse metamodelo; (iii) processo de *parsing* para carregar um diagrama de classe UML, com estereótipos específicos, como instância do metamodelo; e (iv) um verificador de consistência em lógica de descrição, utilizando o validador apresentado na Seção 2.3.3.

4.1.2 Domínio PPML

PPML, abreviação de *Parallel Processing Modeling Language*, é uma linguagem de modelagem para o processamento paralelo de dados compartilhados. Tem o objetivo de ser uma linguagem simples e foi criada principalmente para servir como exemplo de domínio de destino no projeto "Modelagem e validação de aplicações sísmicas". Não tem por objetivo, portanto, funcionar como uma linguagem de espectro amplo, ou de descrição arquitetural. Foi desenhada para atender aos objetivos específicos do projeto sendo, no entanto, bastante genérica e expressiva.

A idéia básica é a de que PPML possa representar diferentes mecanismos de paralelismo, seja o *Master-Slave*[5] (o utilizado neste projeto) ou *peer to peer* [21]. PPML realiza paralelismo em tarefas (em inglês, *task parallelism*), que é uma forma para a execução con-

[5]Segundo [16], o padrão de projeto *Master-Slave* suporta tolerância a falha e processamento paralelo. O componente mestre distribui o trabalho para componentes escravos idênticos e computa o resultado final dos resultados que estes escravos retornam.

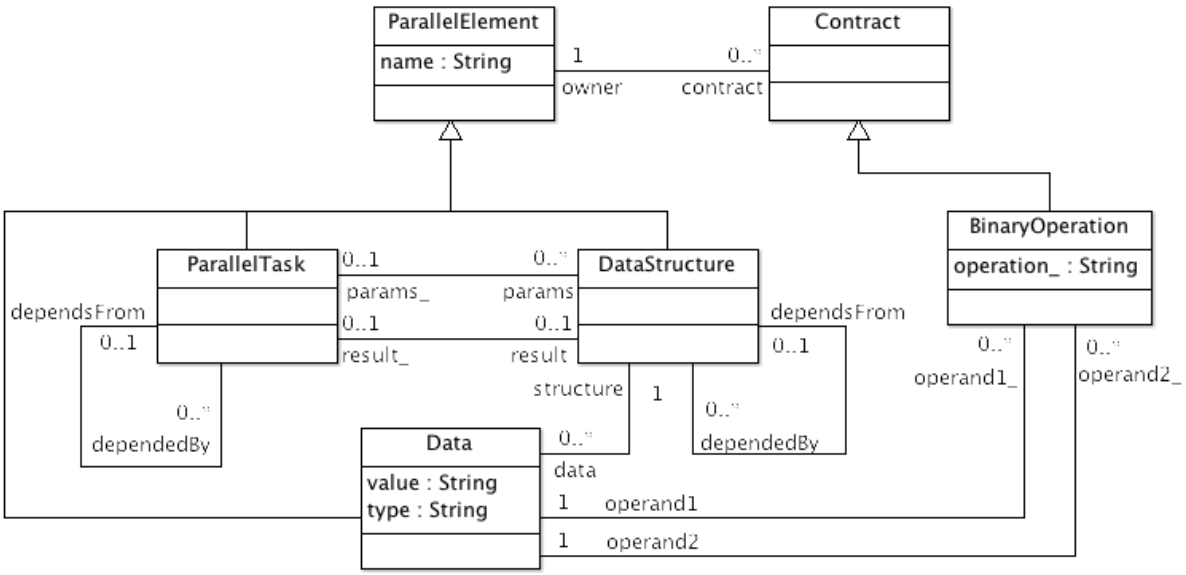


Figura 4.3: Metamodelo PPML utilizado

corrente de processos, ou *threads*, em diferentes nós de processamento. Mais informações em [27]. A Figura 4.3 apresenta o metamodelo PPML.

Instâncias de *ParallelTask* simbolizam as tarefas a serem executadas em paralelo. Para este caso, pode existir uma relação de dependência entre as tarefas, o que simboliza uma hierarquia de execução das tarefas.

A metaclasse *DataStructure* representa as estruturas de dados utilizadas no processamento de uma tarefa, que podem ser parâmetros de inicialização ou o resultado da execução. Semelhante à *ParallelTask*, também é possível definir uma relação de dependência, para simbolizar estruturas dependentes de dados de outras estruturas. Uma instância de *DataStructure* se relaciona com várias instâncias de *Data*, que definem tipos de dados e seus valores.

As metaclasses *ParallelTask*, *DataStructure* e *Data* modelam o paralelismo suportado pelo metamodelo. Logo, elas herdam de *ParallelElement*, que define a informação referente ao nome das instâncias.

A metaclasse *Contract* especifica uma restrição em qualquer das instâncias de *ParallelElement*, que será linearizada (*pretty-printed*) com uma assertiva, a ser verificada em tempo de execução. Para este projeto, foi criado apenas um tipo de contrato chamado *BinaryOperation*, que valida se duas instâncias de *Data* seguem uma dada operação, que pode ser, por exemplo, de igualdade ou diferença.

PPML é uma linguagem de modelagem simples e, portanto, possui poucas proprieda-

des, conforme apresentado a seguir[6].

- Todo instância de *ParallelElement* deve possuir o atributo *name* preenchido.

```
context ParallelElement inv ParallelElementMustHaveName:
self.name.size() > 0
```

- As instâncias de *Data* precisam ter um tipo definido, simbolizado pelo atributo *type*.

```
context Data inv DataMustHaveType:
self.type.size() > 0
```

- Todo instância de *BinaryOperation* precisa possuir o atributo *operation_* preenchido.

```
context BinaryOperation inv BinaryOperationMustHaveOperation:
self.operation.size() > 0
```

- Não é permitida a existência de ciclos na hierarquia de dependência entre instâncias de *ParallelTask*, ou seja, uma tarefa paralela não pode se relacionar pela associação representada pelas pontas de associação chamadas *dependsFrom* e *dependedBy*, seja diretamente ou pelo caminho induzido pela associação entre várias tarefas paralelas.

```
context ParallelTask inv noCyclesinParallelTaskHierarchy:
self.dependsFrom->forAll(pt : ParallelTask | pt.superPlusParallelTask()->excludes(self))

context ParallelTask::superPlusParallelTask():Set(ParallelTask) def:
self.superPlusParallelTaskOnSet(self.emptySet())

context ParallelTask::superPlusParallelTaskOnSet(rs:Set(ParallelTask)):Set(ParallelTask) def:
if self.dependsFrom->notEmpty() and rs->excludes(self) then
  self.dependsFrom->collect(pt : ParallelTask | pt.superPlusParallelTaskOnSet(rs->including(self)))->flatten()->asSet()
else
  rs->including(self)
endif

context ParallelTask::emptyParallelTaskSet():Set(ParallelTask) def:
ParallelTask.allInstances()->select(pt | false)
```

- Também não é permitida a existência de ciclos na hierarquia de dependência entre instâncias de *DataStructure*, através da associação representada pelas pontas de associação chamadas *dependsFrom* e *dependedBy*, seja diretamente ou pelo caminho induzido pela associação entre várias estruturas de dados.

[6]Apesar de existir versões simplificadas dos invariantes do metamodelo PPML, eles são executáveis na ferramenta *EOS*.

```
context DataStructure inv noCyclesinDataStructureHierarchy:
self.dependsFrom->forAll(ds : DataStructure | ds.superPlusDataStructure()->excludes(self))

context DataStructure::superPlusDataStructure():Set(DataStructure) def:
self.superPlusDataStructureOnSet(self.emptySet())

context DataStructure::superPlusDataStructureOnSet(rs:Set(DataStructure)):Set(DataStructure) def:
if self.dependsFrom->notEmpty() and rs->excludes(self) then
  self.dependsFrom->collect(ds : DataStructure | ds.superPlusDataStructureOnSet(rs->including(self)))->flatten()->asSet
      ()
else
  rs->including(self)
endif

context DataStructure::emptyDataStructureSet():Set(DataStructure) def:
DataStructure.allInstances()->select(ds | false)
```

PPML é tratado como do domínio de destino da transformação neste protótipo. Logo, não foi implementado o processo de *parsing*. Já o método *pretty-print* foi implementado, gerando código-fonte para C# e o framework *Ocean/Petrel* (Uma outra abordagem para o processo de *pretty-printing*, que não foi estudada, envolve transformar PPML para um outro metamodelo, como a UML, que permite a geração de código-fonte em C#, o que envolve a especificação de outro contrato de transformação para essa nova transformação.).

Independentemente da instância do metamodelo PPML, sempre são criadas automaticamente certas classes, conforme a seguir: (i) classes para simular a plataforma *Ocean/Petrel* da *Schlumberger*; (ii) uma classe chamada *Dimension*, que representará instâncias de *Data* que possuem o tipo de dados *Dimension*; (iii) uma classe chamada *DimensionType*, que define o tipo da dimensão; (iv) classes chamadas *Inline*, *Crossline* e *Z*, que herdam de *DimensionType* e representam, junto com a classe *Dimension*, as instâncias de *Data* que possuem o tipo *Dimension<Inline>*, *Dimension<Crossline>* ou *Dimension<Z>*; (v) a classe chamada *SeismicAccessWorkset*, que possuirá toda a estrutura necessária para o processamento, de acordo com a instância do metamodelo PPML. Estas classes são criadas devido ao fato de que o método *pretty-print* implementado utiliza o *framework Ocean/Petrel*, que requer um conjunto de classes para o seu uso correto.

Cada instância da metaclasse *ParallelTask* gerará uma declaração de método na classe *SeismicAccessWorkset*, classe principal para o processamento paralelo. Em C#, *threads* são executadas a partir de métodos. Portanto, instâncias de *ParallelTask* geram declarações de métodos. Caso alguma instância de *ParallelTask* possua uma relação de dependência com outra *ParallelTask*, esta informação será descrita em um comentário

```
public void setInlineBuffer(int inlineBuffer) {
    if(!(inlineBuffer <= inputCube.getX())){
      throw new System.InvalidOperationException("Contract wasn't fulfilled.");
    }
    this.inlineBuffer = inlineBuffer;
}
```

Figura 4.4: Trecho do código-fonte de *SeismicAccessWorkset*, referente a uma instância da metaclasse *BinaryOperation*

no código-fonte porque não são abordados aspectos comportamentais em PPML. Caso existam parâmetros para a *ParallelTask* em questão, eles também serão informados como comentários porque em C# todo método que é uma *thread* deve possuir apenas um parâmetro do tipo *object*, o que obriga a linearização correta dos parâmetros em um único objeto, por meio de uma estrutura de dados.

Para cada instância de *DataStructure*, será criada uma classe, que possuirá as seguintes características: (i) para cada estrutura da qual depende e instância de *Data*, será declarado um atributo, um método *getter* e um método *setter*; e (ii) um construtor que possui como parâmetros todos os atributos declarados.

Durante a geração do *setter* referente a uma instância de *Data*, caso exista um *BinaryOperation* relacionado pela associação $contract - owner$, será criada a restrição antes da atribuição dos valores. Esta restrição é uma condição para garantir que ambos os operandos relacionados com o contrato respeitem a operação definida pelo atributo *operation_*. A Figura 4.4 apresenta o trecho da classe *SeismicAccessWorkset* referente a esta restrição.

O domínio PPML permite representar modelos para processamento de tarefas com a capacidade de validação em tempo de execução dada pelas restrições em *Contract*. Por fim, é possível gerar código-fonte em C#, seguindo as definições propostas em [48].

Então, o domínio PPML apresentado possui as seguintes características: (i) um metamodelo capaz de representar diferentes tipos de paralelismos de tarefas; (ii) propriedades para validação da instância desse metamodelo; (iii) processo de *pretty-printing* para geração de código-fonte em C# utilizando o framework *Ocean/Petrel*; e (iv) um verificador de consistência em lógica de descrição, utilizando o validador apresentado na Seção 2.3.3. A existência destas características são essenciais para a transformação de modelo proposta neste capítulo.

4.2 Contrato de transformação

O contrato de transformação entre SDML e PPML modela como aplicações sísmicas podem ser refinadas como tarefas paralelas usando o padrão de projeto *Master-Slave*. Para tal, é necessário definir o modelo de transformação, chamado de modelo *SDMLtoPPML*, e suas propriedades.

Semelhante ao modelo *UMLEJB*, apresentado no Capítulo 3, *SDMLtoPPML* relaciona os elementos dos metamodelos SMDL e PPML através de classes, que simbolizam as relações entre as linguagens de modelagem, conforme apresentado na Figura 4.5. A seguir são apresentadas as classes referentes ao modelo de transformação.

- *ManagerToTask*: Relaciona instâncias de *Manager* e *MainMethod*, relacionadas entre si, com *ParallelTask*. As informações referentes ao contexto e os cubos de entrada serão representadas como parâmetros da tarefa, assim como o cubo de saída será o resultado.

- *MonitorToTask*: Simboliza a transformação de monitores (metaclasse *Monitor* e *MainMethod*) em tarefas paralelas (metaclasse *ParallelTask*). Por fim, cada tarefa criada depende da tarefa resultante da transformação da instância de *Worker*, que é monitorada no metamodelo de origem.

- *WorkerToTask*: Representa a transformação de trabalhadores, definidos pelas metaclasses *Worker* e *Task*, para tarefas paralelas e estruturas de dados, que armazenam a região que serão utilizadas nos cubos sísmicos de entrada e saída. Além disto, esta tarefa paralela dependerá da instância de *ParallelTask*, criada a partir do gerente responsável pelo trabalhador no metamodelo SDML.

- *SeismicCubeToDataStructure*: Define a relação entre cubos sísmicos e estruturas de dados. Para tanto, é necessária a criação de três instâncias de *Data*, uma para cada dimensão do cubo, dadas por *inline*, *crossline* e *z*.

- *WorkerDimensionToData*: Define a relação entre as dimensões pertencentes a instâncias de *Worker* e *Data*. Com isto, as características referentes aos trabalhadores são registradas na instância de *DataStructure*, criada pelo mapeamento definido por *WorkerToTask*.

- *ContextToDataStructure*: Transforma cada instância de *Context* em uma *DataStructure* pertencente a uma tarefa paralela, que simboliza a instância de *Manager*

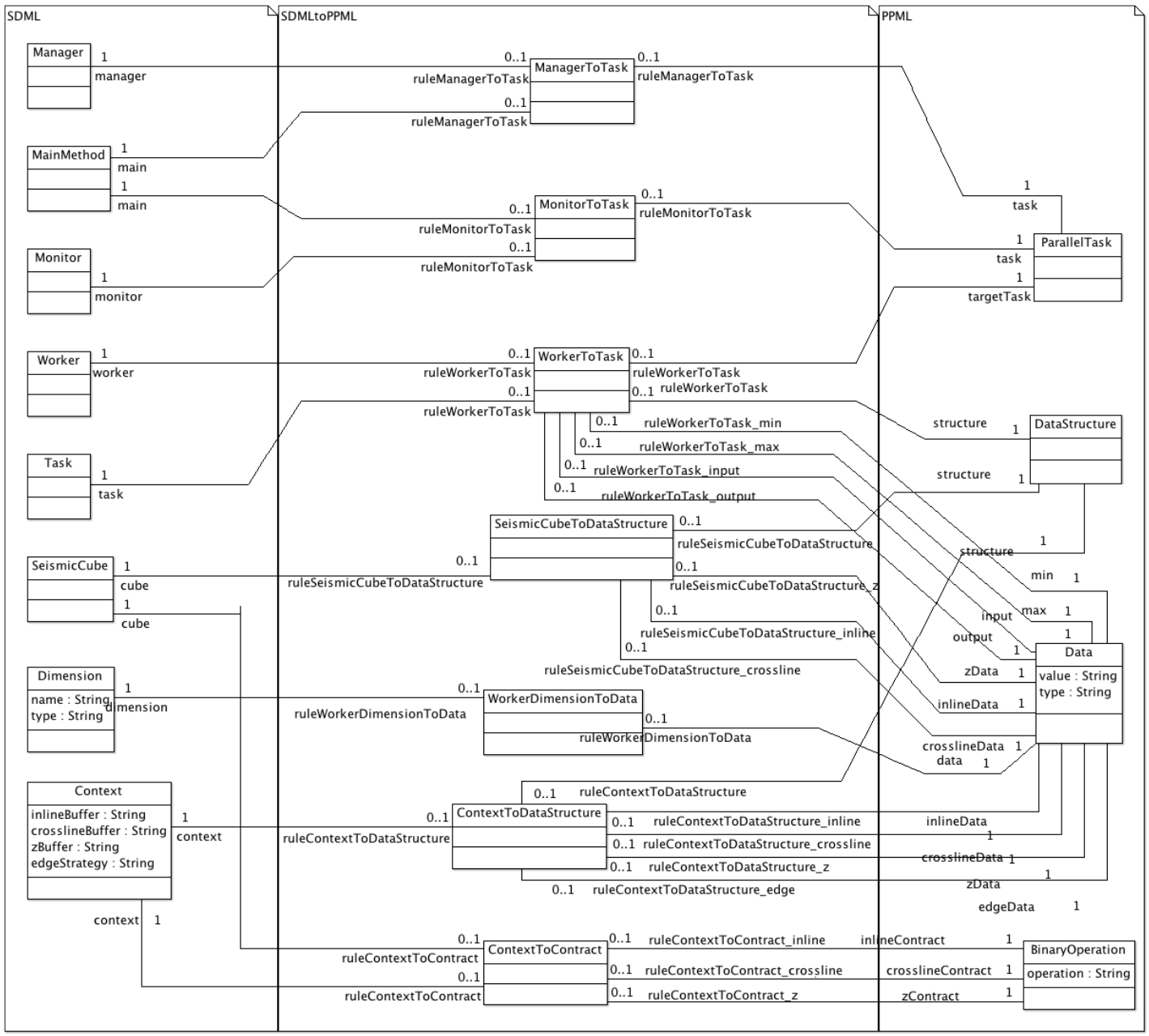


Figura 4.5: Modelo *SDMLtoPPML* utilizado

relacionada ao contexto no metamodelo de origem. Logo, é necessário criar instâncias de *Data* para os atributos *inlineBuffer*, *crosslineBuffer*, *zBuffer* e *edgeStrategy*. Por fim, a estrutura de dados referente a *ContextToDataStructure* depende da estrutura resultante da transformação do cubo sísmico, no qual o contexto atua no metamodelo de origem.

- *ContextToContract*: Define a relação entre contextos e cubos sísmicos de entrada, relacionados indiretamente pela instância de *Manager*, com três contratos do tipo *BinaryOperation*, sendo um para cada dimensão do cubo. O propósito desta regra é garantir que um contexto nunca será maior ou igual ao tamanho dos cubos que atuam.

Além dos invariantes sobre as cardinalidades dos modelos, as propriedades a seguir visam garantir que a estrutura imposta pelo metamodelo de origem é preservada no metamodelo de destino, i.e., as relações entre os elementos como instâncias de SDML devem ser respeitadas após a sua transformação, como instâncias de PPML[7].

O primeiro invariante verifica se a estrutura referente a instâncias de *Manager*, seus cubos sísmicos de entrada e seu contexto são mantidos após a transformação. Para tal, as seguintes afirmações devem ser válidas: (i) as estruturas de dados referentes aos cubos sísmicos de entrada de um gerente devem ser parâmetros da respectiva tarefa paralela; (ii) o contexto e cada cubo sísmico de entrada, relacionados por um *Manager*, tem uma relação de dependência quando transformados em instâncias de *DataStructure*, cuja estrutura gerada pelo contexto é a dependente; (iii) *DataStructure* referente ao contexto deve ser um parâmetro da instância de *ParallelTask* associada ao respectivo *Manager*. A seguir é apresentado a propriedade escrita em OCL.

```
context ManagerToTask inv IntegrityConstraintManagerContextCube:
self.manager.context_->size() > 0 implies (self.manager.input.ruleSeismicCubeToDataStructure.structure->forAll(
    cubeStructure | self.manager.context_.ruleContextToDataStructure.structure.dependsFrom->exists(dependsCube |
    cubeStructure = dependsCube) and self.manager.context_.ruleContextToDataStructure.structure.params_ = self.task
    and cubeStructure.params_ = self.task))
```

A outra propriedade refere-se à preservação da estrutura interna de cubos sísmicos de entrada de um *Manager*. Para tal, é necessário garantir que cada uma das dimensões, transformadas em *Data*, pertençam a estrutura de dados gerada do cubo sísmico em questão. Por fim, esta instância de *DataStructure* deve ser um parâmetro da instância de *ParallelTask*, referente ao *Manager* que gerencia o processamento do cubo. Esta

[7] Apesar de existir maneiras mais simples de definir os invariantes do modelo de transformação sem mudar seu sentido, eles são executáveis na ferramenta *EOS*.

propriedade é definida abaixo:

```
context ManagerToTask inv verifyCubeIntegrityInManagerToTask:
self.task.params->select(ds | not ds.ruleSeismicCubeToDataStructure->isEmpty())->forAll(ds | ds.data->select(dt | not
    dt.ruleSeismicCubeToDataStructure_inline->isEmpty())->forAll(dt | self.manager.input->includes(dt.
    ruleSeismicCubeToDataStructure_inline.cube)) and ds.data->select(dt | not dt.
    ruleSeismicCubeToDataStructure_crossline->isEmpty())->forAll(dt | self.manager.input->includes(dt.
    ruleSeismicCubeToDataStructure_crossline.cube)) and ds.data->select(dt | not dt.ruleSeismicCubeToDataStructure_z
    ->isEmpty())->forAll(dt | self.manager.input->includes(dt.ruleSeismicCubeToDataStructure_z.cube)))
```

O conceito usado por estes invariantes pode ser aplicado para preservar outras estruturas como, por exemplo, a relação entre instâncias de *Manager*, *Worker* e *Monitor*. Este contrato foi criado como exemplo da técnica no contexto dos projetos da *Schlumberger*. O contrato não é completo e falta ainda uma demonstração formal de que todas as propriedades de SDML são preservadas por PPML.

O Apêndice B contêm todas as propriedades, e suas explicações, referentes ao modelo de transformação.

Então, com o modelo de transformação e as propriedades, além de um verificador de consistência, é obtido a definição do contrato de transformação utilizado no transformador de SDML para PPML.

4.3 Protótipo do transformador

Para finalizar o projeto, foi proposto um protótipo utilizando as especificações definidas previamente. A arquitetura da ferramenta utiliza o padrão de projeto apresentado no Capítulo 2. Com isto, foi aplicado a técnica de contratos de transformação no protótipo. A Figura 4.6 apresenta o diagrama de classe UML referente ao seu uso.

A classe *Domain* foi estendida para representar os metamodelos SDML (*SDMLDomain*) e PPML (*PPMLDomain*). O modelo de transformação é definido pela classe *SDMLtoPPMLDomain*, que estende *JoinedDomain* e relaciona *SDMLDomain* como domínio de origem e *PPMLDomain* como de destino. As propriedades de cada metamodelo foram representadas por extensões da classe *InvariantValidator*, sendo SDMLtoPPMLInvariantValidador, SDMLInvariantValidator e *PPMLInvariantValidator* para, respectivamente, o modelo de transformação e os metamodelos SDML e PPML. Por fim, os verificadores de consistência são definidos pelas classes *SDMLtoPPMLConsistencyValidator*, *SDMLConsistencyValidator* e *PPMLConsistencyValidator*. Semelhante ao `UMLtoEJB`, as classes *InvariantValidator* e *ConsistencyValidator* representam as propriedades desejáveis

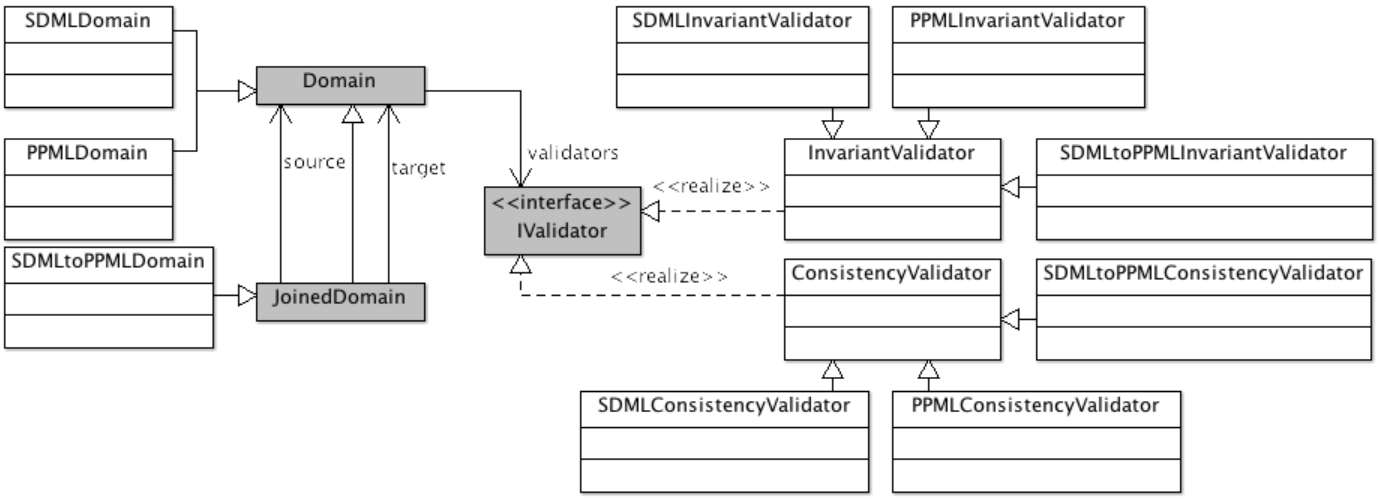


Figura 4.6: Diagrama de classe UML do transformador `SDMLtoPPML`, estendendo a arquitetura genérica

para este transformador, sendo também extensões necessárias da interface *IValidator* e do `TCLib`.

Em *SDMLDomain*, foi implementado o método de *parsing*, para carregar um arquivo XMI, contendo um diagrama de classe UML utilizando os estereótipos definidos na Seção 4.1.1. Em *PPMLDomain*, foi implementado o método de *pretty-printing* que, neste caso, consiste em código-fonte em C#, utilizando classes para simular o framework *Ocean/Petrel*[8], conforme apresentado na Seção 4.1.2.

SDMLtoPPMLDomain implementa em Java a transformação descrita na Seção 4.2. O mesmo tratamento da ordem referente às regras de transformação utilizado no transformador `UMLtoEJB` foi utilizado no protótipo, i.e., primeiro todas as instâncias das metaclasses foram construídas e depois ligadas por instâncias dos relacionamentos definidos pelo modelo de transformação.

A interface gráfica do protótipo foi feita utilizando-se a ferramenta *GenericPad*, como no transformador apresentado na Seção 3.3. A Figura 4.7 apresenta o transformador com o resultado de um comando OCL e exibindo a instância de metamodelo SDML.

4.3.1 Exemplos utilizados

Para a execução do transformador, um modelo de exemplo foi extraído de [48, p. 222], conforme apresentado na Figura 4.2. O objetivo deste modelo é testar o metamodelo

[8]Neste projeto, devido a restrições referente à licença do framework *Ocean/Petrel*, não foi possível utilizá-lo para uso com o resultado do *pretty-printing*.

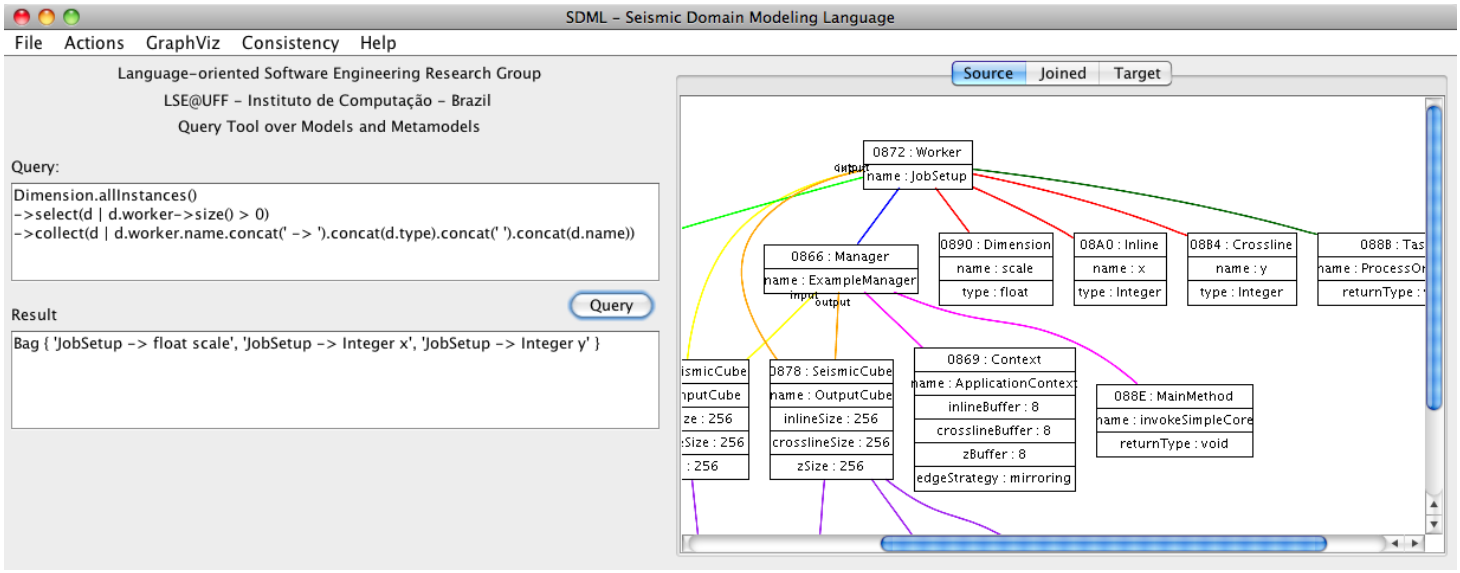


Figura 4.7: Protótipo do transformador de SDML para PPML

SDML, representando um caso real, comparando-o com o código-fonte no qual o exemplo foi baseado.

Para gerenciar o processamento do cubo sísmico, chamado *InputCube*, foi definida uma classe com o estereótipo *«Manager»*, cujo nome é *ExampleManager*, e possui o método *invokeSimpleCore* para representar a sua execução. Seu objetivo é aumentar ou reduzir a escala do cubo de entrada, que será persistido no cubo sísmico de saída chamado *OutputCube*.

O processamento dos cubos sísmicos serão realizados por objetos do tipo *JobSetup*, cujo fator de escala está definido pelo atributo *scale* e operará nas dimensões *inline* e *crossline*. Por fim, a classe *MyMonitor* é responsável por monitorar todo o processo dos trabalhadores.

Informações sobre o contexto e os cubos sísmicos estão informadas em comentários estruturados, devido a limitações na ferramenta de modelagem utilizada (*Tagged values*, serão utilizados em versões futuras da ferramenta.).

4.3.2 Benefícios do uso do contrato na construção e utilização do protótipo

A transformação de SDML para PPML obteve benefícios semelhantes aos obtidos pelo transformador `UMLtoEJB`, conforme apresentados na Seção 3.3.2. A verificação do contrato de transformação especificado permitiu a detecção de erros tanto durante a construção

quanto no uso posterior do transformador.

Como designer da transformação, o invariante *IntegrityConstraintManagerContextCube* permitiu detectar a ausência da hierarquia de dependência entre cubos sísmicos e contextos, evitando a geração de uma instância do modelo de transformação que não está em conformidade com seu modelo.

De acordo com a visão de um usuário do transformador, os invariantes *ManagerMustHaveAOutputcube* e *MustHaveATask*[9] auxiliaram a identificar a ausência de um cubo sísmico de saída em uma instância de *Manager* e um trabalhador sem uma instância de *Task*, ambos erros causados porque os modelos de entrada não eram bem-formados em relação ao metamodelo SDML.

```
context Worker inv MustHaveATask:
self.task->size() > 0
```

Para o designer da linguagem de modelagem, o invariante *MainMethodMustBelongToMonitorOrManager* permitiu detectar que o processo de *parsing* não estava carregando adequadamente as relações de *MainMethod* e *Monitor*, apesar do modelo de entrada ser válido.

O uso de verificador de consistência permitiu identificar quando um modelo não era instanciável, conforme apresentado no Capítulo 2. Para este transformador, foi detectado uma inconsistência quando o modelo de entrada possuía duas instâncias de *Task* atribuídas a uma instância de *Worker*. A Figura 4.8 apresenta quando uma inconsistência é encontrada, com uma mensagem sobre o erro referente ao resultado do raciocinador de lógica de descrição utilizado.

Para a execução do transformador, foi utilizado o exemplo apresentado na Seção 4.3.1 e variações com erros propositais, cuja intenção é garantir que o contrato de transformação é cumprido corretamente.

Após a geração do artefato final, ele foi comparado ao código-fonte que inspirou o modelo de entrada. Esta análise permitiu concluir que o resultado da transformação atende as especificações estruturais referentes ao exemplo, disponíveis em [48, p.222].

O código-fonte gerado a partir de instâncias da metaclasse *Contract* permitiu a validação em tempo de execução da aplicação, semelhante a Desenho por Contrato. No

[9] O invariante *MustHaveATask* não foi listado na Seção 4.1.1 porque trate-se de uma restrição de cardinalidade, que deveria ser avisada pelo interpretador OCL. Devido às limitações do interpretador OCL utilizado, foi necessária a criação de tais invariantes, mas não foram listados por não serem uma contribuição ao trabalho.

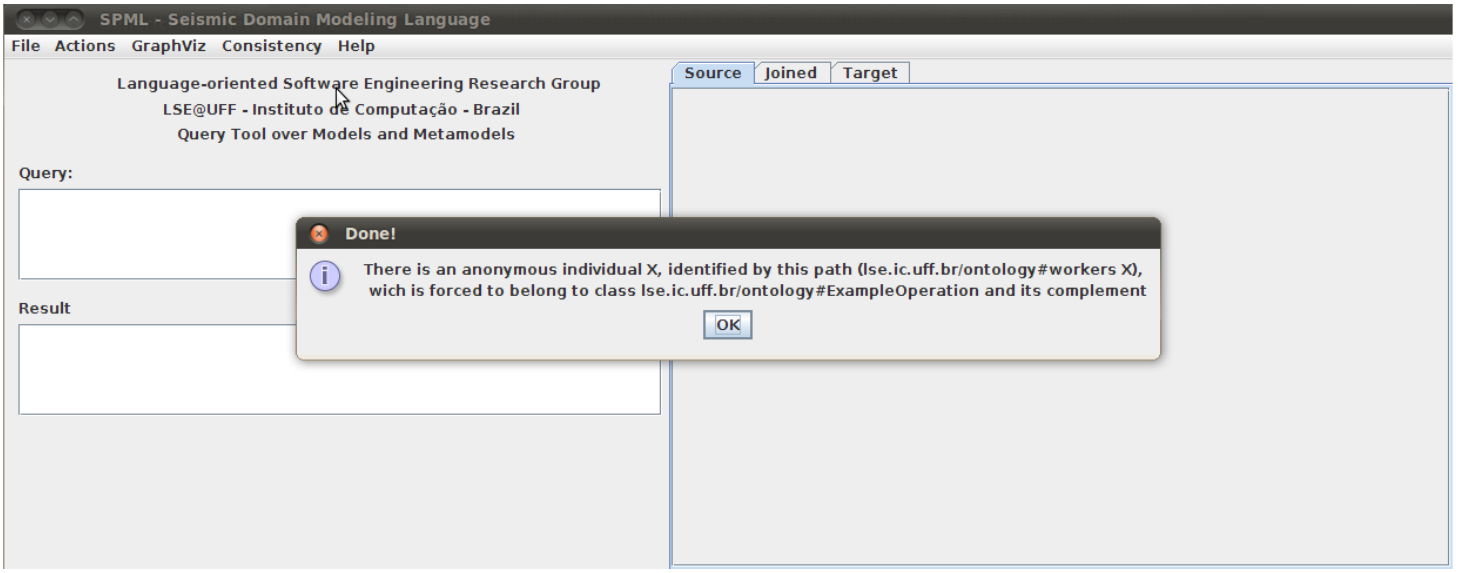


Figura 4.8: Detecção de inconsistência em um modelo SDML

exemplo utilizado, instâncias de *BinaryOperation* resultaram em restrições ao modificar as dimensões de um contexto, que devem ser menores que a do cubo sísmico que atua. Tal abordagem foi elogiada pelos pesquisadores do BRGC, por auxiliar o desenvolvedor a atender as restrições impostas pela abordagem, de forma transparente, sem interferir na modelagem da aplicação.

4.4 Considerações finais

Este capítulo descreveu uma proposta de transformação de modelo com contratos de transformação para o desenvolvimento de aplicações sísmicas. O projeto é uma iniciativa para suportar desenvolvimento rigoroso de aplicações sísmicas para a plataforma *Ocean/Petrel* da *Schlumberger*. Logo, foram propostos metamodelos adequados e uma especificação de contrato de transformação, o que permitiu a construção de um protótipo, para exemplificar a abordagem.

O protótipo permitiu aos pesquisadores da *Schlumberger* entender os benefícios de DDM, com etapas para validação e verificação no processo de transformação, e como aplicá-lo na plataforma *Ocean/Petrel*, apesar de tratar um subconjunto restrito referente a processamento de cubos sísmicos.

O metamodelo SDML representa aspectos estruturais de uma aplicação sísmica. Trabalhos futuros envolvem aumentar a sua expressividade para suportar especificações comportamentais, com o uso de máquinas de estado, e a sua validação, através de ferramentas

de *model checking* [35]. O metamodelo PPML também precisa ser expandido, para suportar mais tipos de contratos.

Por fim, novas técnicas de validação e casos reais para testes são fundamentais para a continuação do projeto e, eventualmente, acoplá-lo ao ambiente de desenvolvimento fornecido pela *Schlumberger*.

Capítulo 5

Trabalhos relacionados

Este capítulo apresenta os trabalhos relacionados a esta dissertação, que estão divididos em quatro seções: (i) especificação de transformações de modelo; (ii) verificação da transformação de modelo; (iii) implementação de transformações de modelo; (iv) comparação entre contratos de transformação e QVT, explicitando as diferenças entre tais abordagens. Por fim, é apresentado as considerações finais, explicitando a relação entre os trabalhos abordados e esta dissertação.

5.1 Especificação de transformações de modelo

O conceito de modelos de transformação é essencial nos contratos de transformação. Em [8], a ideia de modelo de transformação em vez de transformação de modelo é discutida. Os autores descrevem os benefícios da omissão de detalhes do processo de transformação e a concentração na especificação da transformação como um modelo. O principal benefício da modelagem de uma transformação de modelos como um modelo de transformação, que não fica muito claro em [8], é a uniformidade de especificação e verificação. Um modelo de transformação pode ser especificado como um diagrama de classes, por exemplo, e ser analisado sintática e semanticamente através das mesmas técnicas utilizadas para analisar um diagrama de classes. Portanto, não se fazem necessárias outras linguagens de especificação, como QVT [42], para especificar uma transformação, nem novas técnicas específicas para validação de transformações de modelos.

Trabalhos relacionados com contratos de transformação [9, 11, 23] concordam com os conceitos de [20, 29], embora hajam diferenças nos níveis de especificação, validação e implementação. Em [1, 8], diferente de [20, 29] que especificam o contrato de transformação como invariantes em OCL dos elementos dos modelos de entrada e saída, apresentam uma

abordagem relacional para a especificação do metamodelo de transformação. Em [20, 29] também é discutido o uso de pré e pós condições, que podem ser representadas como invariantes.

Além disso, [1, 29] consideram a sintaxe concreta de uma linguagem de modelagem como uma bijeção com a sintaxe abstrata descrita pelo metamodelo. O designer de transformação do modelo é beneficiado por ter um *parser* para qualquer linguagem de modelagem. No entanto, essa escolha é um incômodo para o usuário da transformação, pois é preciso criar um modelo bastante detalhado para que a máquina possa compreendê-lo adequadamente. O uso de uma linguagem de modelagem *específica de domínio* permite descrições no contexto do domínio e, se possível, concisas.

5.2 Verificação de transformações de modelo

A especificação de uma relação entre elementos do modelo de um metamodelo, relacionados por uma transformação de modelo, é essencialmente para entender como diferentes tipos de propriedades são especificadas sobre tais relações, generalizando de invariantes OCL. Para tal, é importante tornar explicita a relação entre os metamodelos. No nível de verificação, diferentes propriedades podem ser raciocinadas, independente do uso de invariantes em OCL. Uma dessas propriedades é a consistência de modelos proposta por [7], entendida como a satisfatibilidade de uma teoria em lógica de descrição associada com um dado modelo. Em [38] também é verificável a consistência de modelos porém através de lógica relacional e OCL, utilizando as ferramentas *Kodkod*[1] [56] e *USE*[2].

A corretude de uma transformação de modelos tem sido discutida por muitos autores. Em [3] o autor enfatiza que a especificação da transformação de modelo pode ser considerada um tipo especial de modelo e, como tal, pode ser sujeita a técnicas existentes de análise de modelo.

Em [19], a corretude de um modelo se torna uma questão importante, já que os defeitos do modelo vão, diretamente, se tornar defeitos de implementação no sistema de software final, devido à aplicação de técnicas de geração de código.

No entanto, linguagens de modelagem populares não são suficientemente formais para provar a corretude de seus modelos. Portanto, um conjunto de técnicas de verificação no

[1] *Kodkod* é uma ferramenta para solucionar problemas de satisfatibilidade (em inglês, *SAT solver*), disponível em `http://alloy.mit.edu/kodkod/`.

[2] *USE* é uma ferramenta para especificação de sistemas de informação baseada em um subconjunto da UML, disponível em `http://www.db.informatik.uni-bremen.de/projects/USE/`.

nível do modelo são necessários para garantir a qualidade de especificações do modelo de software.

Algumas abordagens transformam os modelos em outras linguagens e formalismos, em que a validação ocorre pelo uso de solucionadores (*solvers*, em inglês) ou provadores de teoremas.

Em [18], o Problema de Satisfação de Restrição (CSP, sigla em inglês) é utilizado para verificar modelos UML/OCL usando um método de tradução. Após a tradução, um solucionador de restrição (*constraint solver*, em inglês) é executado para verificar se um modelo UML, com restrições OCL, é satisfatível. A vantagem desta abordagem é a garantia de que o modelo e seus invariantes são bem definidos.

Em [44] é proposta uma verificação das transformações de modelo especificadas em uma linguagem que estende as relações QVT com técnicas de CSP. O uso de técnicas CSP é semelhante ao uso de lógica de descrição, conforme apresentado na Seção 2.3.1, porém sua abordagem é presa ao QVT.

Em [17] é proposta uma abordagem para validação de transformações em ATL por meio da geração de um modelo de transformação, que é, nesse caso, a combinação da sintaxe dos modelos de origem e destino com a semântica de execução da uma transformação em um modelo estrutural integrado. Esses modelos de transformação podem ser utilizados para análise em ferramentas de verificação de consistência e validar certas propriedades, como se a transformação é total através da verificação sistemática por violações das restrições do metamodelo de destino.

Em [50, 26] são utilizadas técnicas de geração de casos de teste para a transformação de modelos. Com a criação de todos os cenários de teste possíveis e transformando-os, é possível validar o comportamento da transformação.

Em [5], contratos de transformação são apresentados como uma das técnicas para geração de casos de teste utilizadas na validação de transformações. Para verificar a consistência da transformação, ele descreve a criação de um novo metamodelo adicionando um elemento para conectar os elementos dos metamodelos de origem e destino. Sua abordagem consiste em gerar casos de teste. Em [9, 11, 23] também visam ganhar confiança nas transformações de modelo. Porém, diferentemente da abordagem em [5], é preferível ver a especificação de uma transformação de modelo como um metamodelo, sobre o qual são aplicadas as mesmas técnicas utilizadas para validar um modelo, sem a necessidade de técnicas adicionais para a verificação de transformações de modelo.

Em [20] contratos de transformação são verificados em três etapas: (i) no modelo de origem; (ii) no modelo de destino; (iii) no modelo de transformação. Para a transformação, apresentam-se dois métodos possíveis para verificar a consistência através de contratos de transformação. No primeiro método, os elementos do modelo de origem são verificados como pré-condições e a transformação é verificada com pós-condições, ao ligar os elementos do modelo de origem com o construtor **@pre** da OCL. Esse construtor permite que os elementos do modelo de destino sejam comparados com os elementos do modelo de origem. Nesse método, a operação de transformação é descrita em OCL e os metamodelos de origem e destino são os mesmos ou muito semelhantes. O segundo método consiste na criação de dois pacotes. O primeiro pacote contém os elementos do modelo de origem e o segundo contém elementos do modelo de destino. A operação de transformação também é definida em OCL, tendo um pacote para o modelo de origem como entrada e resultando em um pacote do modelo de destino. Nesse caso, ambos os elementos dos modelos de origem e destino podem ser referenciados na operação de transformação. O primeiro método apresentado requer que os metamodelos de origem e destino sejam idênticos, enquanto no segundo método é necessário criar um novo operador OCL para mapear os elementos dos modelos de origem e destino. Na abordagem seguida neste trabalho, nenhuma destas duas restrições se aplica, ou seja, não há restrições sobre os metamodelos usados. Mais ainda, o uso de OCL é uma possibilidade para especificação de propriedades sobre os modelos. Outras propriedades como consistência, discutida na Seção 2.3.1, ou fórmulas em lógica temporal também podem ser usadas nas etapas de validação.

5.3 Implementação de transformações de modelo

Dada a generalidade dos contratos de transformação [9, 11, 23], a implementação de transformadores não está confinada a linguagens baseadas em OCL como QVT. QVT é uma possibilidade, que permite a especificação da relação entre os metamodelos relacionados pela transformação de modelo, apesar de estar restrito ao OCL para a especificação das propriedades dos metamodelos. Porém, certas propriedades são melhor especificadas em outras linguagens, tais como consistência em lógica de descrição, conforme discutido na Seção 2.3. QVT é uma forma de descrição do metamodelo de transformação, e não a única. Na Seção 5.4 é abordada uma comparação entre o conceito de contratos de transformação com QVT, explicitando as diferenças sobre tais abordagens.

Uma das aplicações de contratos de transformação, apresentada no Capítulo 3, consiste em uma transformação de modelos envolvendo componentes. Em [37] é apresentado

todo o processo de transformação de modelos seguindo a especificação chamada *Model-driven Architecture* (MDA). Para exemplificar um transformador, é apresentada uma transformação de um diagrama de classe UML para código-fonte em EJB, que é uma arquitetura de software para sistemas baseados em componentes.

De forma semelhante, em [47] é discutida a combinação de abordagens baseadas em componentes e em modelos, seguindo a especificação MDA, para obter adaptabilidade no middleware. Os modelos são transformados em componentes executáveis, com as configurações necessárias para o seu uso em um processo de compilação adaptável, para o framework baseado em componentes chamada GoTM[3], desenvolvido pelo mesmo grupo de pesquisa.

Em outra abordagem, foi desenvolvida uma ferramenta chamada *Genie* [6], que suporta a modelagem, geração e operação de sistemas baseados em componentes altamente reconfiguráveis. Essa ferramenta utiliza duas linguagens específicas de domínio, uma para especificação de variabilidade estrutural, chamada *OpenCOM DSL*, e outra para a variabilidade de ambiente e contexto, chamada de *Transition Diagrams DLS*. Juntamente com modelos de variabilidade ortogonal [45], também conhecidos como modelos de transição, são gerados os componentes, com suas configurações e políticas de reconfiguração, para o *Gridkit* [30], um middleware reflexivo baseado em componentes para grades computacionais. *Genie* utiliza de validação dos modelos baseada em restrições para garantir a consistência dos artefatos gerados. Nesse caso, são utilizadas restrições sobre os modelos relacionados com a variabilidade estrutural para garantir a consistência dos componentes gerados e sobre os modelos de transição para evitar inconsistência nas políticas de reconfiguração.

Em [31] é proposta uma transformação de modelos para melhorar a análise de sistemas baseados em componentes. A linguagem *KLAPER* (*Kernel LAnguage for PErformace and Reability analysis*) é utilizada como uma representação intermediária entre as fases de design e análise. Dessa forma, os modelos utilizados na etapa de análise são gerados a partir dos modelos criados durante a etapa de design do sistema. Como entrada de modelos, é utilizada a linguagem UML, para modelar a estrutura dos componentes, e OWL-S, para representar o comportamento interno dos componentes. Para auxiliar no processo de transformação são utilizadas as especificações da OMG chamadas MOF e QVT.

[3]GoTM, que significa *GoTM is an open Transaction Monitor*, está disponível em `http://gotm.objectweb.org/`.

Em relação à transformação de modelos do domínio sísmico, este trabalho mostrou-se bastante inovador nesta área, uma vez que o objetivo do projeto foi permitir a especificação de processamento paralelo de dados sísmicos. O padrão ISO 15926-2 [32] descreve um metamodelo para a automação de projetos de engenharia, permitindo a representação dos modelos sísmicos. Porém, o ISO 15926-2 é um metamodelo muito geral, dificultando o seu uso prático. Portanto, foi proposta uma linguagem de modelagem específica de domínio. O mesmo aconteceu em relação à linguagem de destino da transformação.

5.4 Comparação entre Contratos de transformação e QVT

Query-View-Transformation, ou QVT, é uma especificação da OMG para realizar transformações de modelo, que é composta por três linguagens: (i) *QVT-Operational*, uma linguagem imperativa para definir transformações unidirecionais; (ii) *QVT-Relations*, uma linguagem declarativa que permite especificar transformações unidirecionais e bidirecionais, tanto de forma gráfica quanto textual; (iii) *QVT-Core*, uma linguagem declarativa simples para ser destino da tradução de *QVT-Relations*. QVT também possui um padrão para a transformação, que deve atender às pré-condições de uma regra antes de sua aplicação e as pós-condições devem ser válidas após a transformação.

Apesar de serem de níveis de abstração diferentes, é esperado que uma comparação entre contratos de transformação e QVT devido o contexto do trabalho. Neste trabalho será considerado o *QVT-Relations* nas comparações, por ser mais abrangente das três linguagens e por ser declarativa, possuindo um nível de abstração mais próximo de contratos de transformação.

Então, ao comparar contratos de transformação com QVT, os seguintes aspectos devem ser considerados:

- QVT é uma linguagem para transformação de modelos, enquanto contratos de transformação é uma abordagem que pode ser implementada em qualquer linguagem, tal como QVT ou mesmo Java, desde que as propriedades de todos os domínios envolvidos sejam mantidas.

- A abordagem de contrato de transformação sugere uma forma abstrata para especificar a transformação de modelo através da relação entre os domínios envolvidos. A descrição em QVT, para a transformação de modelos, pode ser interpretada como

uma possível realização de tal especificação. Quando uma descrição em QVT não é baseada em um contrato de transformação, o designer da transformação precisa lembrar-se de escrever todas as propriedades dos domínio de origem e destino, tais como as propriedades de transformação.

- QVT somente permite a especificação de propriedades OCL referente a pré e a pós-condições das regras de transformação, enquanto contratos de transformação abstraem desta limitação e enfocam que todas as propriedades do domínio devem ser mantidas, o que pode ser descrito em qualquer formalismo adequado, e não somente OCL. Diversas propriedades podem ser melhor descritas e verificadas em diferentes formalismos.

- Domínios são um aspecto essencial em contratos de transformação e não explicitamente suportados em QVT. Isto ocorre, porque QVT possui enfoque na transformação em si, enquanto contratos de transformação possuem foco na corretude e conformidade dos modelos, que também implica na transformação. Por instância, a propriedade, que especifica a ausência de ciclos em hierarquias de classe, não pertence a transformação, mas sim ao domínio UML, o que seria facilmente ignorada em uma descrição em QVT.

- Domínios também ajudam no design de uma transformação do modelo, que deve ser dirigida pela seguinte sintaxe: cada elemento do modelo em um domínio deve ser entendida como uma construção da linguagem, com sua semântica própria. Como exemplo, em [37] a associação de composição não é tratada corretamente porque nenhuma imposição à integridade dos dados é produzida na representação SQL de um diagrama de classes UML. Se a transformação do modelo fosse descrita dirigida pela sintaxe, talvez o designer de transformação a teria definido-a corretamente.

5.5 Considerações finais

Esta dissertação utiliza o conceito de contratos de transformação, apresentado em [9, 11, 23], conforme abordado na Seção 2.3. Também foi adotada uma abordagem relacional para a especificação do metamodelo de transformação semelhante ao proposto em [1, 8]. Além disso, neste trabalho é generalizado o entendimento das sintaxes concretas e abstratas discutidas em [1, 29], considerando a sintaxe concreta de uma linguagem como uma bijeção com a sintaxe abstrata descrita pelo metamodelo.

Em relação a propriedades do metamodelo, além dos invariantes em OCL já utilizados com CT, é apresentada uma verificação de consistência de modelos semelhante ao proposto por [7], conforme apresentada na Seção 2.3.1.2. Porém, as técnicas propostas por [50, 26] não foram consideradas nesta dissertação porque os validadores desenvolvidos validam o cenário utilizado durante a transformação, analisado em tempo de execução.

Em relação a implementação dos contratos de transformação, o estudo de caso apresentado no Capítulo 3 possui os metamodelos e regras de transformação baseados de [37]. Já o Capítulo 4 não considerou [32] por ser muito genérico, o que não é a proposta do projeto em questão.

Apesar de serem propostas diferentes, é notável que QVT poderia se beneficiar de uma formalização mais rígida e de mais mecanismos para validação. O uso de contratos de transformação aparentemente reduziria esta deficiência, apesar de não ser explorado neste trabalho, já que as transformações são implementadas em Java, conforme será abordado nos Capítulos 3 e 4.

Capítulo 6

Conclusão

6.1 Considerações finais

Este trabalho apresentou os avanços de uma abordagem de desenvolvimento dirigido a modelo conhecida como contratos de transformação. Foi incorporado ao processo de transformação um verificador de consistência de modelos, aprimorando a validação da transformação, e a técnica foi aplicada em dois transformadores de modelos de contextos diferentes.

Contratos de transformação é uma abordagem formal de DDM para a especificação, verificação e desenvolvimento de transformadores, que considera modelos como linguagens. Portanto, devido a formalização do processo, é possível utilizar métodos formais para validação dos modelos.

Durante a execução de um contrato, a transformação é validada com assertivas sobre o modelo de transformação (ver Seção 2), garantindo que a relação entre os modelos de origem e destino atendem as propriedades que definem a sua semântica. Uma contribuição deste trabalho é o aprimoramento desta etapa de validação, ao acoplar um verificador de consistência ao contrato de transformação e formalizar o processo de transformação com esta técnica. Assim, as propriedades referentes a semântica somente são aplicadas se o resultado da transformação é consistente, i.e., instanciável.

Estudos já realizados com os contratos de transformação estão relacionados com o domínio de segurança dirigida a modelo. A segunda contribuição realizada por este trabalho é a especificação e implementação de dois transformadores para, respectivamente, modelagem de sistemas distribuídos e para processamento de dados sísmicos, com a geração de código-fonte adequada a cada contexto.

O transformador `UMLtoEJB` realiza transformação entre diagramas de classe UML para código-fonte em EJB. Assim, UML e EJB são consideradas como linguagens e é aplicado um contrato de transformação específico para tais domínios. Por fim, o uso de um contrato auxiliou a detecção de inúmeros problemas de implementação, conforme apresentado na Seção 3.3.2, e aumentou a confiança sobre o resultado da transformação.

`SDMLtoPPML` é um transformador para o processamento de dados sísmicos. Em uma parceria com a BRGC, foi definida uma linguagem chamada SDML para modelar uma estrutura para processamento de cubos sísmicos. Juntamente com PPML, uma linguagem simples para modelar processamento de dados em paralelo, foi realizada uma transformação utilizando contratos de transformação, o que permitiu atestar sua aplicabilidade em um problema real e crítico, pelo impacto financeiro causado por um erro, para empresas petrolíferas. Tanto o uso de invariantes quanto da verificação de consistência foram essenciais para a detecção de erros de implementação e, inclusive, demonstrar que a transformação realmente foi realizada de forma esperada.

Portanto, foi atestado que o uso de contratos de transformação auxiliou na concepção dos transformadores, para determinar o que e como deveria ser transformado, e para detectar problemas de implementação, encontrados pela aplicação das técnicas de validação utilizadas.

Além disto, a técnica mostrou-se aplicável em diversos contextos sem a necessidade de qualquer adaptação, garantindo que não se trata de uma abordagem exclusiva do domínio de segurança dirigida a modelo ou de linguagens específicas de domínio.

As etapas de validação permitiram detectar erros em diferentes etapas de desenvolvimento, como durante a concepção do processo de *parsing*, e tipos de usuários, como do designer da linguagem de modelagem. Assim, esta etapa mostrou-se fundamental no desenvolvimento dirigido a modelo por garantir a qualidade da transformação e, consequentemente, do artefato final gerado.

A intenção deste trabalho foi contribuir com a técnica de contratos de transformação. O ferramental escolhido utiliza especificações defasadas como, por exemplo, o *ArgoUML*, que se baseia em uma especificação antigo da UML. Esta decisão foi tomada porque foram escolhidas ferramentas simples já conhecidas e utilizadas pelo grupo de pesquisa. Assim, foi possível concentrar-se nas contribuições do trabalho em vez de aprender novas ferramentas.

6.2 Trabalhos futuros

Devido à importância da etapa de validação, estudos referentes a novas formas de validação e verificação são importantes e necessárias para aumentar ainda mais a confiança sobre a transformação e a qualidade do artefato final. Possíveis caminhos envolvem o uso de lógica de rescrita para verificação de confluência através do teorema de Church-Rosser e de terminação da transformação.

Até o momento, a transformação de elementos é implementada manualmente, com o auxilio de um interpretador OCL, resultando em uma etapa de implementação complexa, lenta e suscetível a erros. O uso de linguagens específicas para transformação, e.g., QVT e ATL, com contratos de transformação, um ponto ainda inexplorado, pode reduzir os problemas abordados, além de demonstrar que CT independe de uma determinada tecnologia.

Uma característica importante porém pouco explorada é a transformação *many-to-many*, i.e., uma transformação que possui diversas linguagens de origem e de destino. A especificação de contratos de transformação ainda não permite tal abordagem, requerendo mais estudos para suportá-la. Uma possível aplicação desta característica seria em uma expansão do `UMLtoEJB`, gerando as informações necessárias para criação de um banco de dados, com SQL, com base no diagrama de classe UML de entrada e permitir que diagramas relacionados com a segurança da aplicação, e.g., diagrama de classe em SecureUML, sejam utilizados na transformação. Assim, seriam dois domínios de entrada (UML e SecureUML) e três domínios de saída (SQL, EJB e um para a segurança).

Outra característica que ainda requer estudos é a composição de transformações, aplicada com contratos de transformação, permitindo acoplá-las em sequência para alcançar um determinado resultado, reaproveitando transformações já existentes. `SDMLtoPPML` seria beneficiada com essa característica em seus próximos passos de pesquisa, ao definir um metamodelo específico para o framework *Ocean/Petrel*, com todas as suas idiossincrasias, e utilizá-lo como o destino de uma transformação a partir do PPML. Assim, aproveitaria a transformação já existentes de SDML para PPML.

`UMLtoEJB` também pode utilizar a mesma abordagem. Conforme apresentado em [37], o transformador possui um metamodelo refinado de EJB, apesar de ser apenas citado, e é um passo adicional para a transformação de UML para código-fonte em EJB. Este metamodelo refinado possui mais características referente a implementação do framework, tornando a geração de código-fonte mais simples.

Até o momento, apenas aspectos estáticos foram explorados nas transformações envolvendo contratos de transformação. Estudos referentes a aspectos dinâmicos e CT são necessários, o que inclui a implementação de técnicas de validação adequadas. Um exemplo de metamodelo para este aspecto seria a extensão do metamodelo UML para suportar diagramas de estado. Para tal, o uso de formulas temporais é mais adequado para a validação do que as técnicas abordadas até então. O transformador `SDMLtoPPML` também seria beneficiado com tal abordagem, permitindo modelar o comportamento de trabalhadores e, portanto, aprimorar o resultado da geração de código-fonte.

Além das características já citadas para aprimorar o transformador `UMLtoEJB`, mais técnicas de validação poderiam auxiliar na detecção de erros. A geração de código-fonte ainda é simples, pois não é suficiente para a execução adequada dos componentes. A adição da persistência de dados gerada automaticamente e criação de todos os arquivos de configuração de um projeto ainda são características importantes que não foram implementadas. Por fim, uma nova versão do metamodelo, representando a versão mais atual do EJB, é desejável para o uso do transformador em projetos reais, já que a versão 2.1, utilizada neste trabalho, não possui mais suporte pela empresa detentora do framework.

Para o `SDMLtoPPML`, o aumento da expressividade de ambos os metamodelos, SDML e PPML, poderia melhorar a sua utilidade e a qualidade do código-fonte gerado. Até o momento, não foi utilizado o resultado da transformação com o framework *Ocean/Petrel* devido a problemas com a licença, o que dificulta a análise do resultado e é fundamental para o avanço da pesquisa. Por fim, o uso de mais casos de exemplos, assim como a comparação com um caso real, são necessários para o aprimoramento do transformador e, consequentemente, direcioná-lo para um eventual uso junto às ferramentas fornecidas pela Schlumberger.

Referências

[1] Akehurst, D. H.; Kent, S. A relational approach to defining transformations in a metamodel. In *Proc. of the 5th Int. Conf. on UML* (London, UK, 2002), UML '02, Springer-Verlag, pp. 243–258.

[2] Alur, D.; Malks, D.; Crupi, J. *Core J2EE Patterns: Best Practices and Design Strategies*. Pearson Education, Upper Saddle River, NJ, USA, June 2001.

[3] Anastasakis, K.; Bordbar, B.; Küster, J. Analysis of model transformations via Alloy. In *Proceedings of the workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2007), Nashville, TN (USA)* (October 2007), A. B. Baudry, S. Faivre, and A. P. Ghosh, Eds., Springer, pp. 47–56.

[4] Baader, F.; Diego Calvanese, D. M.; Nardi, D.; Patel-Schneider, P. *The Description Logic Handbook*. Cambridge University Press, 2003.

[5] Baudry, B.; Dinh-Trong, T.; Mottu, J.-M.; Simmonds, D.; France, R.; Ghosh, S.; Fleurey, F.; Traon, Y. L. Model transformation testing challenges. *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing* (July 2006).

[6] Bencomo, N.; Grace, P.; Flores, C.; Hughes, D.; Blair, G. Genie: supporting the model driven development of reflective, component-based adaptive systems. In *Proc. of ICSE'08* (USA, 2008), ICSE '08, ACM, pp. 811–814.

[7] Berardi, D.; Calvanese, D.; Giacomo, G. D. Reasoning on UML class diagrams. *Artificial Intelligence 168* (October 2005), 70–118.

[8] Bézivin, J.; Büttner, F.; Gogolla, M.; Jouault, F.; Kurtev, I.; Lindow, A. Model transformations? transformation models! In *Proc. of MoDELS 2006, Italy, October 1-6* (2006), vol. 4199, Springer, pp. 440–453.

[9] Braga, C. From access control policies to an aspect-based infrastructure: A metamodel-based approach. In *Models in Software Engineering* (Berlin Heidelberg, 2009), vol. 5421 of *Lecture Notes in Computer Science*, Springer, pp. 243–256.

[10] Braga, C. Model-driven development from a programming language perspective. In *1st. Brazilian Workshop on Model-Driven Development* (Salvador, September 2010), pp. 69–76.

[11] Braga, C. A transformation contract to generate aspects from access control policies. *Journal of Software and Systems Modeling* (Springer, 2010). DOI: 10.1007/s10270-010-0156-x.

[12] Braga, C.; Hæusler, E. H. Lightweight analysis of access control models with description logic. *Innovations in Systems and Software Engineering 6* (2010), 115–123.

[13] Braga, C.; Menezes, R.; Comicio, T.; Santos, C.; Almeida, E. Transformation contracts in practice (aceito para publicaÃ§Ã£o). IET Software.

[14] Braga, C.; Menezes, R.; Comicio, T.; Santos, C.; Almeida, E. On the specification, verification and implementation of model transformations with transformation contracts. In *14th Brazilian Symposium on Formal Methods* (2011), vol. 7021 of *LNCS*, Springer, pp. 108–123.

[15] Braga, C.; Menezes, R.; Santos, C.; Track, A.; Iversen, T.; Silva, R. Towards model-driven development of seismic applications with transformation contracts. In *2nd Brazilian Workshop on Model-Driven Software Development* (September 2011).

[16] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., 1996.

[17] BÃ¼ttner, F.; Cabot, J.; Gogolla, M. On Validation of ATL Transformation Rules By Transformation Models. In *Proceedings of the workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2011), Wellington, TN (New Zealand)* (October 2011).

[18] Cabot, J.; Clarisó, R.; Riera, D. Verification of UML/OCL class diagrams using constraint programming. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop* (2008), IEEE Computer Society, pp. 73–80.

[19] Cabot, J.; Clarisó, R.; Riera, D. Verifying UML/OCL operation contracts. In *IFM* (2009), M. Leuschel and H. Wehrheim, Eds., vol. 5423 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 40–55.

[20] Cariou, E.; Marvie, R.; Seinturier, L.; Duchien, L. OCL for the specification of model transformation contracts. In *OCL and Model Driven Engineering, UML 2004 Conference Workshop, Portugal* (2004), pp. 69–83.

[21] Casanova, H.; Legrand, A.; Robert, Y. *Parallel Algorithms*. CRC Press, 2008.

[22] Clavel, M.; Egea, M.; de Dios, M. A. G. Building an Efficient Component for OCL Evaluation. *ECEASST 15* (2008).

[23] Comicio, T. A transformation contract approach for model-driven security. Master's thesis, Universidade Federal Fluminense, 2011.

[24] Czarnecki, K.; Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.

[25] Czarnecki, K.; Helsen, S. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture* (2003), pp. 1–17.

[26] Fiorentini, C.; Momigliano, A.; Ornaghi, M.; Poernomo, I. A constructive approach to testing model transformations. In *Proc. of ICMT'10* (2010), vol. 6142 of *LNCS*, Springer, pp. 77–92.

[27] Foster, I. Task parallelism and high-performance languages. *IEEE Parallel Distrib. Technol. 2* (September 1994), 27–36.

[28] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3 ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[29] Gorp, P. V.; Janssens, D. CAViT: a Consistency Maintenance Framework based on Transformation Contracts. In *Transformation Techniques in Soft. Eng.* (2006), no. 05161 in Dagstuhl Seminar Proc.

[30] Grace, P.; Coulson, G.; Blair, G. S.; Porter, B. Deep middleware for the divergent grid. In *Proc. of the ACM/IFIP/USENIX 2005 Int. Conf. on Middleware* (USA, 2005), Springer-Verlag, pp. 334–353.

[31] Grassi, V.; Mirandola, R.; Sabetta, A. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *J. Syst. Softw. 80* (April 2007), 528–558.

[32] ISO/IEC. ISO 15926-2:2003: Industrial automation systems and integration - Integration of life-cycle data for process plants including oil and gas production facilities - Part 2: Data model. Tech. rep., International Organization for Standardization (ISO), Geneva, Switzerland, December 2003.

[33] ISO/IEC. ISO 19501:2005: Information Technology - Open Distributed Processing - Unified Modeling Language (UML) Version 1.4.2. Tech. rep., International Organization for Standardization (ISO), Geneva, Switzerland, January 2005.

[34] ISO/IEC. ISO 19503:2005: Information Technology - XML Metadata Interchange (XMI). Tech. rep., International Organization for Standardization (ISO), Geneva, Switzerland, July 2005.

[35] Jr., E. M. C.; Grumberg, O.; Peled, D. A. *Model Checking*. The MIT Press, 1999.

[36] Kleppe, A. *Software Language Engineering: Creating Domain-specific Languages Using Metamodels*. Addison-Wesley, Upper Saddle River, NJ, 2009.

[37] Kleppe, A.; Warmer, J.; Bast, W. *MDA Explained - The model driven architecture: practice and promise*. Addison Welsey, 2003.

[38] Kuhlmann, M.; Hamann, L.; Gogolla, M. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *Objects, Models, Components, Patterns*, J. Bishop and A. Vallecillo, Eds., vol. 6705 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011, pp. 290–306.

[39] Marinescu, F. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms with Poster*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[40] Mellor, S. J.; Clark, A. N.; Futagami, T. Guest editors' introduction: Model-driven development. *IEEE Software 20* (2003), 14–18.

[41] Meyer, B. *Object-Oriented software construction*, 2nd. ed. Prentice Hall, 1997.

[42] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1. Tech. rep., Object Management Group, December 2009.

[43] OMG. OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3. Tech. Rep. formal/2010-05-03, Object Management Group, 2010.

[44] Petter, A.; Behring, A.; Mühlhäuser, M. Solving constraints in model transformations. In *Proc. of ICMT '09* (2009), vol. 5563 of *LNCS*, Springer, pp. 132–147.

[45] Pohl, K.; Böckle, G.; Linden, F. J. v. d. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[46] Roman, E.; Ambler, S.; Jewell, T. *Mastering Enterprise JavaBeans*. Wiley, 2001.

[47] Rouvoy, R.; Merle, P. Towards a model-driven approach to build component-based adaptable middleware. In *Proc. of the 3rd workshop on Adaptive and reflective middleware* (USA, 2004), ARM '04, ACM, pp. 195–200.

[48] Schlumberger. Ocean Application Development Framework 2009.1 Volume 2 - Petrel Data Access. Tech. rep., Schlumberger, 2009.

[49] Schmidt-Schauss, M.; Smolka, G. Attributive concept descriptions with complements. *Artificial Intelligence 48* (1991), 1–26.

[50] Sen, S.; Baudry, B.; Mottu, J.-M. Automatic model generation strategies for model transformation testing. In *Proc. of ICMT '09*, vol. 5563 of *LNCS*. Springer, 2009, pp. 148–164.

[51] Sommerville, I. *Software Engineering*, 9. ed. Addison-Wesley, Harlow, England, 2010.

[52] Sullins, B.; Whipple, M. *EJB Cookbook*. Manning Publications, May 2003.

[53] Sun Microsystems. Enterprise JavaBeans Specifications Version 2.1. Tech. rep., Sun Microsystems, November 2003.

[54] Sun Microsystems. Java 2 Plataform Enterprise Edition Specification Version 1.4. Tech. rep., Sun Microsystems, November 2003.

[55] Sun Microsystems. The J2EE 1.4 Tutorial Update 7 (for the Sun Java System Application Server Platform Edition 8.2). Tech. rep., Sun Microsystems, December 2005.

[56] Torlak, E. *A constraint solver for software engineering: finding models and cores of large relational specifications*. Tese de Doutorado, Massachusetts Institute of Technology, Cambridge, MA, USA, 2009. AAI0821754.

[57] Warmer, J.; Kleppe, A. *The Object Constraint Language*, 2nd. ed. Addison-Wesley Longman Publishing Co., Inc., 2003.

APÊNDICE A – Propriedades do metamodelo UMLEJB

A seguir são apresentadas todas as propriedades referentes ao modelo de transformação conhecido como *UMLEJB*, referente a transformação de UML para EJB, apresentado no Capítulo 3.

- A transformação não permite a existência de operações conhecidos como *getter* e *setter* no modelo de origem. Esta restrição ocorre porque o metamodelo EJB interpreta seus métodos como o comportamento dos componentes e não como uma abordagem para encapsular o acesso a atributos de uma classe, algo completamente relacionado a técnicas de programação e não de modelagem. Essas operações são detectadas pela análise do padrão existente em seus nomes.

```
context Classifier inv prohibitedGettersandSetters:
self.feature->select(f : Feature | f.oclIsTypeOf(Operation))->collect(f : Feature | f.oclAsType(Operation))->forAll(op :
    Operation | self.feature->select(f : Feature | f.oclIsTypeOf(Attribute))->forAll(f : Feature | 'get'.concat(f.name).
    toLower() <> op.name.toLower() and 'set'.concat(f.name).toLower() <> op.name.toLower()))
```

- Garante que todos os elementos de *DataType* que deviam ser transformados, pela metaclasse *UMLDataTypeToEJBDataType*, realmente foram.

```
context UMLDataTypeToEJBDataType inv verifyUMLDataTypethatmustbetransformedtoEJBDataClass:
DataType.allInstances()->forAll(dt | dt.oclIsTypeOf(DataType) implies dt.transformerToEjbDataType->notEmpty())
```

- Verifica se todo atributo *id*, criado por *UMLClassToEJBKeyClass*, pertence a instância de *EJBKeyClass* gerada pela mesma regra.

```
context UMLClassToEJBKeyClass inv primaryKeymustbelongstoaEJBKeyClassfromUMLClass:
self.id.classifier.name = self.keyClass.name
```

- Garante que todo atributo *id*, criado pela metaclasse *UMLClassToEJBKeyClass*, seja do tipo *EJBInteger*, cujo nome é *Integer*.

```
context UMLClassToEJBKeyClass inv primaryKeymustbeEJBIntegerfromUMLClass:
self.id.type.name = 'Integer'
```

- Verifica se todas as instâncias de *Class* que deviam ser transformadas, pela metaclasse *UMLClassToEJBKeyClass*, realmente foram.

```
context UMLClassToEJBKeyClass inv verifyUMLClassthatmustbetransformedtoEJBKeyClass:
Class.allInstances()->forAll(c | c.oclIsTypeOf(Class) imples c.transformerToClass->notEmpty())
```

- Semelhante ao invariante *primaryKeymustbelongstoaEJBKeyClassfromUMLClass*, porém atesta em relação à metaclasse *UMLAssociationClassToEJBKeyClass*, do modelo de transformação, que possui dois atributos para identificação em vez de apenas um.

```
context UMLAssociationClassToEJBKeyClass inv primaryKeymustbelongstoaEJBKeyClassfromUMLAssociationClass:
self.id1.classifier.name = self.keyClass.name and self.id2.classifier.name = self.keyClass.name
```

- Semelhante a restrição *primaryKeymustbeEJBIntegerfromUMLClass*, porém aplicada a metaclasse *UMLAssociationClassToEJBKeyClass* em vez de *UMLClassToEJBKeyClass*. Nesse caso, dois atributos são verificados, que são *id1* e *id2*.

```
context UMLAssociationClassToEJBKeyClass inv primaryKeymustbeEJBIntegerfromUMLAssociationClass:
self.id1.type.name = 'Integer' and self.id2.type.name = 'Integer'
```

- Verifica se todas as classes associativas UML que deviam ser transformadas, pela metaclasse *UMLAssociationClassToEJBKeyClass*, realmente foram.

```
context UMLAssociationClassToEJBKeyClass inv verifyUMLAssociationClassthatmustbetransformedtoEJBKeyClass:
AssociationEnd.allInstances()->forAll(ae | ae.transformerToAssociationClass->notEmpty())
```

- Garante que todas as instâncias de *Operation* que deviam ser transformadas, por *UMLOperationToBusinessMethod*, realmente foram.

```
context UMLOperationToBusinessMethod inv verifyUMLOperationthatmustbetransformedtoBusinessMethod:
Operation.allInstances()->forAll(op | op.transformerToBusinessMethod->notEmpty())
```

- Atesta se todos os elementos de *Parameter* que deviam ser transformados, pela metaclasse *UMLParameterToEJBParameter*, realmente foram.

```
context UMLParameterToEJBParameter inv verifyUMLParamenterthatmustbetransformedtoEJBParameter:
Parameter.allInstances()->forAll(p | p.transformerToEjbParameter->notEmpty())
```

- Verifica se todos os atributos existentes no modelo UML que deviam ser transformados, por *UMLAttributeToEJBAttribute*, realmente foram.

```
context UMLAttributeToEJBAttribute inv verifyUMLAttributethatmustbetransformedtoEJBAttribute:
Attribute.allInstances()->forAll(a | a.transformerToEjbAttribute->notEmpty())
```

- Garante que apenas as instâncias de *Class* que não são *outermost classes* são transformadas por *UMLClassToEJBDataClass*.

```
context UMLClassToEJBDataClass inv UMLClassfromUMLClassToEJBDataClasscannotbeanOutermostclass:
not self.class.isOuterMostContainer()
```

- Verifica se todas as classes que não são *outermost classes* foram transformadas pela metaclasse *UMLClassToEJBDataClass*.

```
context UMLClassToEJBDataClass inv verifyUMLClassthatmustbetransformedtoEJBDataClass:
Class.allInstances()->forAll(c | c.oclIsTypeOf(Class) and c.feature->select(f | f.oclIsKindOf(AssociationEnd))->collect(f | f
    .oclAsType(AssociationEnd))->exists(ae | ae.composition = true) implies c.transformerToEjbDataClass->notEmpty()
    )
```

- Garante a preservação das relações de composição das classes do modelo UML em EJB, nas instâncias de *EJBDataClass*.

```
context UMLClassToEJBDataClass inv verifyClassIntegrityInEJBDataClasses:
self.class.getOuterMostContainer().transformToEntityComponent.ejbDataClass.feature->select(f | f.oclIsTypeOf(
    EJBAssociationEnd))->collect(f | f.oclAsType(EJBAssociationEnd))->exists(ae | ae.type = self.ejbDataClass)
```

- Instâncias de *Association* usadas na transformação, pela metaclasse *UMLAssociationToEJBDataAssociation*, devem ter, pelo menos, uma de suas pontas de associação com o atributo *composition* verdadeiro.

```
context UMLAssociationToEJBDataAssociation inv UMLAssociationmusthaveoneAssociationEndcompositiontrue:
self.association.oclIsTypeOf(Association) and self.association.end->exists(ae | ae.composition = true)
```

- Garante que todas as instâncias de *Association* que deviam ser transformadas, pela metaclasse *UMLAssociationToEJBDataAssociation*, realmente foram.

```
context UMLAssociationToEJBDataAssociation inv verifyUMLAssociationthatmustbetransformedtoEJBDataAssociation:
Association.allInstances()->forAll(a | a.oclIsTypeOf(Association) and a.end->exists(ae | ae.composition = true) implies a.
    transformerToEjbDataAssociationusingRule5->notEmpty())
```

- Atesta que classes associativas utilizadas na transformação, da metaclasse *UMLAssociationClassToEJBDataClass*, não podem ter pontas de associação com o atributo *composition* verdadeiro.

```
context UMLAssociationClassToEJBDataClass inv
    AssociationClassfromUMLAssociationClassToEJBDataClasscannothaveAssociationEndwithcomposition:
not self.umlAssociationClass.feature->select(f | f.oclIsTypeOf(AssociationEnd))->collect(f | f.oclAsType(AssociationEnd))
    ->exists(ae | ae.otherEnd->exists(oe | oe.composition = true))
```

- Verifica se todas as classes associativas que deviam ser transformadas, pela metaclasse *UMLAssociationClassToEJBDataClass*, realmente foram.

```
context UMLAssociationClassToEJBDataClass inv verifyUMLAssociationClassthatmustbetransformedtoEJBDataClass:
AssociationClass.allInstances()->forAll(ac | ac.transformerToEjbDataClassfromAssociationClass->notEmpty())
```

- Toda instância de *Association*, utilizada por *UMLAssociationEndToEJBDataEndusingRule8*, tem que ser do tipo *Association*, i.e., não pode ser uma *AssociationClass*.

```
context UMLAssociationEndToEJBDataEndusingRule8 inv
    AssociationfromUMLAssociationEndToEJBDataEndusingRule8mustbetypedasAssociation:
self.associationEnd.association.oclIsTypeOf(Association)
```

- Dada uma instância de *AssociationEnd*, para que ela seja utilizada por *UMLAssociationEndToEJBDataEndusingRule8*, a *outermost class* da classe ao qual ela pertence deve ser igual à *outermost class* de seu tipo.

```
context UMLAssociationEndToEJBDataEndusingRule8 inv
    AssociationrelationtypefromUMLAssociationEndToEJBDataEndusingRule8:
self.associationEnd.classifier.getOuterMostContainer() = inst.associationEnd.type.oclAsType(Class).getOuterMostContainer()
```

- Atesta se todas as pontas de associação que deviam ser transformadas, por *UMLAssociationEndToEJBDataEndusingRule8*, realmente foram.

```
context UMLAssociationEndToEJBDataEndusingRule8 inv
    verifyUMLAssociationEndthatmustbetransformedtoEJBDataEndusingRule8:
AssociationEnd.allInstances()->forAll(ae | ae.association.oclIsTypeOf(Association) and ae.navigable = true and ae.classifier.
    getOuterMostContainer() = ae.type.oclAsType(Class).getOuterMostContainer() and ae.navigable = true implies ae.
    transformerToEjbAssociationEndusingRule8->notEmpty())
```

- Semelhante ao invariante *UMLAssociationEndToEJBDataEndusingRule8*, porém analisa a instância de *Association* em relação a metaclasse *UMLAssociationEndToEJBDataEndusing9*.

```
context UMLAssociationEndToEJBDataEndusing9 inv
    AssociationfromUMLAssociationEndToEJBDataEndusingRule9mustbetypedasAssociation:
self.associationEnd.association.oclIsTypeOf(Association)
```

- Oposta ao invariante *AssociationrelationtypefromUMLAssociationEndToEJBDataEndusingRule8*, para que uma instância de *AssociationEnd* seja transformada por *UMLAssociationEndToEJBDataEndusingRule9*, a *outermost class* da classe que ela pertence deve ser diferente da *outermost class* do seu tipo.

```
context UMLAssociationEndToEJBDataEndusingRule9 inv
    AssociationrelationtypefromUMLAssociationEndToEJBDataEndusingRule9:
self.associationEnd.classifier.getOuterMostContainer() <> self.associationEnd.type.oclAsType(Class).getOuterMostContainer
    ()
```

- Verifica se todos os elementos de *AssociationEnd* que deviam ser transformados, pela metaclasse *UMLAssociationEndToEJBDataEndusingRule9*, realmente foram.

```
context UMLAssociationEndToEJBDataEndusingRule9 inv
    verifyUMLAssociationEndthatmustbetransformedtoEJBDataEndusingRule9:
AssociationEnd.allInstances()->forAll(ae | ae.association.oclIsTypeOf(Association) and not ae.classifier.
    getOuterMostContainer() = ae.type.oclAsType(Class).getOuterMostContainer() and ae.navigable = true implies ae.
    transformerToEjbAssociationEndusingRule9->notEmpty())
```

- O atributo *upper* de uma ponta de associação, a ser utilizada na transformação de *UMLAssociationEndEmEJBAssociationusingRule10*, deve ser diferente de *1*.

```
context UMLAssociationEndEmEJBAssociationusingRule10 inv
    AssociationEndUppermustbedifferentthan1forUMLAssociationEndEmEJBAssociationusingRule10:
self.associationEnd.upper <> '1'
```

- Instâncias de *AssociationEnd*, utilizadas pela metaclasse *UMLAssociationEndEmEJBAssociationusingRule10*, devem ter como associações instâncias de *AssociationClass*.

```
context UMLAssociationEndEmEJBAssociationusingRule10 inv
    AssociationofanAssociationEndmustbeanAssociationClassfromUMLAssociationEndEmEJBAssociationusingRule10:
self.associationEnd.association.oclIsTypeOf(AssociationClass)
```

- Para uma instância de *AssociationEnd* ser transformada por *UMLAssociationEndEmEJBAssociationusingRule10*, a *outermost class* de sua associação deve ser igual da *outermost class* de seu tipo.

```
context UMLAssociationEndEmEJBAssociationusingRule10 inv
    outermostAssociationEqualOutermostTypeFromUMLAssociationEndEmEJBAssociationusingRule10:
self.associationEnd.association.oclAsTpe(Class).getOuterMostContainer().name = self.associationEnd.type.oclAsType(Class).
    getOuterMostContainer().name
```

- Garante que o atributo *lower*, pertencente a instância de *EJBAssociationEnd* chamada de *ejbAssociationEnd1*, pela metaclasse *UMLAssociationEndEmEJBAssociationusingRule10*, deve ser igual a *0*.

```
context UMLAssociationEndEmEJBAssociationusingRule10 inv
    lowerEJBAssociationEnd1mustbe0fromUMLAssociationEndEmEJBAssociationusingRule10:
self.ejbAssociationEnd1.lower = '0'
```

- O atributo *upper* do *EJBAssociationEnd*, chamado de *ejbAssociationEnd1*, pela metaclasse *UMLAssociationEndEmEJBAssociationusingRule10*, deve ser igual a *.

```
context UMLAssociationEndEmEJBAssociationusingRule10 inv upperEJBAssociationEnd1mustbe*
    fromUMLAssociationEndEmEJBAssociationusingRule10:
self.ejbAssociationEnd1.upper = '*'
```

- O atributo *lower* referente a *EJBAssociationEnd*, conhecida como *ejbAssociationEnd2*, por *UMLAssociationEndEmEJBAssociationusingRule10*, deve ser igual a *1*.

```
context UMLAssociationEndEmEJBAssociationusingRule10 inv lowerEJBAE2Eq1FromUMLAEToEJBAssocRule10:
self.ejbAssociationEnd2.lower = '1'
```

- Semelhante ao invariante *UMLAssociationEndEmEJBAssociationusingRule10*, porém analisa o atributo *upper*, que deve ter o valor igual a *1*.

```
context UMLAssociationEndEmEJBAssociationusingRule10 inv
    upperEJBAssociationEnd2mustbe1fromUMLAssociationEndEmEJBAssociationusingRule10:
self.ejbAssociationEnd2.upper = '1'
```

- O atributo *composition* da instância de *EJBAssociationEnd* conhecida como *ejbAssociationEnd1*, pela metaclasse *UMLAssociationEndEmEJBAssociationusingRule10*, deve ser igual a *false*.

```
context UMLAssociationEndEmEJBAssociationusingRule10 inv
    compositionAssociationEnd1mustbefalsefromUMLAssociationEndEmEJBAssociationusingRule10:
self.ejbAssociationEnd1.composition = false
```

- As instâncias de *EJBAssociationEnd*, geradas por *UMLAssociationEndEmEJBAssociationusingRule10*, devem pertencer a mesma instância de *EJBDataAssociation*, criada também pela mesma transformação.

```
context UMLAssociationEndEmEJBAssociationusingRule10 inv
    bothEJBAssociationEndsmusthavethesameEJBDataAssociationfromUMLAssociationEndEmEJBAssociationusingRule10:
self.ejbAssociationEnd1.association = self.ejbDataAssociation and self.ejbAssociationEnd2.association = self.
    ejbDataAssociation
```

- Garante que todas as pontas de associação que deviam ser transformadas, pela metaclasse *UMLAssociationEndEmEJBAssociationusingRule10*, realmente foram.

```
context UMLAssociationEndEmEJBAssociationusingRule10 inv
    verifyUMLAssociationEndthatmustbetransformedtoEJBAssociationusingRule10:
AssociationEnd.allInstances(ae | ae.upper <> '1' and ae.association.oclIsTypeOf(AssociationClass) and ae.association.
    oclAsType(AssociationClass).getOuterMostContainerFromAssociationClass() = ae.type->asOrderedSet()->first().
    oclAsType(Class).getOuterMostContainer() and ae.navigable = true implies ae.
    transformerToEjbDataAssociationusingRule10->notEmpty())
```

- O atributo *upper* de uma instância de *AssociationEnd*, utilizada na transformação de *UMLAssociationEndEmEJBAssociationusingRule11*, deve ser diferente de *1*.

```
context UMLAssociationEndEmEJBAssociationusingRule11 inv
    upperAssociationEndmustbedifferentthan1fromUMLAssociationEndEmEJBAssociationusingRule11:
self.associationEnd.upper <> '1'
```

- Pontas de associação, utilizadas por *UMLAssociationEndEmEJBAssociationusingRule11*, devem ter suas associações como classes associativas.

```
context UMLAssociationEndEmEJBAssociationusingRule11 inv
    associationofAssociationEndmustbeanAssociationClassfromUMLAssociationEndEmEJBAssociationusingRule11:
self.associationEnd.association.oclIsTypeOf(AssociationClass)
```

- Para uma instância de *AssociationEnd* ser transformada por *UMLAssociationEndEmEJBAssociationusingRule11*, a *outermost class* de sua associação deve ser diferente da *outermost class* de seu tipo.

```
context UMLAssociationEndEmEJBAssociationusingRule11 inv
    outermostAssociationmustbeequaloutermostTypeAssociationEndfromUMLAssociationEndEmEJBAssociationusingRule11
    :
self.associationEnd.association.oclAsType(Class).getOuterMostContainer().name <> self.associationEnd.type.oclAsType(
    Class).getOuterMostContainer().name
```

- Garante que o atributo *lower*, pertencente a instância de *EJBAssociationEnd*, chamada de *ejbAssociationEnd2*, criada pela metaclasse *UMLAssociationEndEmEJBAssociationusingRule11*, deve ter o valor *1*.

```
context UMLAssociationEndEmEJBAssociationusingRule11 inv lowerEJBAE2Eq1FromUMLAEndToEJBAssocRule11:
self.ejbAssociationEnd2.lower = '1'
```

- Semelhante ao invariante *lowerEJBAE2Eq1FromUMLAEndToEJBAssocRule11*, porém garante o valor igual a *1* ao atributo *upper*.

```
context UMLAssociationEndEmEJBAssociationusingRule11 inv
    upperEJBAssociationEnd2mustbe1fromUMLAssociationEndEmEJBAssociationusingRule11:
self.ejbAssociationEnd2.upper = '1'
```

- Verifica se todas as instâncias de *AssociationEnd* que deviam ser transformadas, por *UMLAssociationEndEmEJBAssociationusingRule11*, realmente foram.

```
context UMLAssociationEndEmEJBAssociationusingRule11 inv
    verifyUMLAssociationEndthatmustbetransformedtoEJBAssociationusingRule11:
AssociationEnd.allInstances()->forAll(ae | ae.upper <> '1' and ae.association.oclIsTypeOf(AssociationClass) and ae.
    navigable = true and ae.association.oclAsType(AssociationClass).getOuterMostContainerFromAssociationClass() <> ae
    .type->asOrderedSet()->first().oclAsType(Class).getOuterMostContainer() and ae.navigable = true implies ae.
    transformerToEjbDataAssociationusingRule11->notEmpty())
```

- Toda classe transformada em um componente de entidade, pela metaclasse *UMLClassToEJBEntityComponent*, deve ser uma *outermost class*.

```
context UMLClassToEJBEntityComponent inv everyoutermostClassbecamesEntityComponent:
self.class.isOuterMostContainer()
```

- Atesta que a instância de *EJBServingAttribute* precisa pertencer ao *EntityComponent* gerado pela mesma instância de *UMLClassToEJBEntityComponent*.

```
context UMLClassToEJBEntityComponent inv
    ServingAttributemustbelongtoEntityComponentfromUMLClassToEJBEntityComponent:
self.servingAttribute.classifier = self.entityComponent
```

- Verifica se o tipo da instância de *EJBServingAttribute* é igual ao de uma *EJBDataClass*, ambos criados pela metaclasse *UMLClassToEJBEntityComponent*.

```
context UMLClassToEJBEntityComponent inv ServingAttributeTypemustbeDataClassfromUMLClassToEJBEntityComponent
    :
self.servingAttribute.type = self.ejbDataClass
```

- Deve-se garantir a relação entre as instâncias de *EJBDataClass* e *EJBDataSchema*, criadas pela metaclasse *UMLClassToEJBEntityComponent*.

```
context UMLClassToEJBEntityComponent inv DataClassPackagemustbeDataSchemafromUMLClassToEJBEntityComponent:
self.ejbDataClass.package = self.dataSchema
```

- Garante se todos as instâncias de *Class* que deviam ser transformadas em componentes por *UMLClassToEJBEntityComponent* realmente foram.

```
context UMLClassToEJBEntityComponent inv verifyUMLClassthatmustbetransformedtoEJBEntityComponent:
Class.allInstances()->forAll(c | c.oclIsTypeOf(Class) and not c.feature->select(f | f.oclIsKindOf(AssociationEnd))->collect
    (f | f.oclAsType(AssociationEnd))->exists(ae | ae.composition = true) implies c.transformerToEntityComponent->
    notEmpty())
```

- Verifica se todos os elementos de *UMLSet* que deviam ser transformados, pela metaclasse *UMLSetToEJBSet*, realmente foram.

```
context UMLSetToEJBSet inv verifyUMLSetthatmustbetransformedtoEJBSet:
UMLSet.allInstances()->forAll(s | s.transformerToSet->notEmpty())
```

APÊNDICE B – Propriedades do metamodelo SDMLtoPPML

A seguir são apresentadas todas as propriedades referentes ao metamodelo *SDMLtoPPML*, apresentado no Capítulo 4.

- Toda instância de *Main*, utilizada na transformação relativa a metaclasse *ManagerToTask*, deve pertencer a uma instância de *Manager*, também utilizada pela mesma regra.

```
context ManagerToTask inv PreVerificationManagerWithMain:
self.manager.main = self.main
```

- Instância de *ParallelTask* criadas pela transformação da metaclasse *ManagerToTask* não pode depender de qualquer outra *ParallelTask*.

```
context ManagerToTask inv ManagerAsTasksDependency:
self.task.dependsFrom->size() = 0
```

- Toda instância de *Main*, utilizada na transformação relativa a metaclasse *MonitorToTask*, deve pertencer a instância de *Monitor*, também utilizada .

```
context MonitorToTask inv PreVerificationMonitorWithMain:
self.monitor.main = self.main
```

- Toda instância de *Task* utilizada na transformação relativa a metaclasse *WorkerToTask* deve pertencer a uma instância de *Worker*, também utilizada pela mesma regra.

```
context WorkerToTask inv PreVerificationWorkerWithTask:
self.worker.task = self.task
```

- Instâncias de *ParallelTask*, criadas pela transformação realizada com base em *WorkerToTask*, devem incluir, como parâmetro, instâncias de *DataStructure* criadas pela mesma regra.

```
context WorkerToTask inv PosVerificationTargetTaskIncludesStructure:
self.targetTask.params->includes(self.structure)
```

- Uma instância de *Data* criada por *WorkerToTask* deve: (i) possuir nome igual a *min*; (ii) tipo de dados igual a *Index3*; (ii) deve pertencer a estrutura de dados criada na mesma transformação.

```
context WorkerToTask inv PosVerificationMinData:
self.min.name = 'min' and self.min.type = 'Index3' and self.min.structure = self.structure
```

- Semelhante ao invariante *PosVerificationMinData*, deve possuir o nome igual a *max*.

```
context WorkerToTask inv PosVerificationMaxData:
self.max.name = 'max' and self.max.type = 'Index3' and self.max.structure = self.structure
```

- A instância de *DataStructure*, referente ao cubo sísmico de entrada de um *Worker*, deve possuir a estrutura da instância de *ParallelTask* criada pela transformação *WorkerToTask*.

```
context WorkerToTask inv PosVerificationInputWithStructure:
self.input.structure = self.structure
```

- Semelhante ao invariante *PosVerificationInputWithStructure*, porém analisa o cubo sísmico de saída de um *Worker*.

```
context WorkerToTask inv PosVerificationOutputWithStructure:
self.output.structure = self.structure
```

- O atributo *inlineBuffer* de um *Context* deve ser transformada, pela metaclasse *ContextToDataStructure*, em uma instância de *Data*, cujo nome é *inlineBuffer*, e pertencer a estrutura de dados criada na mesma transformação.

```
context ContextToDataStructure inv PosVerificationInlineBuffer:
self.inlineData.name = 'inlineBuffer' and self.inlineData.structure = self.structure
```

- Semelhante ao invariante *PosVerificationInlineBuffer*, porém analisa o atributo *crosslineData* de um *Context*.

```
context ContextToDataStructure inv PosVerificationCrosslineBuffer:
self.crosslineData.name = 'crosslineBuffer' and self.crosslineData.structure = self.structure
```

- Semelhante ao invariante *PosVerificationInlineBuffer*, porém analisa o atributo *zData*.

```
context ContextToDataStructure inv PosVerificationZBuffer:
self.zData.name = 'zBuffer' and self.zData.structure = self.structure
```

- Semelhante ao invariante *PosVerificationInlineBuffer*, porém analisa o atributo *edgeStrategy*.

```
context ContextToDataStructure inv PosVerificationEdgeStrategy:
self.edgeData.name = 'edgeStrategy' and self.edgeData.structure = self.structure
```

- Garante que a dimensão *Inline* de um cubo sísmico, após a transformação pela metaclasse *SeismicCubeToDataStructure*, possua o tipo *Dimension<Inline>*.

```
context SeismicCubeToDataStructure inv PosVerificationInlineType:
self.inlineData.type = 'Dimension<Inline>'
```

- Semelhante ao invariante *PosVerificationInlineType*, analisa a dimensão *Crossline*, que deve possuir o tipo *Dimension<Crossline>*.

```
context SeismicCubeToDataStructure inv PosVerificationCrosslineType:
self.crosslineData.type = 'Dimension<Crossline>'
```

- Semelhante ao invariante *PosVerificationInlineType*, analisa a dimensão *Z*, que deve possuir o tipo *Dimension<Z>*

```
context SeismicCubeToDataStructure inv PosVerificationZType:
self.zData.type = 'Dimension<Z>'
```

- Dado uma instância de *Context c* e uma instância de *SeismicCube sc* a serem transformadas pela metaclasse *ContextToContract*, o *Manager* de *c* deve possuir *sc* como seu cubo sísmico de entrada.

```
context ContextToContract inv PreVerificationContextCube:
self.context\_.manager.input->includes(self.cube)
```

- Garante que a instância de *BinaryOperation* referente a uma restrição da dimensão *inline*, criada por *ContextToContract*, possua o atributo *operation* igual a $<=$.

```
context ContextToContract inv PosVerificationInlineContract:
self.inlineContract.operation\_ = '<='
```

- Semelhante ao invariante *PosVerificationInlineContract*, porém para uma restrição da dimensão *crossline*.

```
context ContextToContract inv PosVerificationCrosslineContract:
self.crosslineContract.operation\_ = '<='
```

- Semelhante ao invariante *PosVerificationInlineContract*, porém para uma restrição da dimensão *z*.

```
context ContextToContract inv PosVerificationZContract:
self.zContract.operation\_ = '<='
```

- Uma instância de *Dimension* a ser transformada pela metaclasse *WorkerDimensionToData* precisa possuir alguma instância de *Worker*.

```
context WorkerDimensionToData inv PreVerificationDimensionHasWorker:
self.dimension.worker->size() > 0
```

- Instâncias de *Dimension*, para serem transformadas por *WorkerDimensionToData*, realmente precisam ser do tipo *Dimension*, i.e., não pode pertencer de nenhuma das metaclasses que a herdam.

```
context WorkerDimensionToData inv PreVerificationDimensionType:
self.dimension.oclIsTypeOf(Dimension)
```

- Garante que a estrutura entre instâncias de *Manager*, *SeismicCube* e *Context* sejam preservadas em PPML.

```
context ManagerToTask inv IntegrityConstraintManagerContextCube:
self.manager.context\_->size() > 0 implies (self.manager.input.ruleSeismicCubeToDataStructure.structure->forAll(
    cubeStructure | self.manager.context\_.ruleContextToDataStructure.structure.dependsFrom->exists(dependsCube |
    cubeStructure = dependsCube) and self.manager.context\_.ruleContextToDataStructure.structure.params\_ = self.
    task and cubeStructure.params\_ = self.task))
```

- Garante que a estrutura entre instâncias de *Manager*, seus cubos sísmicos de entrada e suas dimensões são mantidas em PPML.

```
context ManagerToTask inv verifyCubeIntegrityInManagerToTask:
self.task.params->select(ds | not ds.ruleSeismicCubeToDataStructure->isEmpty())->forAll(ds | ds.data->select(dt | not
    dt.ruleSeismicCubeToDataStructure\_inline->isEmpty())->forAll(dt | self.manager.input->includes(dt.
    ruleSeismicCubeToDataStructure\_inline.cube)) and ds.data->select(dt | not dt.ruleSeismicCubeToDataStructure\
    _crossline->isEmpty())->forAll(dt | self.manager.input->includes(dt.ruleSeismicCubeToDataStructure\_crossline.
    cube)) and ds.data->select(dt | not dt.ruleSeismicCubeToDataStructure\_z->isEmpty())->forAll(dt | self.manager.
    input->includes(dt.ruleSeismicCubeToDataStructure\_z.cube)))
```