Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Composição de UML *Profiles*

Kleinner Silva Farias de Oliveira

Orientador: Prof. Dr. Toacy Cavalcante de Oliveira

Porto Alegre
2008

Kleinner Silva Farias de Oliveira

Composição de UML *Profiles*

Dissertação apresentada como requisito para obtenção do grau de Mestre em Ciência da Computação, pelo Programa de Pós-Graduação da Faculdade de Informática da Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador:
Prof. Dr. Toacy Cavalcante de Oliveira

Porto Alegre
2008

Dados Internacionais de Catalogação na Publicação (CIP)

O48c Oliveira, Kleinner Silva Farias de.
Composição de UML profiles / Kleinner Silva Farias de Oliveira. – Porto Alegre, 2008.
195 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Toacy Cavalcante de Oliveira

1. Informática. 2. Engenharia de Software. 3. UML (Informática). 4. Modelagem de Sistemas. I. Título.

CDD 005.114

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada **"Composição de UML Profiles"**, apresentada por Kleinner Silva Farias de Oliveira, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas de Informação, aprovada em 28/02/08 pela Comissão Examinadora:

Prof. Dr. Toacy Cavalcante de Oliveira – Orientador — PPGCC/PUCRS

Prof. Dr. Ricardo Melo Bastos – — PPGCC/PUCRS

Prof. Dr. Sérgio Crespo Coelho da Silva Pinto – — UNISINOS

Homologada em 19/02/08, conforme Ata No. 017/08 pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

Campus Central
Av. Ipiranga, 6681 – P32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

Resumo

Com o sucesso da MDA (*Model Driven Architecture*) e da UML (*Unified Modeling Language*), modelos estão substituindo código como o principal artefato de desenvolvimento de software. Em MDA, a transformação e a composição de modelos são duas atividades essenciais. Enquanto a transformação de modelos tem sido amplamente pesquisada e documentada, a composição de modelos precisa de mais investigação. Com a MDA, surgiram três desafios: (i) criar linguagens de modelagem específicas de domínios (DSML); (ii) compor DSML; (iii) compor modelos representados em DSML. A UML permite a construção de DSML através de UML *profiles*, porém não oferece um mecanismo adequado para tais *profiles*. Neste contexto, o presente trabalho apresenta uma proposta de mecanismo de composição de UML *profiles* fundamentado em regras de composição, regras de transformação de modelos, estratégias de composição, estratégia de comparação e regras de comparação. Um modelo formal deste mecanismo foi construído utilizando a linguagem de modelagem formal *Alloy* e foi realizada uma análise automática do modelo usando *Alloy Analyzer*. Além disso, uma ferramenta de software foi construída com o objetivo de validar o mecanismo e automatizar a abordagem.

Palavras-Chave: Engenharia de Software, UML, Desenvolvimento Dirigido por Modelos, Linguagem de Modelagem Específica de Domínio, UML *profiles*

Abstract

With the success of Model Driven Architecture (MDA) and Unified Modeling Language (UML), models are replacing code as the first software development artifact. In MDA, model transformation and model composition are essential activities. While model transformation has been well researched and documented, model composition needs more investigation. With MDA arise three challenges, such as: (*i*) create domain specific modeling languages (DSMLs); (*ii*) merge DSML; and (*iii*) merge models expressed in DSML. The UML allows building DSML through UML profiles, however it does not provide an adequate mechanism to merge such profiles. With this in mind, this work proposes a UML profiles composition mechanism based on merge rules, model transformation rules, composition strategy, match strategy and match rules. A formalization of this mechanism was built using the Alloy formal language and automatic analysis were accomplished using Alloy Analyzer. Moreover, a model composition tool was developed to evaluate the mechanism and automate the approach.

Keywords: Software Engineering, UML, Model Driven Development, Domain Specific Modeling Language, UML profiles

Agradecimentos

A Deus, por me oferece oportunidades maravilhosas na minha vida, uma fonte inesgotável de força, paz e sabedoria.

Aos meus pais, Maria Sandra e José Carlos, que são tudo na minha vida, por estarem comigo nas horas boas e más, sempre dando todo apóio que precisei. Sem eles minhas conquistas não seriam possíveis. Devo tudo a vocês.

A minha vó querida, Valdenira, que sempre me deu apóio e carinho. Deus abençoe onde ela esteja neste momento. Muitas saudades.

Aos meus irmãos, Kelyne, Kleberson e Klyvia, que sempre me ajudaram e deram forças nos momentos difíceis e alegres da minha caminhada. Apesar da distância durante esta jornada jamais esqueci de vocês.

Ao meu orientador Toacy Cavalcante de Oliveira que depositou confiança no meu trabalho e me incentivou todo o momento. Obrigado pela atenção, dedicação e conhecimento passado. Seus ensinamentos me ajudaram a ver a vida de um modo diferente. Meus sinceros agradecimentos.

Agradeço à minha namorada querida, Carla, pela atenção, carinho e incentivo durante todo o momento. Obrigado pela dedicação e paciência em compreender a dura e árdua jornada enfrentada.

A todos os meus amigos, pelo apoio, torcida e profunda amizade. Aprendi muito com vocês.

À CAPES pelo apoio financeiro

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Abreviaturas

CWM	Common Warehouse Metamodel	16
DSML	Domain Specific Modeling Language	16
EMF	Eclipse Modeling Framework	16
GSD	Global Software Development	16
GUI	Graphical User Interface	16
MDA	Model Driven Architecture	16
MDD	Model Driven Development	16
MoCoTo	Model Composition Tool	16
MOF	Meta Object Facility	16
MSR	Merge Specification Rule	16
OMG	Object Management Group	16
UML	Unified Modeling Language	16
XMI	XML Metadata Interchange	16

Sumário

Capítulo 1

Introdução

Os avanços alcançados no desenvolvimento de sistemas de software têm estimulado a construção de sistema de software com elevado nível de complexidade e tamanho cada vez maiores. Estes sistemas têm destacado o inadequado nível de abstração fornecido pelas modernas linguagens de programação de alto nível, como, por exemplo, Java, C#, Python. Assim, foi gerado uma demanda por linguagens, métodos e tecnologias que aumentam o nível de abstração no qual os sistemas são idealizados, construídos e evoluídos (France et al. 2006).

Uma tendência em Engenharia de Software consiste em tratar, através do uso de linguagens de modelagem, programas no nível dos seus conceitos, com o objetivo de simplificar o projeto, a evolução e a manutenção destes. Além disso, deseja-se aumentar a capacidade de adaptação frente às rápidas mudanças das tecnologias e atender às exigências do *time-to-market*. Esta tendência propõe o aumento do nível de abstração no qual os software são projetados e desenvolvidos, mudando o foco do código para os modelos. De acordo com (Selic 2003), estas mudanças são comparadas com o salto da linguagem *assembly* à terceira geração de linguagem de programação. A MDA (OMG 2003) é um exemplo desta tendência, a qual trata-se de uma abordagem MDD (Atkinson & Kuhne 2003) da OMG (OMG 2007*b*).

No contexto da MDA, ao contrário de se construir software através da integração de componentes de software existentes, ou seja, colocando-os para interoperarem, o objetivo é construir novas aplicações ricas em funcionalidades a partir da transformação e composição d modelos dos componentes de software existentes. Na MDA, a transformação e composição de modelos são duas atividades essenciais e têm se tornado cada vez mais importante. A transformação de modelos tem sido pesquisada, documentada e tem alcançado avanços importantes, por outro lado, a composição de modelos necessita de maiores investigações e esforços para solucionar significantes problemas. De acordo com (Bézivin et al. 2006), a composição de modelos trata-se de uma nova área de pesquisa e se encontra na sua "infância".

Sendo assim, com a mudança do foco do código para os modelos, as linguagens de modelagem ganharam destaques e estão cada vez mais se adaptando e apresentando novos recursos, com o objetivo de aumentar suas capacidades de representação. Várias linguagem de modelagem foram desenvolvidas, destacando-se a UML da OMG. A UML é um linguagem de modelagem de propósito geral que apresenta a capacidade de modelar um grande e diversificado conjunto de domínios de aplicações. Foi adotada como linguagem de modelagem padrão, sendo usada tanto comercialmente quanto na academia. Porém, nem todas as suas capacidades de modelagem são necessariamente úteis em todos os domínios de aplicação, o que levou a linguagem a ser estruturada de forma modular, tornando possível o uso apenas de partes da linguagem as quais são de interesse para um domínio específico.

O sucesso da UML e da MDD têm direcionado e estimulado a proliferação do uso dos modelos. Diante disto, deseja-se desenvolver aplicações grandes e complexas (as quais geralmente são formadas por um conjunto de aplicações menores que representam os diversos domínios da aplicação) a partir da composição dos modelos que representam os diferentes domínios da aplicação. Ou seja, compor modelos específicos de domínio a fim de construir aplicações cada vez maiores. Segundo (Estublier & Ionita 2005), o desafio enfrentado é duplo: (i) criar domínios conceituais capazes de modelar um conjunto grande de aplicações pertencentes a um domínio; e (ii) compor modelos conceituais a fim de modelar uma aplicação específica.

Alinhado com estes dois desafios, surgem duas observações. Primeira, pelo fato da UML ser um linguagem de caráter geral e ter um escopo de aplicação muito grande, em determinada situações não é capaz de modelar aplicações definidas em domínios específicos. Tal fato caracteriza a situação onde a sintaxe ou a semântica especificada na UML não é capaz de expressar conceitos de um sistema ou domínio em particular, ou quando deseja-se tornar os elementos já existentes mais restritos, ou seja, diminuir o seu escopo. Sendo assim, surge a necessidade de criar um extensão da linguagem, ou seja, criar uma linguagem de modelagem específica de domínio (DSML). Segunda, com a MDA surgiram três desafios: (i) criar DSMLs; (ii) compor DSML's; (iii) compor modelos representados em DSMLs.

Diante deste contexto, é possível criar DSML's através de UML *profiles*, o qual consiste em um mecanismo que permite que os elementos definidos no metamodelo da UML possam ser estendidos, permitindo, desse forma, que a UML seja adaptada a fim de aumentar ou especializar sua expressividade. Esta extensão permite a UML atender as exigências de determinadas plataforma e domínios específicos. A composição destes *profiles* consiste em uma necessidade clara, porém não encontrada na literatura configurando, desse modo, um tópico de pesquisa em aberto.

1.1 Motivação

Nesta Seção é especificada a motivação para a construção de uma nova abordagem para o mecanismo de composição da UML, a fim de tornar possível a composição de UML *profiles*. Alguns esforços na área de composição de modelos têm sido realizados considerando a natureza inicial desta área de pesquisa, porém nenhum destes esforços são direcionados para a composição de DSMLs geradas a partir do mecanismo de extensão conservativa da UML. Ou seja, nenhum trabalho focado na composição de *profiles*. Sendo assim, é possível construir e fazer uso de *profiles*, porém não é disponibilizado uma especificação de como realizar sua composição.

Para ilustrar como a ausência de um mecanismo de composição de *profiles* pode levar ao surgimento de problemas durante o seu uso, foi criado dois exemplos ilustrativos descritos a seguir.

Exemplo de Divisão e Conquista

Uma forma simples de entender e resolver problemas complexos é dividi-los em partes menores (princípio de divisão e conquista) (Pólya 2004). Como a Engenharia de Software não é diferente neste aspecto, é seguido este princípio no exemplo de motivação. Sendo assim, construir e modelar aplicações de grande porte pode implicar em fazer uso de diferentes plataformas de desenvolvimento e domínios de aplicação simultaneamente. Além disso, modelar uma aplicação seguindo o princípio da separação de interesses, implica em evitar a construção de modelos grandes e monolíticos, os quais são difíceis de serem manipulados e entendidos.

Com isto em mente, uma boa forma de modelar uma aplicação, é dividi-la em domínios da aplicação para trabalhar com a heterogeneidade dos modelos e os conceitos específicos dos domínios. Observando a Figura 1.1(a) é possível notar uma especificação abstrata de uma aplicação Web, *WebApp*, a qual pode ser definida através da integração de diferentes visões abstratas, $WebApp_1$, $WebApp_2$,..., $WebApp_n$, as quais definem um conjunto de entidades relacionadas aos seus domínios. Para cada domínio, existe um D_x ($1 \leq x \leq n$, $n \in N_+^*$) para representar uma visão abstrata do mesmo. Os conceitos identificados em $WebApp_x$ são representados e definidos seus relacionamentos em D_x, no qual podem ser representados em termos de um metamodelo. Este metamodelo inclui a definição dos conceitos do domínio, os relacionamento entre eles, e as restrições que governam tantas as estruturas estáticas, quanto as comportamentais destas entidades.

Para cada D_x é criado um *profile* baseado no metamodelo do domínio. Para cada elemento relevante, que é identificado no metamodelo de domínio, deve ser criado uma notação mais adequada para melhor expressá-lo ou representá-lo. Então, *stereotypes* são criados e inseridos no *profile*. Quando surge a necessidade de integrar os D_x's, a fim de obter uma visão única do metamodelo de domínio, também surge a necessidade de

compor os *profiles* que representam os D_x's, com o objetivo de ter um *profile* único capaz de representar uma visão integrada dos domínios.

Neste momento, apresenta-se um incapacidade apresentada pela UML. Ela permite a construção dos *profiles*, porém não oferece recursos para realizar a composição dos mesmos. Logo, um *profile* único que possua uma visão completa/integrada dos conceitos não pode ser construído.

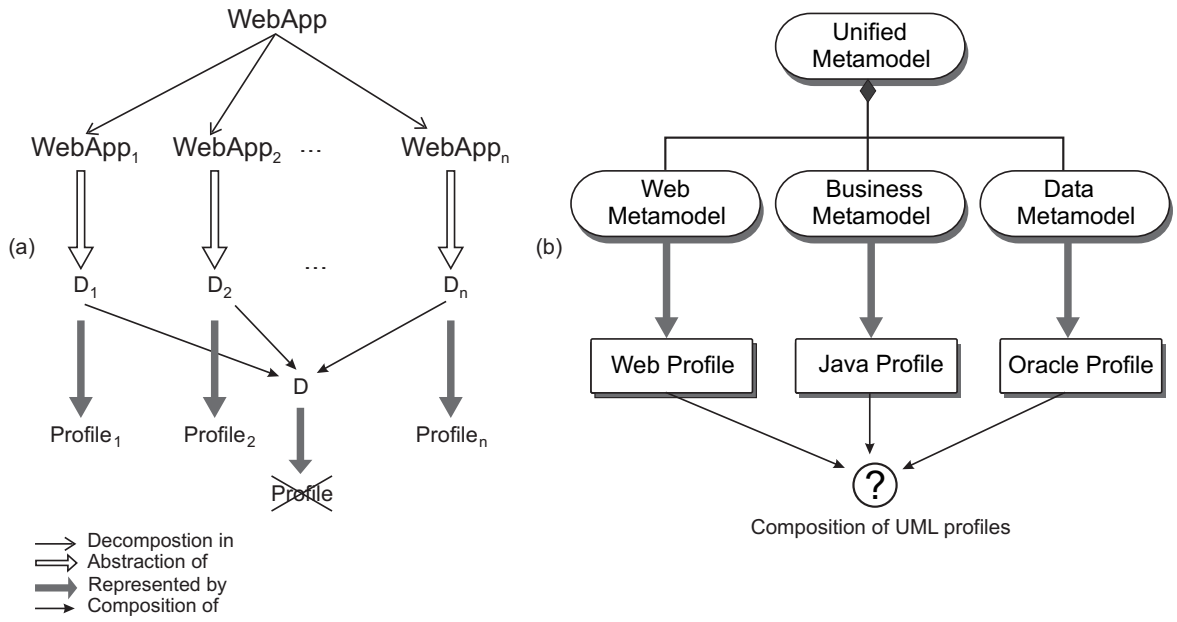

Figura 1.1: Exemplo de problemática na composição de *profiles*

De modo similar, a Figura 1.1(b) mostra uma aplicação que é implementada através de três camadas: *interface*, *business*, e *database*. Para cada camada existe um metamodelo que representa os conceitos específicos do domínio e o relacionamento entre eles. Para representar e expressar estes conceitos com UML, são criados os *profiles*, *Web Profile*, *Java Profile*, e *Oracle Profile*. Porém, não é possível novamente realizar a composição destes *profiles*.

Exemplo de Problemática na Aplicação de *Profiles*

Uma vez que tenham sido definidos alguns *profiles*, quando os mesmos são utilizados pode surgir alguns problemas. A Figura 1.2 ilustra este problema. Dado dois *profiles* `Tree` e `Topology`, os quais possuem um conjunto de *stereotypes* e classes, quando são aplicados em `MyApplication` alguns problemas surgem, tais como: (i) não é possível saber se o *stereotype* `Node` aplicado a `ApplicationServer` é do *profile* `Tree` ou `Topology`; (ii) o *stereotype* `Edge` possui a *tagged values* `name`?; (iii) os *stereotypes* `EndNode` e `Leaf` representam conceitos de domínios iguais sendo aplicado com a mesma finalidade.

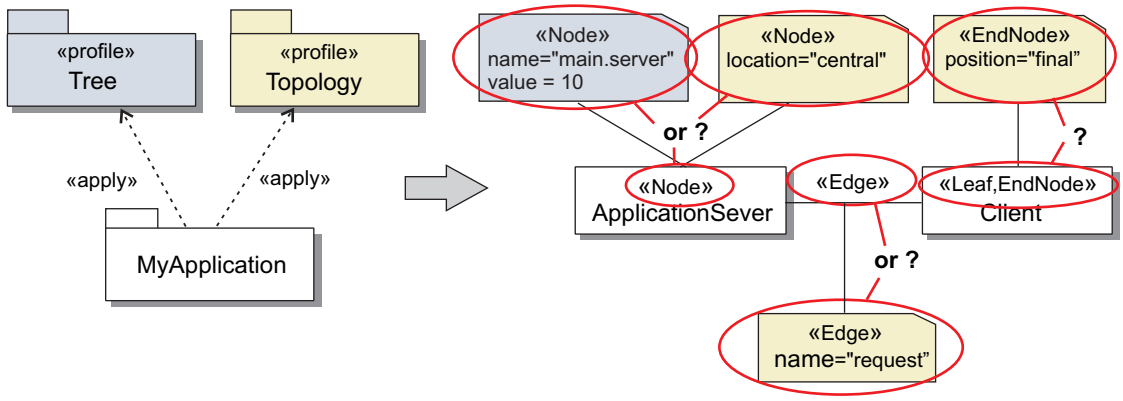

Figura 1.2: Exemplo ilustrativo de um problema no uso de *profiles*

1.2 Problemática

A semântica do mecanismo de composição da UML não é bem definida, sendo difícil de especificar o que ele realmente faz, e quais as suas propriedades. Sendo assim, é possível criar *profiles*, porém não é possível realizar a sua composição. Além disso, as mais populares ferramentas de modelagem (como IBM Rational Software Modeler) não implementam o mecanismo de composição da UML. De acordo com Rumbaugh, Jacobson and Booch (Rumbaugh et al. 2005): "o uso do mecanismo de composição da UML pode gerar confusões e deve ser evitado se possível". Sendo assim, é listado algumas problemáticas específicas que são abordadas nesta dissertação e que são descritas a seguir.

Problemática 01 (P1). Não existe uma especificação clara de como deve ser realizada a composição de modelos, quais atividades devem ser realizadas e os passos a serem seguidos.

Problemática 02 (P2). A semântica do mecanismo de composição da UML não é bem definida. Em (Rumbaugh et al. 2005), isto tem sido definido como complexo, difícil e complicado de ser usado. Tal argumentação é fundamentada pela ausência de uma especificação que defina o papel de cada modelo no processo de composição.

Problemática 03 (P3). O mecanismo de composição da UML não é capaz de realizar a composição de UML *Profiles* (OMG 2007*c*), apresentando ambigüidade e inconsistência na sua definição. Além disso, este mecanismo é incompleto, pois não define, por exemplo, como dois *stereotypes* devem ser compostos.

Problemática 04 (P4). Ferramentas de modelagem não implementam o mecanismo de composição de modelos definido na UML, por exemplo, IBM Rational Software Modeler (IBM 2007). Um vez que surja a necessidade de realizar a composição de *profiles*, esta composição será realizada manualmente o que torna esta atividade

difícil, custosa e de grande esforço quando os *profiles* possuem um número elevado de elementos.

Questão de Pesquisa

A UML permite a criação de DSML através de UML *profiles*, porém seu mecanismo de composição (*Package Merge* (OMG 2007*c*)) apresenta inconsistência, ambigüidade e define regras de composição gerais as quais são aplicadas aos *metatypes* do metamodelo que são mais freqüentemente utilizados, por exemplo: *Classes*, *Associations*, *Properties* e entre outros. Não existe um definição em (OMG 2007*c*) de como deve ser realizada a composição de *metatypes*, como: *Packages*, *Profiles*, *State Machines*, *Use Case*, etc. Neste sentido, surge a questão de pesquisa deste trabalho: **"Como realizar a composição de UML *Profiles*?"**

1.3 Objetivos

O estudo realizado neste trabalho tem como objetivo geral desenvolver um mecanismo de composição de modelos capaz de realizar a composição de UML *profiles*. Como objetivos específicos têm-se:

1. Identificar as atividades necessárias para realizar a composição de modelos e as fases nos mecanismos de composição.
2. Elaborar um guia de composição de modelos.
3. Definir operadores de composição juntamente com regras de comparação, regras de composição e regras de transformação.
4. Propor a partir das necessidades do mecanismo de composição, uma extensão ao metamodelo da UML para suporta a representação do mecanismo de composição de modelos proposto.
5. Realizar uma especificação formal do mecanismo de composição utilizando *Alloy* e usando o *Alloy Analyzer* (Jackson 2007).
6. Desenvolver uma ferramenta que possibilite a composição de *profiles*, baseado no mecanismo de composição especificado.
7. Realizar a composição de UML *profiles* utilizando o mecanismo proposto, utilizando como base a ferramenta desenvolvida.

1.4 Contribuições

Nas seções anteriores foram definidos os objetivos, os problemas encontrados na composição de *profiles* e a ausência de uma definição de uma mecanismo de composição de *profiles*. Desse modo, a abordagem desenvolvida representa um esforço na solução destes problemas, constituindo, desta forma, a base da contribuição. Além disto, este trabalho representa um primeiro esforço na realização de composição de DSML no contexto da UML e na elaboração de um guia de composição de modelos. Assim, um resumo das contribuições geradas pela pesquisa descrita nesta dissertação é apresentado a seguir:

Estudo Comparativo dos Mecanismos de Composição: a linha de pesquisa de composição de modelos necessita de estudos comparativos das abordagens existentes devido ao fato de se encontrar no seu estado inicial (Bézivin et al. 2006). Logo, a descrição e comparação das abordagens de composição de modelos existentes caracteriza-se como uma contribuição. Sendo assim, foi elaborado um resumo comparativo das semelhanças e diferenças de outras abordagem existentes com a da abordagem proposta. Além disto, este trabalho representa uma bom referencial na área de pesquisa de composição de modelos.

Guia para Composição de Modelos: a partir da especificação das atividades necessárias para realizar a composição de UML *Profiles* e da definição do fluxo entre elas, buscou-se disponibilizar um guia para realizar a composição de modelos. Desta forma, de acordo com os modelos a serem composto (por exemplo, UML *profiles*, diagrama de classes), tem-se um embasamento de como deve se proceder a composição destes modelos. Este guia servirá como base para futuros esforços na área de composição de modelos, assim como, para o desenvolvimento de aplicações neste contexto. O guia de composição de modelos elaborado foi publicado em (Oliveira & Oliveira 2007*a*).

Operadores de Composição: quatro operadores de composição são apresentados: (i) *Match Operator*; (ii) *Merge Operator*; (iii) *Composition Strategy Operator*; e (iv) *Model Transformation Operator*. Estes operadores fazem uso de *match rules*, *merge rules*, e *model transformation rules* para desempenhar seus respectivos papeis dentro do mecanismo de composição. Estes operadores fazem parte do artigo publicado em (Oliveira & Oliveira 2007*b*).

Extensão do Metamodelo da UML: com a construção de um metamodelo de composição de modelos que estende o metamodelo da UML é possível especificar o relacionamento de composição entre dois *profiles*. Para isto, foi desenvolvido: (i) um metamodelo que define a sintaxe e semântica do relacionamento de composi-

ção; (ii) regras de boa formação que especificam as restrições no relacionamento de composição; (iii) descrição da semântica de composição.

Modelo Formal do Mecanismo de Composição: o mecanismo de composição foi modelado em Alloy, uma linguagem formal baseada em teoria dos conjuntos e lógica de predicado, apresentando um aspecto formal do proposta, algo não apresentado na UML. Este modelo formal permite realizar verificações, análises automáticas do mecanismo, e checar a corretude da proposta. Uma vez definido o modelo formal, foram realizadas análises de algumas propriedades algébricas utilizando o *Alloy Analyzer*. Além disto, com o modelo formal, futuras evoluções da proposta ou do metamodelo da UML podem ser analisadas com o objetivo de verificar o impacto das mudanças na abordagem. Com isso, é possível checar se a abordagem é válida, ou inválida perante as modificações.

Ferramenta de Composição de Modelos: a implementação do mecanismo de composição de *profiles* disponibiliza uma forma automática de realizar a composição dos mesmos, o que não é apresentado nas mais populares ferramentas de modelagem.

Uma outra importante contribuição é a construção do mecanismo de composição baseado em regras de composição, regras de transformação de modelos, regras de comparação, estratégia de comparação e estratégias de composição. Com isso, é possível desenvolver outros trabalhos adaptando estas regras e estratégias para outros contextos. Além disso, a construção de uma ferramenta coloca em prática o mecanismo de composição proposto, fornecendo um ambiente de composição de UML *profiles*.

1.5 Etapas da Pesquisa

O trabalho apresentado neste documento foi realizado por meio de várias etapas, as quais estão representadas pela Figura 1.3 e são descritas a seguir:

Etapa 1: nessa etapa realizou-se um estudo sobre os mecanismos de composição existentes, onde coletou-se informações sobre as formas e técnicas utilizadas na realização de composição. Ainda nessa etapa, realizou-se um levantamento das estratégias usadas para solucionar os conflitos que surgem durante a composição e um estudo aprofundado da UML.

Etapa 2: nessa etapa realizou-se uma análise dos resultados obtidos da Etapa 1 para identificar os procedimentos e as fases que são freqüentes no mecanismos de composição. Com base nestas observações, foi desenvolvido o mecanismo de composição

de UML *profiles*. Ainda nesta fase, foi desenvolvido uma extensão do metamodelo da UML para representar o mecanismo de composição.

Etapa 3: nessa etapa realizou-se um especificação formal do mecanismo de composição proposto na Etapa 2 na linguagem *Alloy*.

Etapa 4: baseado no mecanismo desenvolvido na Etapa 2 e na especificação da Etapa 3 foi desenvolvido a ferramenta de composição de modelos MoCoTo (*Model Composition Tool*).

Etapa 5: após o desenvolvimento da ferramenta na Etapa 4, foi realizado a composição de *profiles* na prática. Ainda nesta fase, foi elaborada a dissertação e defesa da dissertação.

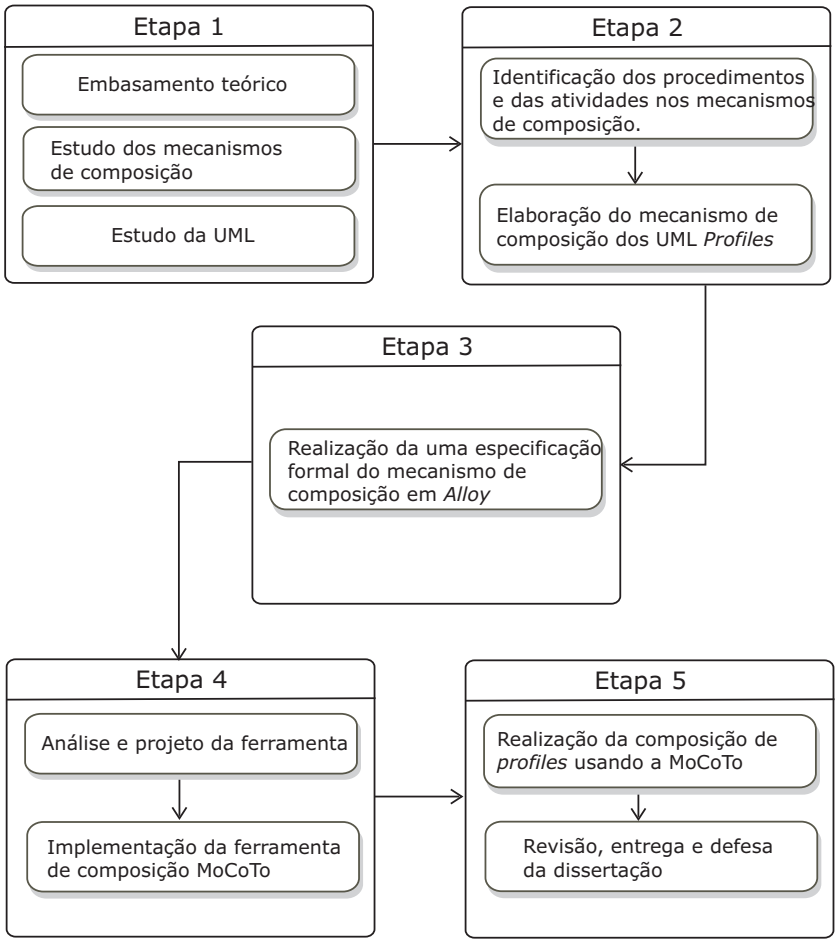

Figura 1.3: Esboço das etapas realizadas durante o desenvolvimento da pesquisa

1.6 Estrutura do Trabalho

A dissertação é organizada nos seguintes capítulos:

Capítulo 2: apresenta as fundamentações teóricas, discutindo sobre os principais temas discutidos nesta dissertação.

Capítulo 3: apresenta uma análise comparativa dos trabalhos relacionados com a proposta apresentada nesta dissertação.

Capítulo 4: descreve a definição do mecanismo de composição de *profiles*.

Capítulo 5: apresenta uma descrição das estratégias de composição do mecanismo de composição proposto.

Capítulo 6: apresenta uma extensão do metamodelo da UML, bem como um exemplo de uso.

Capítulo 7: apresenta a especificação formal do mecanismo do mecanismo de composição em Alloy.

Capítulo 8: apresenta a ferramenta de composição de modelos e sua aplicação na composição de UML *profiles*.

Capítulo 9: descreve as conclusões deste trabalho, assim como as suas perspectivas futuras.

Anexo A: são apresentados diagramas do metamodelo da UML que são utilizados ao longo da dissertação.

Anexo B: é apresentado uma descrição das regras de composição utilizadas na abordagem.

Anexo C: é apresentada a modelagem do mecanismo de composição em Alloy, juntamente com a análise modelagem.

Anexo D: é apresentado um histórico dos trabalhos desenvolvido durante o transcorrer da pesquisa.

Capítulo 2

Fundamentação Teórica

Antes de apresentar a definição do mecanismo de composição, é necessário descrever alguns conceitos relacionados a proposta desta dissertação. Neste capítulo é apresentada uma descrição dos UML *profiles* e a relação destes *profiles* com o mecanismo de extensão da UML. Esta descrição serve como base para justificar as peculiaridades apresentadas pelos *profiles*. Além disso, apresenta-se uma descrição da UML que dá suporte a construção e definição dos *profiles*. Por fim, o mecanismo de composição definido na UML é descrito de forma resumida a fim de caracterizar a necessidade de uma abordagem para promover a composição dos *profiles*.

2.1 *Unified Modeling Language*

A UML (Rumbaugh et al. 2005, OMG 2007*c*) é uma linguagem visual que é aplicada para a especificação, construção e documentação de artefatos de software. Trata-se de uma linguagem de modelagem de propósito geral que pode ser usada com o paradigma orientado a objetos ou a componentes. Além disso, a linguagem caracteriza-se pela aplicabilidade em diferentes domínio de aplicações e plataformas de implementação (por exemplo J2EE (Sun 2007*a*), J2ME (Sun 2007*b*), CORBA (OMG 2007*a*)). Além disso, a UML tem se tornado um linguagem de modelagem padrão, sendo usada tanto na indústria quando na academia.

2.1.1 Arquitetura da UML

A especificação da UML é baseada na abordagem de metamodelagem, o que implica em ter um metamodelo responsável pela sua definição. Este metamodelo foi implementado seguindo os seguintes princípios (OMG 2007*c*):

- **Modularidade:** visa uma forte coesão e um baixo acoplamento na definição da linguagem. Isto é caracterizado pela definição de pacotes, que representam áreas

conceituais, e a organização dos recursos da linguagem em metaclasses.

- **Camada:** a arquitetura do metamodelo é definida em quatro camada com o objetivo de separar os interesses nas quartos camadas de abstração. Além disso, é aplicado na definição dos pacotes para separar em camadas os principais elementos daqueles que o utilizam.

- **Partições:** é aplicado para organizar áreas conceituais dentro das camadas. Isto fornece flexibilidade exigida pelos padrões atuais de metamodelagem. No metamodelo da UML, isto é representado pela forte coesão dentro dos pacotes e baixo acoplamento.

- **Extensibilidade:** é aplicado para tornar a linguagem extensível. A UML é estendida por duas formas: (i) criando um dialeto da linguagem através do uso de UML *profiles*; ou (ii) criando um nova linguagem que faz uso de parte da definição da UML.

- **Reuso:** a definição do metamodelo de forma granular facilita o reuso da definição da linguagem.

A infra-estrutura da UML é definida através da *InfrastructureLibrary* ilustrada na Figura 2.1, a qual possui dois pacotes *Core* e *Profiles*. O pacote *Core* trata-se de um metamodelo projetado para ter alto grau de reuso, a fim de que outros metamodelos importem ou especializem suas metaclasses. O pacote *Core* possui quatro pacotes: *PrimitiveTypes*, contém a definição de alguns tipos que são usados na definição de modelos; *Abstractions*, contém metaclasses abstratas as quais são especializadas ou são reusadas por outros metamodelos; *Basic* possui alguns elementos que são usados como base para produzir XMI (*XML Metadata Interchange*) para UML, MOF (*Meta Object Facility*) e outros metamodelos que são baseados na *InfrastructureLibrary*; e *Constructs*, contém metaclasses concretas que são usadas para modelar linguagem orientada a objetos, sendo usado pelo MOF e UML. *Core* é usado para definir os construtores usado na especificação de metamodelo, sendo usado para definir, por exemplo, a UML e o MOF.

O pacote *Profiles* tem uma dependência com *Core* e define o mecanismo usado para adaptar metamodelos existentes para uma plataforma ou domínio específico. O objetivo é adaptar a UML, porém é possível usá-lo para adaptar qualquer metamodelo que seja baseado no *Core*. Um dos objetivos da *InfrastructureLibrary* é promover o alinhamento arquitetural da UML e do MOF. Para isto, a UML e o MOF compartilham o pacote *Core*, e assegura que a UML (modelo) é definida baseada no MOF (metamodelo). Todo elemento definido na UML representa uma instância de um elemento definido no MOF.

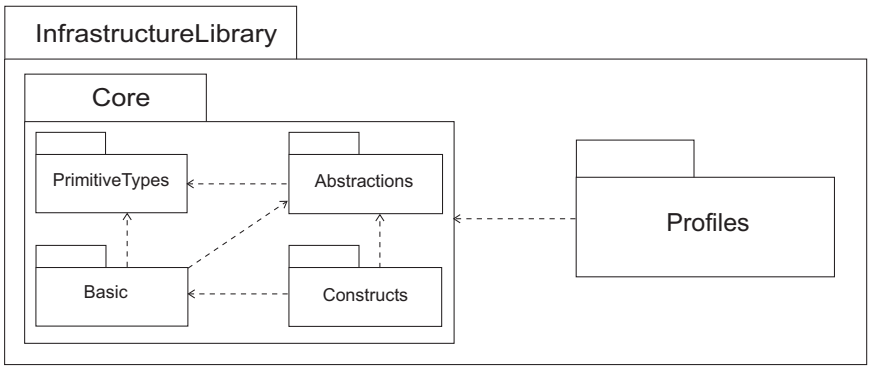

Figura 2.1: Pacote que define a infra-estrutura da UML (OMG 2007)

A definição do metamodelo da UML é estruturada em quatro camadas hierarquicamente organizadas. A camada de meta-metamodelagem (M3) tem o objetivo de especificar a linguagem utilizada para definir a UML, nesta camada encontra-se o MOF. A UML encontra-se especificada na camada de metamodelo (M2) que tem como objetivo definir linguagens para definir modelos. A camada de modelo (M1) contém instâncias dos metamodelos definidos em M2, representando os modelos do usuário. Esta estrutura é ilustrada na Figura 2.2. Os *profiles* tem uma característica especial que consiste em co-existir na mesma camada com os elementos que o definem, ou seja, eles são definidos pelos elementos presentes na camada M2 e também fazem parte desta, quebrando a ordem natural do processo de instanciação.

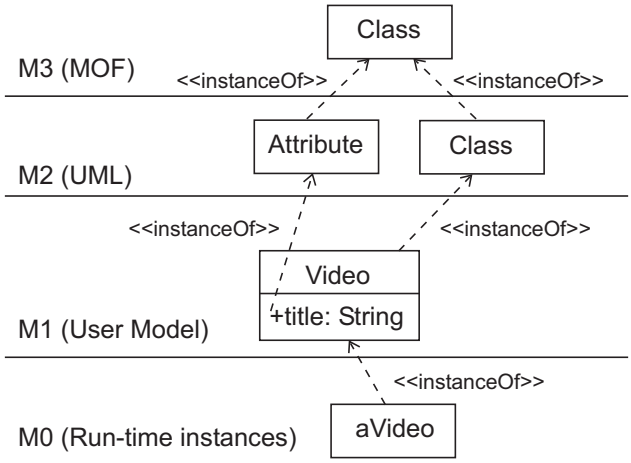

Figura 2.2: Estrutura de camada da UML (adaptado de (OMG 2007))

Pelo fato da UML ser uma linguagem de caráter geral e ter um escopo de aplicação muito grande, em determinada situações não é capaz de modelar aplicações definidas em domínio específico (por exemplo, área financeira, aplicações de tempo real, aplicações multimídia, entre outros). Isto caracteriza a situação onde a sintaxe ou a semântica especificada na UML não é capaz de expressar conceitos de um sistema em particular, ou

quando deseja-se tornar os elementos já existentes mais restritos, ou seja, diminuir o seu escopo. Sendo assim, surge a necessidade de criar um extensão da linguagem.

A OMG (OMG 2007*b*) define duas formas de definição de linguagem específicas de domínios. A primeira é baseada na definição de uma nova linguagem, representando uma linguagem alternativa a UML. Para isto, é utilizado os mecanismos fornecidos pela OMG para definição de linguagens visuais orientada a objetos. Esta abordagem representa a mesma forma utilizada para a definição da UML. Sendo assim, a sintaxe e a semântica dos elementos na nova linguagem são definidas para atender as características específicas do domínio (Fernández & Moreno 2004, OMG 2007*c*). Novas linguagens são construídas usando o MOF (OMG 2002), o qual consiste de uma linguagem projetada para a definição de linguagem de modelagem orientada a objetos. CWM (OMG 2001) representa um exemplo de linguagem criada com o MOF, onde a semântica dos elementos da linguagem não são equivalentes aos definidos na UML.

A segunda alternativa consiste na especialização da UML. Para isto, alguns elementos definidos da UML são especializados através da definição de restrições sobre os mesmos. Porém, com esta abordagem, o metamodelo da UML é respeitado, deixando a semântica original dos elementos da UML inalteradas (por exemplo, as propriedades de classes, operações e entre outros elementos permanecem inalteradas). Novas notações podem ser definidas a fim de uma melhor representação dos conceitos.

Com o objetivo de disponibilizar extensão conservativa, a UML fornece um conjunto de mecanismo de extensão (como *stereotype*, *constraints* e *tagged values*) para especializar seus elementos. Com isto, é possível gerar versões customizada da UML para atender às exigências de domínios específicos. Um vez definidas estas adaptações, estas são agrupadas em um UML *profile*.

A construção de uma linguagem baseada na adaptação da UML produzirá elementos e notações que atendem as exigências e as necessidades apresentadas pelos conceitos específicos de um domínio. Entretanto, como não é respeitado a semântica definida na UML, a extensão não permite o uso de ferramentas comerciais de modelagem para gerar os diagramas, gerar código, fazer engenharia reversa, e etc (Fernández & Moreno 2004, OMG 2007*c*, Rumbaugh et al. 2005). Por outro lado, os UML *profiles* são suportados pelas ferramentas comerciais de modelagem, porém podem não ser capazes de especificar determinadas necessidades e exigências. Definir qual tipo de extensão utilizar não é uma tarefa fácil, a qual dependerá das necessidades e da experiência com a UML. Segundo (Fernández & Moreno 2004) os benefícios do uso de *profiles* supera as suas limitações.

2.2 UML *Profiles* em Detalhes

O pacote *Profiles* (OMG 2007*c*) definido na *InfrastructureLibrary* contém mecanismos que permitem metaclasses de determinados metamodelos serem estendidas a fim de adaptá-las à diferentes propósitos. Como a UML é definida baseada no MOF, esta capacidade de adaptação é inserida no metamodelo da UML, tornando-a capaz de se ajustar as diferentes plataformas e aos domínios (Rumbaugh et al. 2005, OMG 2007*c*).

Para construir um *profile* alguns exigências devem ser satisfeitas, as quais são definidas em (OMG 2007*c*). A seguir as mesmas são apresentas:

- Todo *profile* deve ter algum metamodelo de referência ao qual o mesmo especializa. Uma nova semântica é definida, porém não deve ser contraditória a definida no metamodelo de referência.
- Um *profile* deve apresentar a capacidade de fazer referência à bibliotecas específicas de domínio da UML.
- Deve apresentar notações gráficas para representar os *stereotypes*.
- Ser capaz de especializar a semântica de determinados elementos do metamodelo da UML. Por exemplo, para representar um relacionamento de herança definido em Java, o *profile* restringe a herança múltipla definida na UML para uma herança única.
- É possível aplicar vários *profiles* simultaneamente em um modelo, logo a combinação destes *profiles* deve ser suportada.
- Um *profile* deve ser dinamicamente aplicado e removido de um modelo.
- Os profiles devem fornecer um mecanismo para especializar o metamodelo padrão da UML, de tal modo que a nova semântica não viole a semântica original do metamodelo. As restrições dos profiles podem normalmente definir regras de boa formação que são mais restritivas que (mas consistente com) as especificadas pelo metamodelo da UML, sendo possível definir formalmente (por exemplo em OCL) qualquer "semântica especializada" ou através de linguagem formal.
- Deve apenas utilizar os três mecanismos de extensão da UML, sendo eles: *stereotypes*, *tagged values*, e *constraints*.
- Deve ser possível realizar a troca de *profiles* entre a ferramentas de modelagem, juntamente com os modelos que os mesmos devem ser aplicados, por meio do uso dos mecanismos de troca da UML fazendo uso de XMI. Um *profile* deve, portanto, ser definido como um modelo UML permutável.

- Um *profile* deve ser capaz de fazer referência às bibliotecas da UML específicas de domínio, onde determinados elementos de modelos são pré-definidos. Este é o caso de C++ (Stroustrup 2000), IDL, Java e outros domínios técnicos. Por exemplo, UML profile para CORBA (OMG 2007*a*) precisa referenciar conjuntos pré-definidos de tipos básicos de IDL, os quais podem ser definidos na UML como um pacote de tipos de dados.

- deve ser possível executar certas operações sobre os *profiles* (por exemplo, criar relacionamentos entre *profiles*), principalmente:

  1. Especialização de *profiles*: obter um novo profile a partir de um *profile* existente (ou *profiles*), onde a semântica especializada não viole a semântica dos *profiles* pai. Por exemplo, o domínio de teste de software pode ser dividido em específicos sub-domínios, cada um destes sub-domínios podem exigir seus próprios *profiles* derivados de um *profile* comum de teste de software. Do mesmo modo, *profiles* customizados para plataformas ou aplicações específicas podem ser especializados a partir de um *profile* padrão.
  2. Composição de *profiles*: geração de um novo *profile* através da composição de dois ou mais *profiles* mutuamente compatíveis. Sendo esta operação o foco deste trabalho.

- A definição de *tagged values* deve ser formalizada. Em particular, *tagged values* são tipificadas com o objetivo de especificar exatamente quais valores são e quais as naturezas deles. O tipo de um *tagged value* deve ser algum tido de dados definido na UML, ou alguma classe ou *stereotype*, podendo ser um tipo de dado incorporado ou definido pelo usuário. Além disso, *tagged values* multivaloradas.

A UML trata-se de uma linguagem de modelagem de propósito geral que cobre um grande espectro de domínios de aplicação. Porém, quando aplicada para modelar domínios específicos apresenta algumas limitações diante da necessidade de representar conceitos específicos destes domínios. Sendo assim, uma especialização ou extensão dos conceitos definidos na UML permite adaptá-la a fim de torná-la capaz de representar peculiaridades em um domínio específico.

A extensão do metamodelo com o objetivo de aumentar o poder de representação da UML pode ser alcançado com o uso de *profiles*. Os *profiles* podem sofrer especializações com o objetivo de satisfazer as necessidade e as exigências de um subdomínio. Para isto, a UML fornece mecanismos de refinamento, são eles: (*i*) *tagged values*; (*ii*) *stereotypes*; e *constraints*. Estes elementos podem ser usados para adaptar a semântica da UML, sem alterar o metamodelo da UML de fato. Uma vez definidos são agrupados em um *profile*.

Stereotype

Stereotype define como uma existente metaclasse pode ser estendida com o objetivo de modificar modificando de alguma forma a sua semântica. Adiciona ou substitui terminologias ou notações já existentes com o objetivo de gerar adaptações. Na sua definição no metamodelo, um *stereotype* trata-se de uma construtor que estende a metaclasse *Class*.

A definição dos *stereotypes* no metamodelo da UML segue o mesmo padrão da definição dos outros construtores. Porém, a criação de um *stereotype* tem um peculiaridade, pois a sua instância não caracteriza a criação de um "modelo de usuário" (ver Figura 2.2), mas sim um novo construtor inserido na definição do metamodelo. Em outras palavras, um *profile* é criado a partir da M2 e sua instância permanece na M2, esta co-existência do metamodelo e da sua instância representa uma peculiaridade dos *profiles*. Os *profiles* são vistos como uma "camada virtual" na camada M2. A fim de ter uma melhor representação é possível associar imagens aos *stereotypes*. A Figura 2.3 ilustra o uso de *stereotypes*.

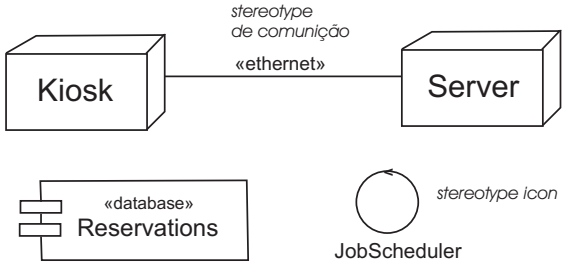

Figura 2.3: Uso de *stereotypes*

Tagged Value

Na UML 1.3, *tagged values* poderiam estender o metamodelo sem a necessidade de ter *stereotype* associado. Na UML 2.0, uma *tagged value* pode ser utilizada apenas associada a um *stereotype* e sua estrutura é basicamente formada por "nome" e "tipo". Sendo assim, para fazer uso de *tagged value* é necessário primeiro definir um *stereotype* e depois inserir a *tagged value* no *stereotype*.

Constraints

Uma *constraints* trata-se de uma restrição semântica representada como uma expressão textual. Cada expressão tem uma linguagem de interpretação implícita (Rumbaugh et al. 1999), a qual pode ser uma notação de matemática formal, por exemplo a notação de teoria dos conjuntos; linguagem de restrição baseada em computação, por exemplo OCL; um linguagem de programação, tais como Java; ou linguagem natural informal.

O objetivo do uso das *constraints* consiste na especificação de restrições que não são possíveis de serem representadas usando a sintaxe abstrata da UML. As restrições

são representadas através de *string's* entre chaves, podem ser fixadas em uma lista de elementos, entre dependências, ou dentro de um nota. Além disso, estas *constraints* tem o papel fundamental na definição de regras de boa formação, as quais devem ser respeitadas para a construção de um modelo válido. A Figura 2.4 ilustra de forma simples o uso de *constraints*.

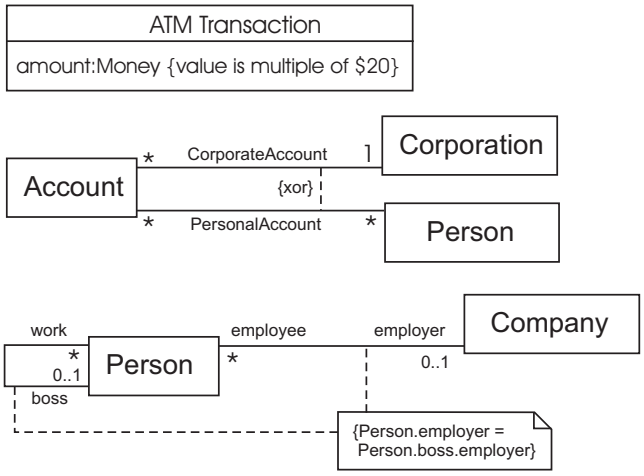

Figura 2.4: Exemplo de uso de *constraints*

Existem várias razões para fazer o uso dos *profiles*, tais como: (i) fazer uso de terminologias que são adaptadas ao uma particular domínio ou plataforma; (ii) atribuir notações diferentes ou alternativas para elementos definidos no metamodelo de referência; (iii) adicionar semântica que não são especificadas; (iv) adicionar restrições limitando o uso do metamodelo e de seus construtores; (v) inserir informações que podem ser usadas quando o modelo é transformado em outro modelo ou em código.

2.2.1 Propósito dos UML *Profiles*

O propósito dos UML *profiles* é permitir a construção e permutação de modelos UML que exijam especificação semântica, além das quais podem ser representados com a UML padrão. Observando que as semânticas adicionadas são completamente consistentes com a semântica geral da UML. Por exemplo, novas restrições associadas aos *stereotypes* não podem contradizer as restrições em OCL do metamodelo da UML, mas podem torná-las mais restritas ou especializadas, realizando uma extensão conservativa.

A seguir são mostradas duas possíveis aplicações dos *profiles*:

- **Especializar a UML para determinadas tecnologias:** linguagem de programação, *middleware*, ou banco de dados são naturais candidatos a definir um específico UML *profile*. Notações que são específicas das linguagens são introduzidas como estensões, tais como: "métodos virtuais", "identificadores", restrição "*not null*", etc.

Alguns restrições podem ser especificadas nos *profiles*, com o objetivo de fornecer algumas regras.

- **Especializar a UML para modelagem de processo de desenvolvimento de software:** a UML é utilizada de forma diferentes nas fases do desenvolvimento do software. Dessa forma, os *profiles* podem ser utilizados para especificar particularidades do processo de desenvolvimento. Por exemplo, durante a fase de análise é possível expressar com *profiles* que um ator deve interagir com ao menos um caso de uso.

A definição de um *profile* pode apresentar regras de boa formação em adição as já especificas pelo metamodelo da UML com o propósito de especializar a semântica de determinado elemento do metamodelo. Regras de boa formação é o termo utilizado na especificação do metamodelo da UML para descrever um conjunto de restrições especificadas em OCL, ou em linguagem natural, que contribui para a definição da semântica dos elementos do metamodelo.

2.3 Mecanismo de Composição da UML

A UML apresenta um mecanismo de composição, *package merge*, que define como dois pacotes são compostos. O *package merge* é aplicado no metamodelo da UML a fim de tornar possível a implementação dos *compliance levels* definidos na mesma. Este mecanismo define terminologias e uma semântica para o relacionamento de composição entre dois pacotes. O uso do *package merge* é indicado quando elementos definidos em pacotes diferentes tem nomes iguais e se deseja que estes representem um mesmo conceito (OMG 2007*c*). É aplicado para fornecer diferentes definições de um dado conceito para diferentes propósitos, iniciando com uma definição base. Dado um conceito base, este é estendido em incrementos, com cada incremento definido em um *merged package* diferente. Através da seleção do incremento que deve ser composto é possível obter uma definição de um conceito para um determinada finalidade.

Pela natureza da atividade desenvolvida pelo *package merge*, este pode ser visto como uma operação que tem dois pacotes de entradas e gera um pacote de saída que representa a combinação do conteúdo dos pacotes envolvidos na composição (OMG 2007*c*). Além disso, os pacotes de entrada e o de saída possuem semântica equivalente.

O *package merge* trata-se de um *DirectedRelationship* (OMG 2007*c*) entre dois pacotes representado pela palavra-chave "*merge*". O pacote de origem do relacionamento é definido como *receiving package*; e o destino como *merged package*. O resultado da composição é representado pelo *resulting package*. Na Figura 2.5 é mostrado um exemplo simples do uso do *package merge*.

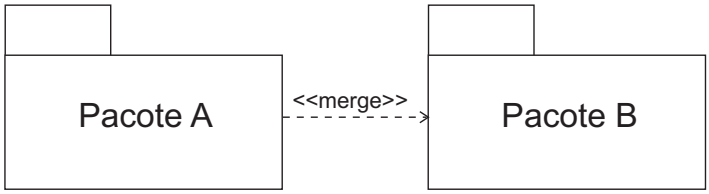

Figura 2.5: Exemplo de uso do *package merge*

O *package merge* é definido basicamente com *match rules, constraints* e *transformations*. As *match rules* são utilizadas para especificar quando um elemento do pacote *receiving* e um do *merged* representam conceitos equivalentes, devendo ser compostos para representar um elemento único no *resulting package*. A comparação dos elementos dos pacotes é baseada no "nome" dos mesmos.

As *constraints* especificam as pré-condições que devem ser satisfeitas para tornar a composição possível. Em determinada situação é possível ter dois elementos que são considerados equivalentes, porém não podem ser compostos devido à violação de algum restrição. Sendo assim, a composição destes elementos pode produzir algum elemento inválido. Em geral, a composição é considerada inválida se os elementos possuem alguma incompatibilidade. Por exemplo, a composição de duas operações é considerada inválida se apresentam algum inconformidade, o fato de as duas operações terem nomes iguais, porém com tipo de retorno diferente invalida a composição. Outro possível exemplo seria dois elementos considerados equivalentes, porém com restrições contraditórias também invalidam a composição.

As *transformations* definem como a composição é realizada de fato. Estas definem a pós-condição da composição, ou seja, qual o resultado esperado da composição, se todas as restrições são satisfeitas. Como regra padrão para realizar a composição é definido que se um elemento do *merged package* não possui um elemento equivalente no *receiving package*, este elemento deve ser copiado para o *resulting package*. O *package merge* não apresenta *transformations* para todos os metatipos presentes no metamodelo, apresentando apenas para os mais comuns (como *Class*, *Association*, *Properties*, e entre outros). Se dois elementos são iguais, o elemento resultante da composição é também igual a eles. Por outro lado, se os elementos são diferentes a *transformation* define como estes elementos são inseridos no *resulting package*.

Capítulo 3

Análise Comparativa das Abordagens de Composição de Modelos

Este seção tem como principal objetivo apresentar os principais trabalhos relacionados à composição de modelos utilizados no contexto desta pesquisa. Analisando a literatura atual, é possível observar que a composição de modelos tem sido objeto de pesquisa na comunidade de Engenharia de Software em diferentes contextos. Porém, foi constatado que por ser uma área de pesquisa nova, conseqüentemente, tendo trabalhos recentes e em processo de desenvolvimento, porém nenhum com foco na composição de UML *profiles*. Sendo assim, o seu desenvolvimento tem se baseado em trabalhos em diversas áreas de pesquisa, como: (i) integração de banco de dados (Bernstein & Melnik 2007, C.Parent & Spaccapietra 1998, Batini et al. 1986*b*, Batini et al. 1986*a*); (ii) modelagem orientada aspecto (Baudry et al. 2005, Reddy et al. 2006, Reddy et al. 2005); (iii) composição de código fonte (Tempero & Biddle 2000, Simons 2004); (iv) transformação de modelos (OMG 2003, Bézivin 2001, France & Rumpe 2007).

Como o principal interesse desta pesquisa está relacionado à identificação das necessidades e na construção de um mecanismo que atenda às peculiares dos UML *profiles*, serão analisados, a seguir, os trabalhos que contribuíram para esta pesquisa. A análise comparativa das abordagens de composição de modelos é baseada nos critérios listados abaixo:

1. **Etapas da composição:** quais são as etapas necessárias para realizar a composição? Tem um fluxo de atividades?

2. **Forma de verificar equivalência:** como é realizada a comparação dos modelos de entrada? O que é levado em consideração?

3. **Forma de realizar a composição:** o que é utilizado para realizar a composição? Possui formas diferentes de realizar a composição?

4. **Tratamento de problemas:** os problemas que surgem da composição são tratados? Como são tratados os conflitos gerados?

5. **Representação da proposta:** a proposta apresenta algum metamodelo? É possível expressar a composição em UML?

6. **Escopo da abordagem:** a proposta compõe qual tipo de modelo?

7. **Validação da abordagem:** como a abordagem é validada? Quais avaliações são feitas?

8. **Relação com a abordagem proposta:** relação entre as abordagens e a proposta apresentada.

Abordagem (Reddy et al. 2005)

Etapas da composição: a abordagem apresentada em (Reddy et al. 2005) contempla as etapas de comparação e composição dos modelos. Poucas considerações são feitas sobre quais atividades são necessárias para realizar a comparação e a composição, não ficando explícito como os modelos de entrada são manipulados para compor suas estruturas.

Forma de verificar equivalência: a proposta apresenta um processo de composição baseado em assinatura, onde uma assinatura consiste de um conjunto de propriedades sintáticas do modelo. Para esta abordagem, dois modelos são equivalentes, se, e somente se, tiverem as propriedades definidas nas assinaturas iguais. Porém, considerado apenas a assinatura como elemento de comparação torna o processo de comparação deficiente. Por exemplo, modelos que possuem nomes diferentes podem representar conceitos iguais.

Forma de realizar a composição: nenhuma consideração é feita a respeito como as propriedades dos modelos devem ser modificadas para expressar a composição. É apresentado um algoritmo que realiza a composição de dois modelos. Porém nada é exposto sobre como os valores das propriedades dos modelos que apresentam valores diferentes são tratadas. Por exemplo, como duas classes com nomes iguais, porém com visibilidades diferentes são compostas. Logo, como compor dois métodos sendo um de visibilidade *public* e o outro de visibilidade *private*? Questão deste tipo não pode ser respondida.

Tratamento de problemas: Nenhuma consideração é feita explicitamente no trabalho. Porém, é feito referência à diretivas de composição como as responsáveis pelo tratamento de conflitos.

Representação da proposta: a abordagem apresenta um metamodelo de composição que descreve como realizar a composição baseada em assinatura. Para isto apresenta propriedades estáticas e comportamentais para dar suporte a composição de modelos orientados a aspectos. Porém, o metamodelo foi idealizado com o objetivo de ser usado no

desenvolvimento da ferramenta de composição, não para expressar a abordagem proposta. Além disso, não apresenta uma instância do metamodelo expressando o uso do mesmo.

Escopo da abordagem: a proposta destina-se a composição de modelagem orientada a aspectos. Ou seja, composição dos aspectos que representam os *cross-cut concerns* e os *primary model*.

Validação da abordagem: para validar a proposta é utilizado a linguagem *Kermeta* (Muller et al. 2005, Triskell 2005) para desenvolver uma ferramenta que implementa a técnica de composição baseada em assinatura.

Relação com a abordagem proposta: apesar desta abordagem se basear na comparação por assinatura, existe uma grande distância entre a mesma e a apresentada nesta dissertação. A forma de comparação não apresenta flexibilidade na maneira de comparar e apresenta dificuldades na realização de verificação de novos tipos de modelos, por exemplo, um modelo do tipo *enumeration*. A extensão do metamodelo da UML não apresenta uma definição semântica e nem regras de boa formação, limitando-se a uma descrição em linguagem natural, ao contrário da apresentada neste trabalho de dissertação. Além disso, não apresenta uma especificação formal da abordagem .

Abordagem (Clarke 2001)

Etapas da composição: a abordagem apresentada em (Clarke 2001) dá suporte a comparação, composição e tratamento de conflitos. Porém, não apresenta uma descrição de como estas "atividades" se relacionam e não descreve um fluxo entre as atividades.

Forma de verificar equivalência: a proposta apresenta um processo de composição baseado em nomes, o que representa uma desvantagem da proposta. Pois comparação de modelos baseada apenas em "nome", não é refinada o bastante para identificar a similaridade dos conceitos representados pelos modelos. Por exemplo, dois atributos com nomes iguais e tipos diferentes serão compostos pela abordagem, o que representa um problema. Por exemplo, dois atributos sendo de tipo inteiro e outro do tipo *String* são compostos sem nenhuma consideração.

Forma de realizar a composição: esta abordagem apresenta duas formas de realizar composição, sendo representado por: *override* e *merge strategy*. Apesar desta abordagem ser uma das mais completas na especificação de diferentes formas de realizar composição, a mesma apresenta limitadas regras de composição para as estratégias em linguagem natural. Nada é exposto como os valores das propriedades dos modelos que apresentam valores diferentes são tratadas. Por exemplo, como duas classes com nomes iguais, porém com propriedades (*isAbstract* = *true* e *isAbstract* = *false*) diferentes são compostas.

Tratamento de problemas: apresenta tratamento de apenas quatro tipos de confli-

tos: *precedence*, especifica a precedência entre os elementos correspondentes; *explicit*, conflito tratado em nível do modelo composto cujo objetivo é especificar uma referência explícita a outro modelo; *default*, especifica valores padrão para determinada propriedades dos modelos; *TransformationFunction*, especifica uma função para definir a especificação da reconciliação entre modelos correspondentes.

Representação da proposta: é apresentado uma extensão da UML e a sua construção foi construída seguindo o padrão apresentado na UML, definindo: (i) sintaxe abstrata; (ii) regras de boa formação; (iii) semântica. A extensão apresenta conceitos que também são encontrados nesta extensão do metamodelo da UML apresentada nesta dissertação, como: *CompositionRelationchip*; *ComposableElement*; e *CompositeElement*. Porém, apresentam semânticas diferentes.

Escopo da abordagem: composição de *subjects* (Harrison & Ossher 1993).
Validação da abordagem: nenhuma consideração é feita sobre a validação utilizando algum formalismo ou através da construção de ferramenta.

Relação com a abordagem proposta: apesar da abordagem (Clarke 2001) ser bem estrutura e tratar pontos importantes na composição de modelos, a mesma apresenta desvantagens em relação as questões tratadas nesta dissertação. O processo de comparação apresentado nesta dissertação tem maior poder de identificação de equivalência entre os modelos, justificado pelas diferentes pontos considerados na momento da comparação, como: tipografia, estrutura do modelo e a semelhança entre conceitos do domínio. Esta abordagem, além disso, não apresenta nem nenhuma verificação formal, nem ferramenta que faça uma validação conceitual da abordagem.

Abordagem (Zito 2006)

Etapas da composição: a abordagem apresentada em (Zito 2006) tem como objetivo retirar ambiguidade e inconsistência no *package merge* (OMG 2007*c*). Além disso, visa dar suporte a comparação e composição. Porém, não apresenta uma descrição da interação entre as duas etapas presentes.

Forma de verificar equivalência: a abordagem usa apenas o "nome" dos modelos para definir a correspondência entre eles, caracterizando uma desvantagem da abordagem, sendo equivalente a (Clarke 2001) neste sentido. Para realizar a comparação são utilizados regras de correspondências, que se comparam com as *default match rules* descritas nesta dissertação. Apesar da presença das regras de correspondência, fica distante da flexibilidade e o poder de definição de similaridade apresentado nesta dissertação.

Forma de realizar a composição: esta abordagem se caracteriza por apresentar regras de composição considerando as propriedades sintáticas definidas no metamodelo da UML,

caracterizando um ponto forte da abordagem. Porém, estratégias de composição não são apresentadas, sendo assim, apenas uma forma de composição é possível de ser realizada, sendo esta forma comparável à realizada pela *merge strategy* apresentada neste trabalho de dissertação.

Tratamento de problemas: nenhuma consideração é feita sobre os problemas que surgem com a composição.

Representação da proposta: não é apresentado nenhuma extensão do metamodelo da UML, porém é representado por um *DirectedRelationship* representado por com $\ll merge \gg$.

Escopo da abordagem: composição de diagrama de classes UML e pacotes.

Validação da abordagem: a validação da proposta é realizada através de uma verificação formal. Para isto, é utilizado a linguagem *Alloy*. Além disso, foi desenvolvido uma ferramenta utilizando como base o EMF (Eclipse Project 2007, Budinsky et al. 2003) e integrada ao *Rational Software Architecture*.

Relação com a abordagem proposta: (Zito 2006) apresenta desvantagens em relação a questão de definição de equivalência, devido à forma de realizar a comparação baseado apenas no nome. Assim como o trabalho apresentado nesta dissertação, é apresentada uma modelagem *Alloy* e uma ferramenta visando a validação conceitual da proposta. As regras de composição desta abordagem foram tomadas como base para a elaboração das diferentes regras de composição, visando implementar as semânticas definidas nas estratégias de composição. .

Abordagem (Reddy et al. 2006)

Etapas da composição: a abordagem apresentada em (Reddy et al. 2006) considera a comparação, composição e o tratamento de conflitos como atividades essenciais na composição de modelos orientado a aspecto. É enfatizado como deve ser realizada a comparação, composição e o tratamento de conflitos. Além disso, é apresentado um fluxograma das atividades necessárias para solucionar problemas relacionados a composição através de diretivas de composição.

Forma de verificar equivalência: para obter uma visão integrada dos modelos orientados a aspecto, é considerado nomes dos modelos como base para determinar a equivalência entre os mesmo. Além disso, é utilizado o conceito de "assinatura" (conjunto de propriedades sintáticas que devem ser consideradas para realizar a comparação) a fim de melhorar o processo de definição de equivalência. Pois é possível verificar a estrutura do modelos usando assinaturas. Dois modelos são considerados equivalentes se, e somente se, tiverem assinaturas iguais, porém para modelos com nomes diferentes e que representam um mesmo conceito, a abordagem não é capaz de identificá-los como equivalentes. Sendo

assim, a não consideração do valor semântico dos modelos configura uma limitação da abordagem.

Forma de realizar a composição: esta abordagem utiliza diretivas de composição para realizar a composição, sendo esta representada como um empacotamento de modelos primários (representam os *core concerns*) e aspectuais (representam os *crosscut concerns*). Para realizar a composição, modelos com assinaturas equivalentes representam visões consistentes de um mesmo conceito, logo, devem ser compostos. Estas diretivas são especificadas usando um formato pré-definido.

Tratamento de problemas: o tratamento de problemas que podem surgir durante a composição é considerado pela abordagem com um atividade fundamental. Para isto, são definidas diretivas de composição (*post-merge directives*) com o objetivo de realizar modificações no modelo visando solucionar os conflitos e os problemas encontrados no modelo composto.

Representação da proposta: um metamodelo de composição é definido, o qual apresenta propriedades estáticas e comportamentais associadas a composição de modelos. Para isto, é utilizado a *Kermeta* (Muller et al. 2005, Triskell 2005), uma linguagem de metamodelagem que extende o MOF com linguagem de ações, sendo compatível com o EMF. Dentro os conceitos apresentados, têm-se: *Mergeable*, representa os modelos que podem ser compostos; *Signature*, instância deste elemento define representações de assinaturas.

Escopo da abordagem: composição de modelos orientado a aspectos.

Validação da abordagem: para validar a proposta é utilizado a linguagem *Kermeta* para implementar a parte principal do metamodelo. Porém, não apresentando, por exemplo, a implementação das diretivas de composição.

Relação com a abordagem proposta: assim como o trabalho apresentado nesta dissertação, (Reddy et al. 2006) utiliza assinatura na comparação entre os modelos. Porém, apenas a utilização das assinaturas para definir a equivalência entre os modelos configura uma desvantagem. Os objetivos delegados às diretivas de composição, se assemelham com aos das regras de composição e transformação de modelos utilizadas pelo *merge operator* e *model transformation operator*, respectivamente. O metamodelo se diferencia do metamodelo da UML e do metamodelo apresentado no Capitulo 6 por não definir a composição como um relacionamento entre dois modelos (conforme o conceito de *relationship* especificado no metamodelo da UML (OMG 2007*c*)), mas como um comportamento *Mergeable::merge()*, o qual todo modelo adquire quando estende a classe abstrata *Mergeable*.

Abordagem (Brunet et al. 2006)

Etapas da composição: a abordagem apresentada em (Brunet et al. 2006) especifica

a necessidade da realização de comparação e composição efetivamente. Para isto, são definidos um conjunto de operações que realizam estas atividades.

Forma de verificar equivalência: para definir a correspondência entre os modelos, é especificado a necessidade de um operador de *match*. Porém, não especifica como a comparação e a definição da equivalência deve ser determinada.

Forma de realizar a composição: não é especificado como a composição deve ser realizada. É definido a necessidade de um operador de composição para tornar a composição de modelos possível, o qual gera um modelo composto a partir de dois modelos de entradas e um definição de relacionamento entre eles.

Tratamento de problemas: nenhuma consideração é feita em relação a problemas e a conflitos que surgem durante a composição.

Representação da proposta: a proposta é apresentada através de uma descrição de um conjunto de operadores de composição. Nenhum consideração é feita em relação a sua representação no contexto da UML.

Escopo da abordagem: a proposta visa ser aplicável a todos os domínios de aplicação de composição modelos.

Validação da abordagem: nenhuma validação é realizada, apenas exemplos mostrando aplicabilidade da abordagem são descritos.

Relação com a abordagem proposta: (Brunet et al. 2006) consiste de um *framework* unificado para o entendimento dos problemas da composição de modelos e visa fornecer um vocabulário para a discussão das idéias centrais em composição. Os operadores de *match* e *merge* descritos na abordagem tem objetivos iguais aos dos *match* e *merge operator* descritos no Capitulo 4. As propriedades algébricas definidas em (Brunet et al. 2006) são usadas na validação formal do trabalho proposto nesta dissertação, tais como: idempotênica; comutatividade; associatividade; inversa; monotonicidade; e totalidade.

Abordagem (Nejati et al. 2007)

Etapas da composição: a abordagem apresentada em (Nejati et al. 2007) tem foco na comparação e composição.

Forma de verificar equivalência: é especificado um operador de *match* para encontrar correspondência entre os modelos. Para determinar a equivalência, o *match operator* faz uso de heurísticas para encontrar a similaridade tipográfica, estrutural e semântica entre os modelos. O objetivo da comparação é gerar uma tabela de similaridade que especifica o grau de correspondências entre os elementos de duas *statecharts*.

Forma de realizar a composição: para realizar a composição é definido um operador

de composição. Este operador tem dois modelos de entrada e uma descrição do relacionamento de correspondência entre os mesmos. O objetivo do operador é construir um modelo composto que contenha os comportamentos compartilhados pelos modelos de entradas, como um comportamento normal, e os não compartilhados como variabilidades. A composição preserva a semântica dos modelos de entrada, o que representa um ponto muito positivo desta abordagem.

Tratamento de problemas: nenhuma consideração é feita em relação aos problemas e aos conflitos que surgem durante a composição.

Representação da proposta: nenhuma consideração é feita em relação a representação da proposta como uma extensão da UML.

Escopo da abordagem: composição de *statecharts*.

Validação da abordagem: para avaliar os operadores propostos foi desenvolvido a ferramenta TReMer (Sabetzadeh et al. 2006), a qual compõe *statecharts* dado um correspondência entre elas. Além disso, é analisado a complexidade da realização de um comparação considerando aspectos estáticos ($O(n_1 \times n_2)$, onde n_1 e n_2 representam o número de estados nos modelos de entradas) e os comportamentais ($O(c \times m_1 \times m_2)$, onde c é o número máximo de interações para o algoritmo de comparação de comportamento, e m_1 e m_2 representam o número de transição nos modelos de entrada)

Relação com a abordagem proposta: a relação com a abordagem proposta nesta dissertação consiste na forma de comparar dois modelos através de similaridade tipográfica. Não é considerado aspectos comportamentais como em (Nejati et al. 2007), nem conservação da semântica dos modelos de entrada não é garantida no modelo de saída. Como semelhança têm-se: (i) as atividades de comparação e composição são desempenhadas por operadores; (ii) usa-se similaridade tipográfica e lingüística; (iii) representação da correspondência entre os modelos de entrada através de uma tabela de similaridade.

Abordagem (Estublier & Ionita 2005)

Etapas da composição: a abordagem apresentada em (Estublier & Ionita 2005) define apenas a comparação e a composição como etapas importante para a representação da composição de modelos.

Forma de verificar equivalência: nenhuma consideração é realizada quanto a forma de se realizar a comparação, apenas que a comparação e a definição de correspondência é necessária. A correspondência é representada através do *stereotype ConceptOverlap*. A correspondência entre *features* de classes pertencentes a domínios diferentes é representado com o *stereotype FeatureCorrespondence*.

Forma de realizar a composição: a composição é representada como uma associa-

ção entre conceitos de diferentes domínios, tendo com alguns deles partes equivalentes. A composição de visões estáticas de diferentes domínios visa formar uma linguagem de meta-modelagem que seja capaz de modelar um domínio ou a composição destes domínios.

Tratamento de problemas: nenhuma consideração é feita em relação aos problemas e aos conflitos que surgem durante a composição.

Representação da proposta: (Estublier & Ionita 2005) define um extensão do meta-modelo da UML capaz de representar sua modelagem e a sua composição. Ou seja, uma linguagem de meta-modelagem específica de domínio.

Escopo da abordagem: composição de modelos em *software federation*.

Validação da abordagem: nenhuma forma de validação é apresentada.

Relação com a abordagem proposta: a relação com a abordagem proposta nesta dissertação consiste na apresentação de uma extensão do metamodelo da UML para representar composição de modelos, porém estas extensões possuem pouca semelhança.

3.1 Resultado da Análise

Com o objetivo de comparar a abordagem definida nesta dissertação com outros trabalhos, foi realizado uma análise comparativa anteriormente descrita. Sendo assim, agora é apresentado o resultado desta análise das abordagens de composição de modelos. Na Figura 3.1, é ilustrado um resumo desta análise comparativa tendo como base os critérios de comparação definidos anteriormente.

A partir da Figura 3.1 observa-se que:

- **Etapas:** nenhuma das abordagens estudadas fornecem um guia de composição de modelos. Todas as abordagens tem a comparação e composição como atividades fundamentais.

- **Comparação:** nenhuma abordagem apresenta estratégias de comparação. As abordagens não combinam diferentes maneiras de realizar comparação a fim de melhor definir a correspondência entre os modelos de entrada.

- **Composição:** alguns abordagens não deixam claro como a composição é realizada efetivamente. Apenas uma abordagem estudada apresenta a definição de estratégias de composição e nenhuma abordagem apresenta simultaneamente estratégias de composição, regras de composição e operador de composição.

		Reddy et al. 2005	Clarke 2001	Zito 2006	Reddy et al. 2006	Brunet et al. 2006	Nejati et al. 2007	Estublier & Ioneta 2005
Repres.	Semântica		X					
	Reg. Boa Form.		X					X
	Sint. Abstrata	X	X		X			X
Val.	Ferramenta	X		X	X		X	
	Formal			X				
Escopo	Outros			X			X	
	Metamodelo					X		X
	Class	X	X	X	X	X		
Trans.	Operador					X		
	Regras				X			
Comp.	Operador					X	X	
	Regras		X	X				
	Estratégias		X					
Comparação	Operador					X	X	
	Regras			X				
	Semântica						X	
	Estrutural				X			
	Nome	X	X	X				
	Tipográfica						X	
	Estratégias							
Etapas	Guia							
	Transformação		X		X			
	Composição	X	X	X	X	X	X	X
	Comparação	X	X	X	X	X	X	X
	Análise				X			

Abordagens

Legenda:
X – Suportado pela Abordagem
Comp. – Composição
Trans. – Transformação de Modelos
Val. – Validação
Repres. – Representação

Figura 3.1: Tabela comparativa dos mecanismos de composição de modelos

- **Transformação de Modelos:** apenas duas abordagens apresentam a preocupação em tratar problemas que surgem durante a composição, porém apenas uma delas faz uso de algo semelhante as regras de transformação de modelos definidas nesta dissertação.

- **Escopo:** boa parte das abordagens se concentram na composição de classes, nenhuma dedicada à composição de *profiles*.

- **Validação:** apenas uma abordagem faz uso de linguagem formal para validação da abordagem, sendo a maioria validada através de ferramentas. Porém, alguns não apresentam nenhum forma de validação, sendo a maioria tendo seus resultados questionáveis.

- **Representação:** verifica-se uma parcial preocupação em adaptar a UML a fim de torná-la capaz de representar composição de modelos. Apenas uma abordagem faz uso de sintaxe abstrata, regras de boa formação e definição de uma semântica para definir a extensão.

Capítulo 4

Definição do Mecanismo de Composição de UML *Profiles*

O problema central resolvido neste dissertação consiste em fornecer um mecanismo de composição de UML *profiles*. Neste sentido, o Capítulo 1, apresenta uma especificação dos problemas encontrados na composição dos UML *profiles*, as conseqüências e os aspectos negativos que a ausência deste mecanismo proporciona. Sendo assim, neste capítulo é apresentado o mecanismo de composição de modelos com o objetivo de solucionar tais problemas. Esta abordagem é baseada em quatro operadores: (i) *Match Operator*, responsável por definir a equivalência entre os modelos de entradas do mecanismo de composição; (ii) *Merge Operator*, responsável por realizar a composição dos modelos; (iii) *Composition Strategy Operator*, responsável por definir a estratégia de composição a qual a composição deve seguir; (iv) *Model Transformation Operator*, responsável por solucionar problemas e conflitos que surgem durante a composição dos modelos de entrada.

Desse modo, a proposta para tornar possível a composição de UML *profiles* é descrita e detalhada nas seguintes seções:

- *Um Semântica para Composição de UML Profiles:* com a deficiência na definição da semântica de composição de modelos na especificação da UML (OMG 2007*c*), tem-se como objetivo definir uma semântica que torna possível especificar a composição dos *profiles*.

- *Especificando os Modelos de Entrada:* nesta seção são especificados os modelos de entrada do mecanismo de composição.

- *Problemas Encontrados na Definição do Mecanismo de Composição da UML:* nesta seção são apresentados alguns problemas encontrados no mecanismo de composição da UML juntamente com a solução.

- *Definição dos Operadores de Composição:* esta seção se detém na definição dos operadores de composição. Cada operador é responsável por um conjunto de atividades,

sendo o conjunto destas atividades a representação do mecanismo de composição na sua essência.

- *Um Guia para Composição de Modelos:* nesta seção é definido como os UML *profiles* são compostos. Para isto foi definido um guia de composição de modelos, o qual define um alinhamento de execução das atividades delegadas aos operadores.

4.1 Uma Semântica para Composição de UML *Profiles*

No momento da composição de dois *profiles*, os mesmos devem ser identificados com o objetivo de evitar interpretações erradas e com o objetivo de definir o papel de cada modelo no mecanismo de composição. Dessa forma, a definição de uma semântica para o mecanismo de composição é fundamental para tornar o processo de composição bem definido (Oliveira & Oliveira 2007*a*).

Com isso, é apresentado uma semântica de composição baseado na definição apresentada na especificação da UML 2.1 (OMG 2007*c*). Desse modo, o mecanismo de composição é representado através de um "relacionamento de composição" (*composition relationship*) entre dois *profiles*. A Figura 4.1 mostra a semântica de composição de *profiles*, na qual são definidos:

1. *receiving profile:* é o primeiro operando da composição. Este *profile* terá seu conteúdo composto com o do *merged profile*.

2. *merged profile:* é o segundo operando da composição. Ele é composto com o *receiving profile*.

3. *resulting profile:* este termo é usado para especificar o *profile* que foi obtido através do mecanismo de composição.

4. *merged element:* faz referência ao conteúdo dos *merged profile*. Ou seja, este termo é usado para especificar e fazer referência a todos os elementos que o *merged profile* possui.

5. *receiving element:* faz referência ao conteúdo dos *receiving profile*. Ou seja, este termo é usado para especificar e fazer referência a todos os elementos que o *receiving profile* possui.

6. *resulting element:* faz referência ao conteúdo dos *resulting profile*. Ou seja, este termo é usado para especificar e fazer referência a todos os elementos que o *resulting profile* possui.

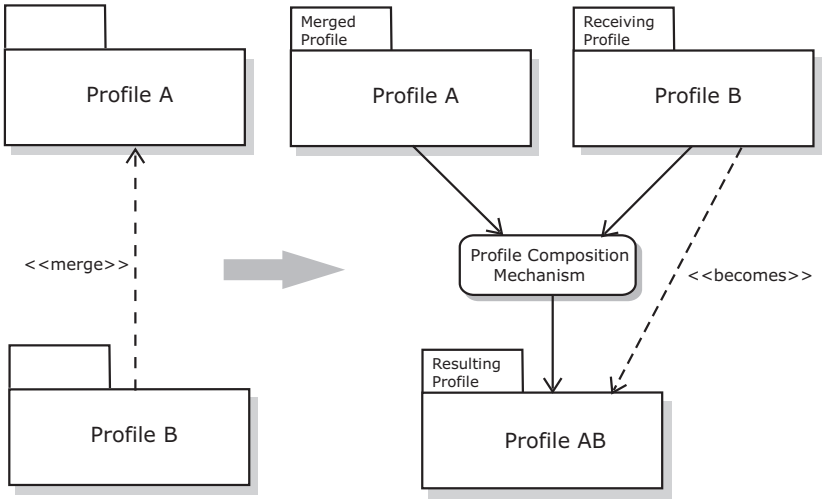

Figura 4.1: Semântica de composição de *profiles* (adaptada de (OMG 2007))

No contexto de orientação a objetos, tendo duas classes, *Classe A* e *Classe B*, quando é estabelecido um relacionamento de herança da *Classe B* em relação à *Classe A*, isto implica que os atributos e métodos da *Classe A* devem ser inseridos (herdados) na *Classe B*. Observa-se que quando é estabelecido o relacionamento de herança, a *Classe A* assume o papel de superclasse e a *Classe B* o papel de subclasse. Desse modo, qualquer referência realizada à *Classe B* (agora como subclasse), na verdade, implica em fazer referência a todos os atributos e métodos definidos tanto na *Classe A* (agora como superclasse) quanto na *Classe B* (observando que alguns atributos e métodos da *Classe A* podem ser sobrescritos).

Fazendo um paralelo entre a semântica do mecanismo de herança da orientação a objetos e a semântica apresentada nesta seção, tem-se: (i) o papel da superclasse é desempenhado pelo *merged profile*; (ii) o papel da subclasse é desempenhado pelo *receiving profiles* e pelo *resulting profile*. Na Figura 4.2, é mostrado um paralelo entre a semântica do relacionamento de herança do paradigma orientado a objetos e a semântica de composição de *profiles* proposta, a fim de tornar mais fácil o entendimento através da comparação.

	Paradigma Orientado Objeto		Abordagem Proposta	
Tipo de relacionamento	Herança		Composição	
Elementos de entrada	Classe A	Classe B	Profile A	Profile B
Estabelecimento do relacionamento	Superclasse	Subclasse	Merged Profile	Receiving Profile
Resultado do relacionamento	Subclasse		Resulting Profile	

Figura 4.2: Paralelo entre a semântica do relacionamento de herança do paradigma orientado a objetos e a semântica de composição de *profiles* proposta

A principal dificuldade de entender a semântica é o fato do *receiving profile* ora atuar como operando do mecanismo de composição, ora representar o resultado da composição, passando a representar o *resulting profile*. Isto como na herança, depende do contexto no qual o *receiving profile* é considerado. Fazer referência ao conteúdo do *receiving profile* implica em fazer referência à composição do seu conteúdo com o conteúdo do *merged profile*. Ou seja, a composição do *receiving element* com o *merged element*.

Assim, este fato é ilustrado na Figura 4.3 na qual encontra-se quarto *profiles*: `Profile A`, `Profile B`, `Profile C` e `Profile D`. O `Profile B` se relaciona com o `Profile A` através do relacionamento de composição ≪merge≫, o `Profile C` importa o conteúdo do `Profile B`, assim como o `Profile D` importa o conteúdo do `Profile A`, ambos através do relacionamento ≪import≫. O `Profile A` e o `Profile B`, ambos, definem o *stereotype* `Professor`, logo como existe um relacionamento de composição entre eles, isto implica que qualquer referência ao *stereotype* `Profile B.Professor`, na verdade, representa uma referência à composição do *stereotype* `Profile A.Professor` e do *stereotype* `Profile B.Professor`. O `Profile C` importa o *stereotype* `Professor` do `Profile B` para definir o *stereotype* `Pesquisador`. Neste caso, o estereótipo Professor do Profile C representa a composição dos *stereotype* `Profile A.Professor` e `Profile B.Professor`. Ao passo que, o `Profile D` faz uso do *stereotype* `Profile A.Professor` que não representa a composição dos *profiles* `A.Professor` e `Profile B.Professor`. Sendo assim, o `Profile B` ora se comporta como *receiving profile* diante do relacionamento ≪merge≫, e ora como *resulting profile* diante do relacionamento ≪import≫.

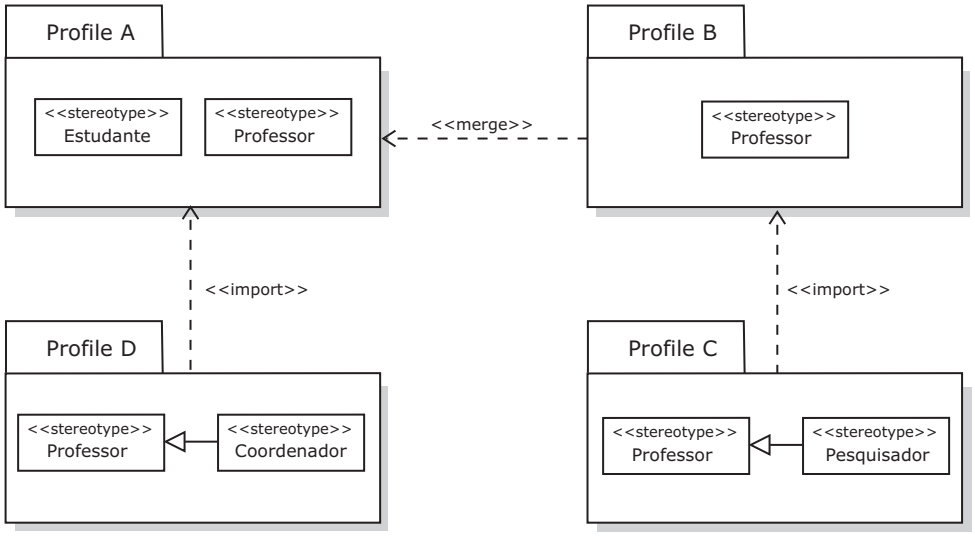

Figura 4.3: Exemplo ilustrando uma composição

4.2 Especificando os Modelos de Entrada

Especificar os modelos de entrada implica em definir o domínio do mecanismo de composição, o qual é definido a partir da especificação do tipo de modelos que o mesmo é capaz de compor, ou seja, os operandos da composição. Os elementos que não fazem parte deste domínio, poderão ser compostos possivelmente com a extensão desta abordagem. Desse modo, o domínio de composição da abordagem proposta concentra-se nos UML *profiles*.

Para realizar a composição de *profiles* é necessário em combinar todos os elementos que o compõe e aqueles elementos que o mesmo faz referência também devem ser considerados. Os tipos de modelos que um *profile* pode ter são definidos no metamodelo da UML (OMG 2007*c*)

Na perspectiva do mecanismo de composição, os elementos a serem compostos podem ser agrupados e distribuídos na forma de uma árvore. Esta representação é baseada na associação no relacionamento de tipo "composição" entre eles e na observação do metamodelo da UML, onde determinados modelos possuem propriedades e podem conter outros elementos através de alguma forma de associação. Com a finalidade de mostrar os elementos que são candidatos a serem compostos pelo mecanismo proposto, os mesmos são representados na Figura 4.4 através de uma estrutura de árvore.

Sendo assim, existem dois tipos de elementos, os elementos que são "nodo", que representam os elementos que são formados por outros elementos, e os que são "folhas", que não são compostos por nenhum outro elemento. Os nodos serão chamados *composite*, e as folhas de *primitive*, termos definidos em (Gamma et al. 1995, Clarke 2001). Um exemplo de *composite* é *Stereotype*, *Class*, *Association*, *Enumeration* e *Interface*. Como exemplo de *primitive* tem-se: *Property*, *Parameter*, *Constraints* e *Tagged Values*. Para definir quais elementos serão *composites* e quais elementos serão *primitives*, foi levado em consideração o metamodelo do UML *profiles* e a associação do tipo associação, como mencionado anteriormente de "composição", como segue:

- *composite:* são elementos não granulares, ou seja, aqueles elementos que contêm outros elementos, sendo isto especificado no metamodelo da UML através de associação de composição. Diante do relacionamento de composição, os *composites* não são vistos como unidades elementares, sendo sua composição representada pela composição de suas partes. Por exemplo, dado um *stereotype*, o qual é um elemento *composite*, a sua composição é realizada pela composição de suas *operações*, *tagged values* e todas suas propriedades sintáticas.

- *primitive:* são elementos vistos de forma granular diante do relacionamento de composição. No metamodelo da UML são caracterizados por fazer referências a outros elementos, porém não através de associação de composição. Isto pode ser ilustrado com as *Property* (ver Figura A.3, no Anexo A página. 143) que faz referência

a *Type* (esta referência à *Type* ela herda de *StructureFeature*) e não possui nenhum associação de composição com outros elementos. Observando que *Association* também possui uma referência à *Type*, porém possui uma associação de composição com *Property* o que a caracteriza como uma *Composite*.

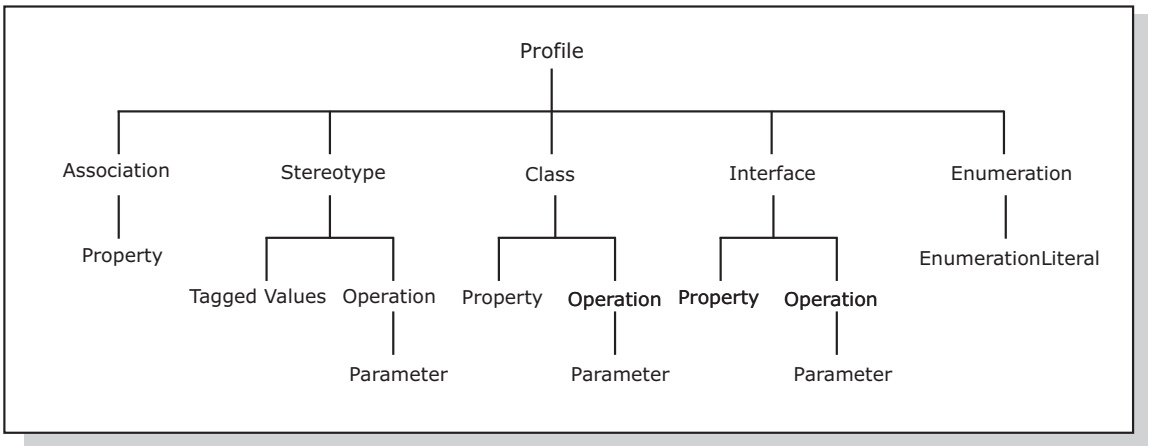

Figura 4.4: Elementos que podem ser compostos pelo mecanismo de composição

A partir da Figura 4.4 é possível observar que:

- *Profile:* é um *composite* que pode ter: *Association*, *Stereotype*, *Class*, *Interface* e *Enumeration*.
- *Association:* é um *composite* que possui *Property*.
- *Stereotype:* é um *composite* que possui *Tagged Values* e *Operation*.
- *Class:* é um *composite* que possui *Property* e *Operation*.
- *Enumeration:* é um *composite* que possui *EnumerationLiteral*.
- *Operation:* é um *composite* que possui *Parameter*.
- *Property:* é um *primitive*, o que implica em não ter outro tipo de elemento.
- *Tagged Values:* é um *primitive*, o que implica em não ter outro tipo de elemento.
- *Parameter:* é um *primitive*, o que implica em não ter outro tipo de elemento.
- *EnumerationLiteral:* é um *primitive*, o que implica em não ter outro tipo de elemento.

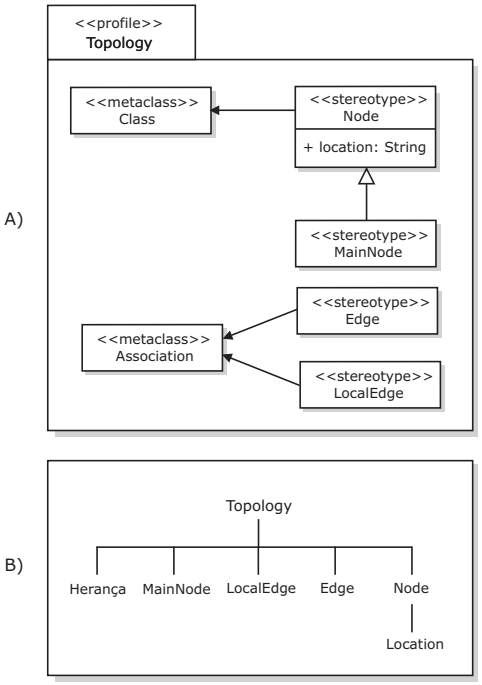

Figura 4.5: Exemplo ilustrativo de um *profile* e sua representação em árvore.

Em (Fernández & Moreno 2004) é definido um *Topology profile*, o qual é mostrado na Figura 4.5–A. Observa-se que o *profile* possui dois tipos de *Stereotype*: (i) os aplicáveis a *Association*, representado por *Edge* e *LocalEdge*; e (ii) os aplicáveis a *Class*, representado por *Node* e *MainNode*. Além disso, apresenta uma herança entre *Node* e *MainNode*. A fim de identificar os elementos que são *composite* e *primitive*, o *profile* é representado através de uma árvore ilustrada na Figura 4.5–B. Onde *Edge*, *LocalEdge*, *Node*, *MainNode* e Herança representam o *composite*, porém apenas *Node* possui elemento em seu interior. *Location* trata-se de um *primitive* e encontra-se definido em *Node*.

Na Figura 4.6, é apresentado um exemplo de composição de dois *profiles*, juntamente com a representação em árvore com o objetivo de facilitar o entendimento do conteúdo que será apresentado nas próximas seções. São apresentados dois *profiles*: `Tree` e `Topology` com elementos equivalentes. O *stereotype* `Topology.Node` (Figura 4.6-A) é equivalente à `Tree.Node` (Figura 4.6-B), esta equivalência é representada em Figura 4.6-C e Figura 4.6-D. Por fim, o resultado da composição é mostra na Figura 4.6-E e Figura 4.6-F.

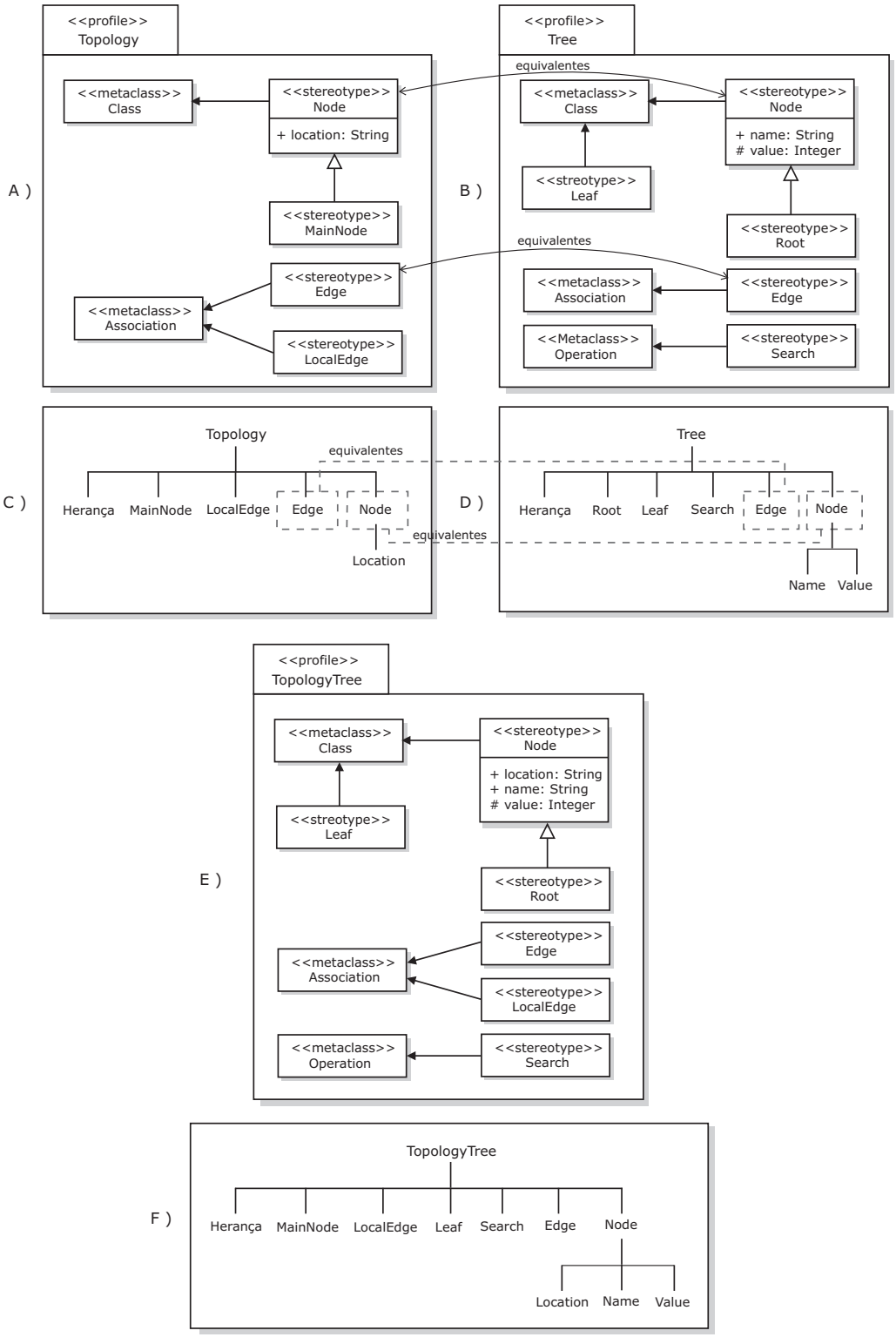

Figura 4.6: Composição de dois *profiles* e sua estrutura em árvore

4.3 Problemas Encontrados na Definição do Mecanismo de Composição da UML

O objetivo desta seção é apresentar os problemas encontrados na definição do mecanismo de composição da UML (OMG 2007*c*, página 154) e mostrar as soluções para estes problemas. Este mecanismo é formado por *match rules*, *constraints* e *transformations*, como descrito na Seção 2.3, porém apresenta ambigüidade, suas regras não são completas e apresenta inconsistência de acordo com a semântica de composição definida no mecanismo. A seguir é apresentada uma descrição.

Definição não é completa

- A fim de que as regras do mecanismo de composição sejam completas, as mesmas devem apresentar, para cada metaclasse definida na especificação da UML (OMG 2007*c*) uma *match rule*; para cada elemento da metaclasse (como por exemplo atributos, operações ou associações) deve existir uma *transformation* associada (Zito 2006). Sendo assim, foram elaboradas regras de comparação e regras de composição para os elementos presentes no metamodelo dos *profiles*. Além disso, são propostos quatro operadores de composição especificado na Seção 4.4 e um guia de composição de modelos especificado na Seção 4.5.

Apresenta ambigüidade

- O *package merge*, assim como a UML, é especificado através de linguagem natural, o que implica no surgimento de dúvidas, incerteza e diferentes interpretações da sua definição, configurando, desta forma, sua ambigüidade. Para solucionar esta ambigüidade cada *match rule*, *constraint* e *transformation* deve ser especificadas através de alguma linguagem ou notação formal. Neste contexto, a solução apresentada nesta dissertação foi modelada em um linguagem formal as regras propostas com o objetivo de verificar sua corretude e evitar ambigüidade.

Apresenta inconsistência

- A inconsistência do *package merge* se configura através das suas definições que são incompatíveis com a sua semântica atual definida, por exemplo, algumas *transformation* contradizem o que é especificado na semântica do *package merge*. Sendo assim, foram propostas regras de composição de acordo com a semântica apresentada nesta dissertação e a especificação formal permitiu retirar as inconsistência.

Para definir quais *match rules* e *transformations* não foram definidas e quais delas são ambíguas, foi tomado como base: (*i*) o padrão existentes nas regras já definidas; (*ii*) observação do metamodelo da UML; e (*iii*) o trabalho em (Zito 2006). Estas são descritas como segue:

Metaclasse *Package*

As regras ambíguas e as regras não definidas para a metaclasse *Package* presente no metamodelo dos *profiles*:
Regras Ambíguas (RA): não apresenta regras ambíguas.
Regras Não Definidas (RnD):

- RnD 1.1: não é definido nenhuma transformação para o valor *default*.
- RnD 1.2: não é definido nenhuma transformação para as operações.
- RnD 1.3: não é especificado nenhuma transformação para o *elementImport* ou *packageImport* herdados de *Namespace*.
- RnD 2.4: não é especificado nenhuma transformação para realizar a composição de comentários.

Metaclasse *Class*

As regras ambíguas e as regras não definidas para a metaclasse *Class* presente no metamodelo dos *profiles*:
Regras Ambíguas (RA): não apresenta regras ambíguas.
Regras Não Definidas (RnD):

- RnD 2.1: não é definido nenhuma transformação para o atributo *superClass*.
- RnD 2.2: não é definido nenhuma transformação para as operações.
- RnD 2.3: não é especificado nenhuma transformação para o *elementImport* ou *packageImport* herdados de *Namespace*.
- RnD 2.4: não é especificado nenhuma transformação para realizar a composição de comentários.

Metaclasse *Association*

As regras ambíguas e as regras não definidas para a metaclasse *Association* presente no metamodelo dos *profiles*:
Regras Ambíguas (RA): as *Match Rules* são ambíguas.
Regras Não Definidas (RnD):

- RnD 3.1: não é definido nenhuma transformação para as associações especializadas.
- RnD 3.2: não é especificado nenhuma transformação para o *elementImport* ou *packageImport* herdados de *Namespace*.

Metaclasse *Parameter*

As regras ambíguas e as regras não definidas para a metaclasse *Parameter* presente no metamodelo dos *profiles*:
Regras Ambíguas (RA): as *Match Rules* são ambíguas.
Regras Não Definidas (RnD):

- RnD 4.1: não é definido nenhuma transformação para o *default* valor.
- RnD 4.2: não é especificado nenhuma transformação para o atributo *direction*.
- RnD 4.3: não é definido nenhuma transformação para *type* herdado de *TypedElement*.

Metaclasse *Operation*

As regras ambíguas e as regras não definidas para a metaclasse *Operation* presente no metamodelo dos *profiles*:
Regras Ambíguas (RA): as *Match Rules* são ambíguas.
Regras Não Definidas (RnD):

- RnD 5.1: não é definido nenhuma transformação para as *exceptions*.
- RnD 5.2: não é especificado nenhuma transformação para o atributo *direction*.
- RnD 5.3: não é definido nenhuma transformação para *type* herdado de *TypedElement*.

Metaclasse *Property*

As regras ambíguas e as regras não definidas para a metaclasse *Property* presente no metamodelo dos *profiles*:
Regras Ambíguas (RA): as *Match Rules* são ambíguas.
Regras Não Definidas (RnD):

- RnD 6.1: não é definido nenhuma transformação para o valor *default*.
- RnD 6.2: não é especificado nenhuma transformação para o atributo *isDerivedUnion*.
- RnD 6.3: não é definido nenhuma transformação para *type* herdado de *TypedElement*.

Sendo assim, é apresentado uma solução a qual visa solucionar estes problemas em complemento as citadas anteriormente. Esta é organizada em três formas, como segue:

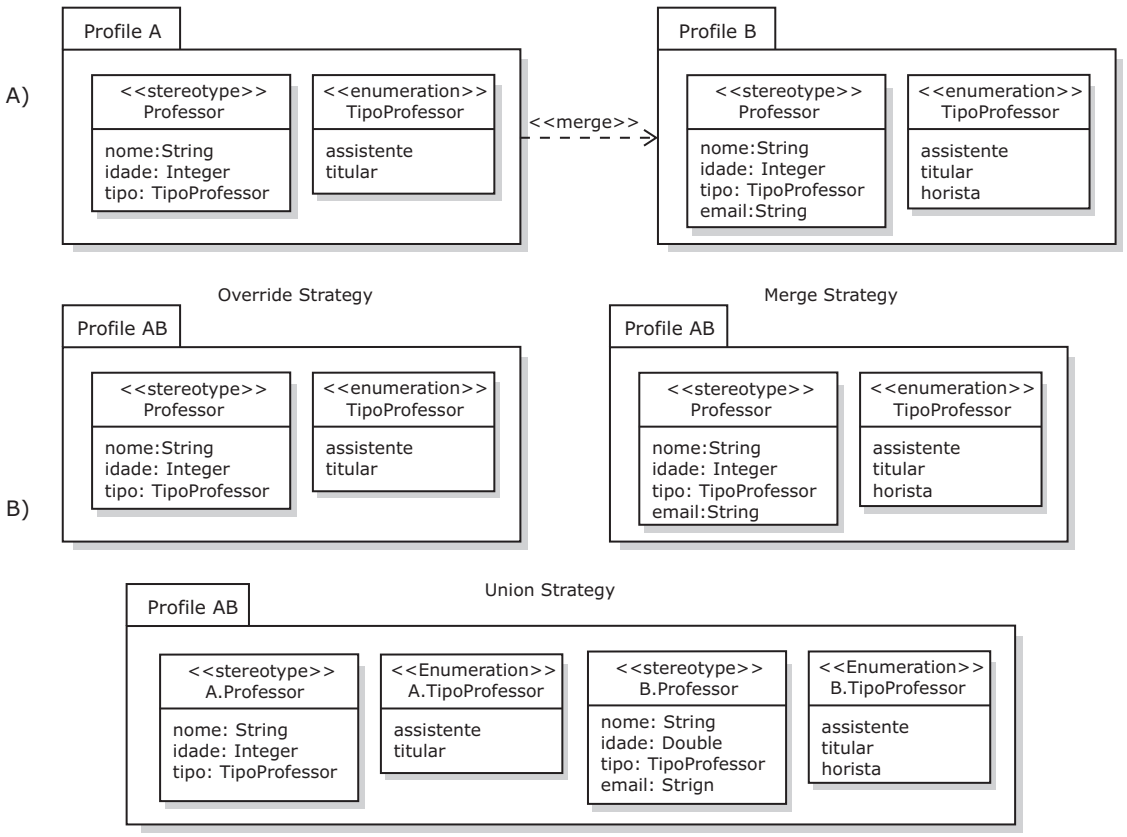

Figura 4.7: Exemplo de composição por estratégia

Composição Iterativa: todos os elementos que são contidos por um outro elemento serão compostos de forma iterativa, ou seja, todo *composite* deve ter seus elementos *primitive* compostos iterativamente. Como mostrado anteriormente, um elemento é contido por outro, quando existe um relacionamento de associção do tipo "composição" entre eles. Por exemplo, a metaclasse *Class* possui dois relacionamentos de composição sendo um com a metaclasse *Operation* e o outro com a metaclasse *Property* no Anexo A na página 143). Para realizar a composição de forma iterativa deve seguir os seguintes passos:

1. para cada *merged element* que não possui um *receiving element* equivalente é inserido no *resulting profile*, ou seja, fará parte dos *resulting elements*, sem alterações sendo a recíproca também verdadeira. Porém em caso de surgimento de conflitos, estes devem ser solucionados através da aplicação das regras de transformação definidas na Subseção 4.4.4.

2. quando um *merged element* possui um *receiving element* equivalente, eles serão compostos de acordo com as regras de composição definidas para as suas metaclasses de acordo com a estratégia de composição definida para a composição. Após a composição dos mesmos, estes serão inseridos no *resulting profile*

fazendo parte dos *resulting elements*.

Composição Dirigida por Estratégia: elementos que constituem os *merged elements* e os *receiving elements* fazem referência a outros elementos, porém não os contém. Por exemplo, a metaclasse *Operation* faz apenas referência à metaclasse *Type*, porém não a contendo. Estas referências que os *merged elements* e os *receiving elements* fazem deve ser compostas seguindo um estratégia de composição e aplicação de regras de transformação quando surgirem conflitos. A Figura 4.7 ilustra a composição de dois *profiles* os quais ambos definem o *stereotype* `Professor` e a *enumeration* `TipoProfessor`. É estabelecido o relacionamento de composição entre os *profiles*, sendo isto ilustrado na Figura 4.7-A na página 58. Sendo assim, é possível obter três resultados seguindo as três estratégias de composição. Esta estratégias são descritas com o objetivo de ilustrar. Estes resultados são ilustrados na Figura 4.7-B. Desse modo, observa-se que:

- **Override Strategy:** o *output model* produzido seguindo esta estratégia tem seus *stereotypes* `Professor` e `TipoProfessor` originados do `Profile A`.
- **Merge Strategy:** o *output model* produzido seguindo esta estratégia tem seus *stereotypes*: (*i*) `Professor` originado do `Profile A`, com o atributo *email: String* originado do `Profile B`; e (*ii*) `TipoProfessor` originados do `Profile A` com o *EnumerationLiteral* `horista` originário do `Profile B`.
- **Union Strategy:** o *output model* produzido seguindo esta estratégia tem seus *stereotypes*: (*i*) `A.Professor` originado do `Profile A`; (*ii*) `B.Professor` originado do `Profile B`; (*iii*) `A.TipoProfessor` originado do `Profile A`; e (*iv*) `B.TipoProfessor` originados do `Profile B`. Quando o *stereotype* `Professor` do `Profile A` e `Profile B` são inseridos no `Profile AB`, surge um conflito de nome, pois existem dois *stereotype* com nomes iguais em um mesma *namespace*. Logo, seus nomes são alterados através inclusão do nome dos seus *profiles* de origem.

Estas estratégias de composição são descritas com maiores detalhes no Capítulo 6.

Composição de Propriedades: para realizar a composição de propriedades de determinas metaclasses é necessário realizar um tratamento adequado das características das propriedades. Estas propriedades são definidas no metamodelo da UML como atributos das metaclasses, como por exemplo, o atributo *isAbstract* da metaclasse *Class* pode assumir os valores *true*, para definir que a instância de *Class* é uma classe abstrata, e *false*, para definir que a instância de *Class* é uma classe concreta. Dessa forma, como compor uma classe abstrata com uma concreta? Isto é possível? Respostas para estas perguntas são definidas nas regras de composição especificas

para a propriedade. Outro exemplo é o atributo *visibility* da metaclasse *Element*, a mesma pode assumir os valores *public*, *private*, *protect* e *package*, então surge novamente outra questão: como compor dois atributos sendo um *public* e outro *private*? Respostas para estas perguntas são apresentadas na definição da regra de composição desta propriedade. Sendo a forma de atuação da regras depende do tipo de estratégia de composição.

4.4 Definição dos Operadores de Composição

Nesta seção são definidos os operadores relacionados ao mecanismo de composição, são eles: *match operator*; *merge operator*; *composition strategy operator*; e *model transformation operator*. Estes operadores são os responsáveis pelas atividades pertencentes ao mecanismo de composição, fazendo uso de regras de composição (*merge rules*), regras de comparação (*match rules*) e regras de transformação de modelo (*model transformation rules*). Este operadores são descritos a seguir.

4.4.1 *Match operator*

O *Match Operator* trata-se de uma heurística e seu objetivo é definir a grau de equivalência entre os elementos dos modelos de entrada. Para definir o grau de equivalência entre os elementos dos *profiles* é utilizado dicionário de sinônimos, assinatura de modelos e a similaridade tipográfica entre nomes do modelos de entrada.

Dicionário de Sinônimo

Com o dicionário de sinônimo é possível o especialista de domínio fazer uso da sua *expertise* no domínio e aplicá-la no processo da definição do grau de equivalência entre o modelos de entrada. Assim, é possível definir a equivalência entre conceitos (representados através de assinaturas dos modelos) do domínio que é difícil de ser identificado pela comparação tipográfica e pela análise da estrutura sintática dos modelos. A Tabela 4.1 mostra um exemplo simples de dicionário de sinônimos, a qual possui duas colunas : (i) `Name`, para representar o conceito que possui sinônimos associados a ele; e (ii) `Synonym`, para especificar, de fato, os sinônimos.

O grau de similaridade entre um elemento do *receiving profile (r)* e um elemento *merged profile (m)* é representado como $\mathcal{D}(r,m) \rightarrow[0,1]$. Existem apenas duas possibilidades na comparação: uma é o *(r)* e o *(m)* serem sinônimos, sendo assim retorna 0 (zero) como resultado; e a outra é *(r)* e o *(m)* não serem sinônimos, sendo assim retorna 1 (um) como resultado. Para todos os possíveis pares de *(r,m)* é calculado $\mathcal{D}(r,m)$. Inicialmente, todo par de *(r,m)* são considerados como não sendo sinônimos, então $\mathcal{D}(r,m) = 0$ para todo e qualquer par de elementos dos *profiles* de entradas. Por exemplo, de

Tabela 4.1: Representação do dicionário de sinônimos

Name	Synonym
Leaf	EndNode, FinalNode
Edge	Border, Limit, Margin
Search	Research, Searching, Query

acordo com o dicionário de sinônimos mostrado na Table 4.1, os *stereotypes* `Tree.Leaf` e `Topology.EndNode`, mostrados na Figura 4.11, na página 77, representam o mesmo conceito, logo $\mathcal{D}$(*Leaf,EndNode*) = 1.

Similaridade Tipográfica

Durante a composição de modelos baseada na assinatura, nomes semelhantes, os quais representam o mesmo conceito em um domínio, pode ser considerados não equivalentes. Por exemplo, dado dois *stereotype* `Estudante` e `Estudantes`, os mesmos representam *string*'s diferentes, logo devem ser considerados elementos "totalmente" não equivalentes, o que é errado. Diante deste problema, é utilizado similaridade topográfica para definir um grau de similaridade baseado na assinatura.

A similaridade tipográfica é determinada como $\mathcal{T}(r,m) \rightarrow$ [0..1] para todos os possíveis pares de elementos dos *receiving (r)* e *merged (m) profiles*. Para implementar a similaridade tipográfica é usado o algoritmo N-gram (Manning & Shütze 1999), o qual define um valor entre [0..1] para todo par de *(r,m)*. Os pares são definidos a partir do produto cartesiano de (R×M), onde `R` e `M` representam o conjunto dos elementos do *profile* R e do *profile* M. Este algoritmo assegura um grau de similaridade para um par de *string* baseado no número de substring idênticas de tamanho N (nesta abordagem foi usado N = 2). Por exemplo, dado que o *profile* R possui 5 (cinco) elementos e *profile* M possui 7 elementos, logo serão formados (R×M) = (5×7) = 49 comparações, assim tendo 49 graus de similaridade.

Assinatura dos Modelos

A fim de levar em consideração a estrutura dos modelos de entradas para determinar sua equivalência são determinadas as "assinaturas dos modelos". Para cada tipo de modelo de entrada existe uma definição de uma assinatura para o mesmo. A assinatura é definida em termos das propriedades sintáticas dos modelos definidas no metamodelo da UML, onde as propriedades sintáticas definem a estrutura do modelo. A assinatura é uma coleção de valores para um dado subconjunto de propriedades sintáticas definidas no metamodelo da UML. Por exemplo, *isAbstract* é uma propriedade sintática definida dentro da metaclasse *Class* especificada no metamodelo da UML. Se a instância de *Class* é abstrata,

então *isAbstract* = `true`, caso contrário a instância é uma classe concreta, *isAbstract* = `false`.

O conjunto de propriedades sintáticas utilizadas para definir a assinatura de um *profile element* é chamada *signature type*, como definido em (Reddy et al. 2006, Straw et al. 2004). Uma *signature type* que consiste de todas as propriedades sintáticas associadas de modelo é então chamada de *complete signature type*, para as *signature type* que são apenas baseadas em algumas propriedades sintáticas dos modelos são nomeadas como *partial signature type*, já para as *signature type* que são apenas baseadas no nome dos modelos é nomeada como *default signature type*.

Por exemplo, uma possível *partial signature type* para uma operação seria um subconjunto consistindo das seguintes propriedades definidas na metaclasse *Operation*, tais como: `name`, o seu valor consiste do nome da operação; e `ownedParameter`, o seu valor é uma coleção de parâmetros associados com as operações. Sendo assim, quando é realizado a comparação de duas operações, considerando esta *partial signature type* é considerado apenas o nome da operação e parâmetros definidos para a mesma. Observando que para todo *profile element* deve ser definido uma *signature type*.

As assinaturas são estruturas em "níveis hierárquicos de comparação". Por exemplo, na Figura 4.11, uma possível definição dos níveis de comparação de uma *partial signature type* para os *stereotypes*, a qual leve em consideração apenas as propriedades sintáticas *name* e *property*, seria: `Tree.Node` (*name*) (nível 2), com `Tree.Node.name` (*property*)(nível 1).

Para determinar o grau de similaridade baseada na *signature type* ($\mathcal{M}$) entre um *receiving element (r)* e um *merged element (m)* $\mathcal{M}$(r,m) foi criado uma equação com objetivo de mensurar/dar um valor ao grau de similaridade. Esta equação consiste em uma média pondera de médias aritméticas, representada na Equação 4.1.

$$\mathcal{M} = \frac{\sum_{i=1}^{n} p_i \cdot \left[\sum_{j=1}^{k} \frac{\varphi_{i,j}}{k} \right]}{\sum_{i=1}^{n} p_i} \rightarrow [0..1] \tag{4.1}$$

- n é o number de níveis aplicado para os elementos dos profiles. Por exemplo, se a *partial signature type* definida para *stereotype* tem quatro níveis hierárquicos de comparação, logo o valor de $n = 4$.

- p_i representa os pesos, sendo $p_i = i$, onde $i \geq 1$ e $i \in N_+^*$;

- k expressa o número de elementos em cada nível, onde $k \geq 1$ e $k \in N_+^*$. Por exemplo, `Tree.Node` tem duas propriedades, como estas propriedades representam um nível, logo $k = 2$.

- $\varphi_{i,j}$ (onde i e j representam o nível e o item de um elemento de *profile* que são comparados, respectivamente) é usado para denotar se um item do *receiving element* (por exemplo `name:Strig` em `Tree.Node`) é equivalente a outro item do *merge element*. Ele representa uma variável booleana, sendo seu valor especificado através de regras de correspondência (*match rules*). As *match rules* comparam os ítens do *receiving element* e os ítens do *merge element*, retornando 1 (um) caso a regra seja satisfeita, caso contrário a mesma retorna 0 (zero). Por exemplo, quando é realizada a comparação dos *stereotypes* `Tree.Root` e `Topology.MainNode` tem-se $\varphi_{2,1} = 0$, aplicando a *match rule* MR1 (descrita na página 65), e $\varphi_{1,1} = 1$, aplicando a *match rule* MR3 (descrita na página 65).

O grau de similaridade entre um *receiving element* e um *merged element* é representado por $\mathcal{S}$. Para definir $\mathcal{S}$ é necessário combinar $\mathcal{D}$, $\mathcal{T}$, e $\mathcal{M}$ definidos anteriormente. O valor de $\mathcal{S}$ é obtido através da Equação 4.2. Se $\mathcal{D} = 1$, então $\mathcal{T}$ também assume valor 1 (um), e vice-versa.

$$\mathcal{S} = \frac{(\mathcal{D} + \mathcal{T} + \mathcal{M})}{\mathcal{D} + 2} \rightarrow [0..1] \tag{4.2}$$

Sendo assim, para realizar a composição dos *profiles* de entradas, é necessário saber o grau de similaridade dos modelos dos mesmos. Então, é calculado o grau de similaridade entre *receiving elements* e os *merged elements*, sendo agrupados e organizados em uma "tabela de similaridade". Além disso, é necessário definir um "limiar" (*Threshold* (t)) para ser utilizado como base para a escolha dos elementos equivalentes. Assim, todos os pares de *receiving element* e os *merged element* com grau de similaridade acima do limiar são considerados equivalentes. Ou seja, se $\mathcal{S}(r,m) > t$, então r e m são considerados equivalentes.

Comparando os *profiles* definidos e ilustrados na Figura 4.11 é gerado um tabela de similaridade, a qual é ilustrada na Figura 4.8, e possui alguns elementos destacados representando os elementos que estão acima do limiar para $t = 0.7$, ou seja, os elementos que são considerados equivalentes. Sendo assim, tem-se como pares de elementos equivalentes (Tree.Node,Topology.Node), (Tree.Edge,Topology.Edge), (Tree.Leaf,Topology.EndNode) e (Tree.StateKind, Topology.StateKind).

Match Rules

A fim de auxiliar o processo de definição de equivalência entre os *receiving elements* e *merged elements* foram definidas algumas *match rules*. O *match operator* é o responsável pelo uso destas regras e, de acordo com o resultado da execução, o mesmo define o valor de $\varphi_{i,j}$, o qual foi especificado anteriormente. Para todos os possíveis tipos de elementos que os *profiles* de entrada podem ter, é necessário ter definido uma *match rule* associada ao mesmo.

		Topology Profile					
		Node	MainNode	Edge	LocalEdge	EndNode	StateKind
Tree Profile	Node	0,83	0,22	0	0,08	0	0
	Root	0	0,05	0	0	0	0
	Edge	0	0	1	0,22	0	0
	Search	0	0	0	0	0	0
	Leaf	0	0	0	0	1	0
	StateKind	0	0	0	0	0	0,96

grau de similaridade acima do limiar (t = 0.7)

Figura 4.8: Tabela de similaridade entre os elementos dos *profiles*

As *match rules* são definidas considerando as definições apresentadas no MOF (OMG 2002) e no metamodelo da UML (OMG 2007*c*). Existem três tipos de *match rules*:

- *Default match rules:* são um conjunto de regras que se relacionam para verificar a equivalência dos modelos consideram apenas os nomes dos elementos que os formam. Por exemplo, *default match rule* para um *stereotype* levaria em consideração o nome do *stereotype*, o nome das suas propriedades, o nome das suas operações, e etc. Quanto mais refinada for necessária a comparação entre os modelos mais nomes das propriedades devem ser considerados.

- *Partial match rules:* são um conjunto de regras que se relacionam para verificar a equivalência dos modelos consideram um subconjunto das propriedades sintáticas dos elementos que os formam. Por exemplo, *partial match rule* para um *stereotype* levaria em consideração o nome do *stereotype*, a propriedade *isAbstract* do *stereotype*, a propriedade *isDerived*, e etc. Quanto mais refinada for necessária a comparação entre os modelos mais propriedades devem ser consideradas.

- *Complete match rules:* são um conjunto de regras que se relacionam para verificar a equivalência dos modelos consideram todas as propriedades sintáticas dos elementos que os formam. Por exemplo, *complete match rule* para um *stereotype* levaria em consideração todas as propriedades sintáticas definida no metamodelo da UML. Uma vez que uma *complete match rule* deve considerar todas propriedades sintáticas do elemento, logo quando esta regra retorna 1 (um) como resultado da comparação entre dois modelos, isto indica que os mesmos são iguais.

O mecanismo de composição proposto apresenta três tipos de estratégias de comparação entre os modelos de entrada: (i) *default match strategy*, faz uso das *default match rules*; (ii) *partial match strategy*, faz uso das *partial match rules*; e (iii) *complete match strategy*, faz uso das *complete match rules*. A diferença entre cada uma delas é tipo

de *match rules* as quais as mesmas fazem uso. Se é desejado uma comparação mais refinada, deve ser utilizado a *complete match strategy*, para uma comparação com um baixo grau de refinamento deve ser utilizado a *default match strategy*.

A execução de uma *match rule* retorna um valor booleano ao *match operator* informando a equivalência ou não. Se a regra falhar, então os modelos que estão sendo comparados não são equivalentes para a *match rule* ($\varphi_{i,j} = 0$). Caso contrário, os modelos são equivalentes ($\varphi_{i,j} = 1$). A seguir são apresentados exemplos de *default match rules*:

MR1. Stereotype match rule:
MatchStereotype(Stereotype rcv, Stereotype mrgd) →
rcv.name = mrgd.name AND
MatchAttribute(rcv, mrgd) AND
MatchOperation(rcv, mrgd)

MR2. Association match rule:
MatchAssociation(Association rcv, Association mrgd) →
(rcv.name = mrgd.name) AND (rcv.memberEnds =
mrgd.memberEnds)

MR3. Attribute match rule:
MatchAttribute(Stereotype rcv, Stereotype mrgd) →
(rcv.ownedAttribute.name = mrgd.ownedAttribute.name)
AND (rcv.ownedAttribute.TypedElement = mrgd.
ownedAttribute.TypedElement)

MR4. Operation match rule:
MatchOperation(Stereotype rcv, Stereotype mrgd) → (rcv.
ownedOperation.name = mrgd.ownedOperation.name)
AND (rcv.ownedOperation.ownedParameter.length =
mrgd.ownedOperation.ownedParameter.length) AND
(∀x(rcv.ownedOperation.ownedParameter[x] =
mrgd.ownedOperation.ownedParameter[x])

MR5. Enumeration match rule:
MatchEnumeration(Enumeration rcv,
Enumeration mrgd) → rcv.name = mrgd.name AND
MatchEnumerationLiteral(Enumeration rcv,
Enumeration mrgd)

MR6. Enumeration Literal match rule:
MatchEnumerationLiteral(Enumeration rcv,
Enumeration mrgd) → ∀x(rcv.ownedLiteral.name[x] =
mrgd.ownedOperation.name[x])

4.4.2 *Composition Strategy Operator*

O objetivo deste operador é especificar como a composição deve ser realizada. Em outras palavras, este operador determinar quais regras de composição deve ser utilizada pelo *Merge Operator*, definido na SubSecção 4.4.3. Para isto, este operador faz uso das estratégias de composições definidas. Foram definidas três estratégias de composição: (i) *override*; (ii) *union*; e (iii) *merge*. Estas estratégias são brevemente descritas a seguir:

Override Strategy. Esta estratégia especifica que as partes que são equivalentes (*overlapping parts*) do *receiving element* devem sobrescrever as *overlapping parts* do *merged element*. Sendo assim, as *overlapping parts* do *receiving element* são inseridas no *composed model*. Enquanto que as partes que não são compartilhadas (*non-overlapping parts*) do *receiving element* e do *merged element* são inseridas no *composed model*. Quando um *profile* precisa ter seu conteúdo modificado, ou ter apenas novos elementos inseridos, esta estratégia deve ser usada, configurando, desse modo, um cenário de aplicação desta estratégia.

Union Strategy. Esta estratégia especifica que todo *receiving element* e *merged element* devem ser inseridos no *composed model*. A composição realizada seguindo esta estratégia se caracteriza por ser uma composição conservativa. Quando é necessário ter todo o conteúdo do *receiving profile* e do *merged profile* em uma *profile* único, esta regra deve ser usada configurando, desse modo, um cenário de aplicação desta estratégia.

Merge Strategy. Esta estratégia especifica que as *overlapping parts* do *receiving element* e *merged element* são combinadas a fim de obter uma visão integrada das mesmas, enquanto que as *non-overlapping parts* são apenas inseridas no *composed model*. Isto implica que os elementos que são equivalentes aparecem uma única vez no *composed model*. Dado dois *profiles*, os quais possuem alguns conceitos equivalentes de um determinado domínio de aplicação, para se ter um *profile* único que represente os conceitos destes domínios, é necessário realizar uma composição destes *profiles* seguindo esta estratégia, representando, desse modo, um cenário de aplicação.

4.4.3 *Merge Operator*

O *merge operator* trata-se de uma heurística e seu objetivo é realizar a composição de modelos. Para realizar a composição é necessário saber o grau de equivalência entre os modelos de entradas e uma estratégia de composição a ser seguida.

Para realizar a composição, de fato, o *merge operator* faz uso de regras de composição (*merge rules*) as quais são especificadas no Anexo B na página 147. Dado dois

modelos de entrada, o *merge operator* faz uso de *merge rules* para obter um modelo que representa uma visão integrada dos mesmos.

Este operador assume que os modelos de entrada representam visões diferentes e consistentes de um mesmo conceito. O *merge operator* necessita de três entradas: (i) *match model*, os modelos que são equivalentes; (ii) *match description*, uma descrição da equivalência definindo quais elementos devem ser compostos; (iii) *composition strategy*, especifica qual estratégia de composição deve ser seguida para realizar a composição. Uma ilustração do *merge operator* é mostrado na Figura 4.9.

Para compor dois modelos de entrada, o *merge operator* deve identificar as partes que são compartilhadas entre eles, partes estas chamadas de *overlapping parts*. As partes que não são iguais são chamadas de *non-overlapping parts*. Para produzir uma visão integrada dos modelos de entrada é necessário compor as *overlapping parts*, sendo a forma desta composição definida pela estratégia de composição especificada, e as *non-overlapping parts*, as quais são apenas inseridas no *composed model*.

De acordo com (Bézivin et al. 2006), a composição de modelos pode ser vista como um operação, a qual dado dois modelos M_A e M_B, tem como saída um produto M_{AB}. Isto pode ser representado por $M_A + M_B \longrightarrow M_{AB}$. Em nosso contexto, a composição de UML *profiles* pode ser representada através da equação $P_A +_{\textcircled{S}} P_B = P_{AB}$, onde P_A é o *receiving profile*; $+_{\textcircled{S}}$ representa a composição e o tipo de estratégia de composição a ser seguida; P_B representa o *merged profile*; e P_{AB} representa o modelo de saída, o resultado da composição.

Esta equação é representada para o *merge operator* como: *Merge(MM,MD,CS)* = CM, sendo estes elementos descritos abaixo.

Match Models (MM): MM representa o conjunto de *receiving element* e *merged element* que são equivalentes. Por exemplo, a Figure 4.11(a) possui basicamente quatro equivalências entre `Tree` e `Topology`: (*i*) `Tree.Node` e `Topology.Node`; (*ii*) `Tree.Leaf` e `Topology.EndNode`; (*iii*) `Tree.Edge` e `Topology.Edge`; (*iv*) `Tree.StateKind` e `Topology.StateKind`.

Match Description (MD): MD denota o relacionamento de equivalência entre os elementos de MM. Por exemplo, o *Match Description* do *MM* anteriormente descrito seria: (`Tree.Node`→`Topology.Node`); (`Tree.Leaf`→`Topology.EndNode`); (`Tree.Edge`→`Topology.Edge`) (`Tree.StateKind`→`Topology.StateKind`);

Composition Strategy (CS): CS especifica como os modelos devem se integrados. Os *profiles* de entrada podem ser compostos seguindo de diferentes maneiras, dependendo do objetivo inicial da composição, representando, desse modo, uma forma flexível de realizar composição.

Composed Model (CM): CM representa o resultado da composição.

Merge rules são usadas pelo *merge operator* para especificar a composição entre *receiving element* e *merged element* que são equivalentes. Estas regras definem efetivamente como a composição deve ser realizada. Sendo assim, para cada elemento de entrada do mecanismo de composição deve existir um *merge rule* definida capaz de especificar como este elemento deve ser composto com o seu equivalente. As *merge rules* são formadas basicamente por um nome, dois elementos de entrada, e uma saída que representa o resultado da composição. As *merge rules* são definidas no Anexo B.

Sendo assim, com o objetivo de especificar a composição, foi criado *Merge Specification Rule* que é especificada seguindo o modelo de especificação (Oliveira & Oliveira 2007*a*), a seguir:

Name: define o nome da regra.
Description: usado para descrever como a regra deve ser aplicada.
Syntax: especifica a sintaxe da regra.
Pre-Condition: especifica as condições que devem ser satisfeitas para que uma regra seja executada. Estas condições podem ser expressadas através de linguagem formais (por exemplo Z, VDM, Alloy), OCL ou através linguagem natural.
Post-Condition: define as condições que deves ser satisfeitas após uma regra ter sido executada. Estas condições podem ser expressadas através de linguagem formais (por exemplo Z, VDM, Alloy), OCL ou através linguagem natural.

Uma vez definido o modelo é possível descreve a regra de especificação de composição seguindo este modelo, como segue:
Merge Specification Rule (MSR)
Name: regra de especificação de composição
Description: o objetivo desta regra é especificar a composição de dois modelos de acordo com uma estratégia de composição. Cada estratégia de composição produzirá, possivelmente, modelos compostos diferentes. Esta regra tem seis operandos: (i) dois modelos de entrada; (ii) duas *namespace* dos modelos de entrada; (iii) a especificação da estratégia de composição; (iv) estratégia de comparação.
Context: usada na composição de dois modelos
Syntax:
merge[Model.type] {
name: Model **in** owner: Namespace **with**
name: Model **in** owner: Namespace
by strategy: CompositionStrategy **and**
and match: MatchStrategy
}

Pre-condition: a *CompositionStrategy* deve descrever uma estratégia de composição válida. Para ser "válida" a mesma deve ser definida dentro do mecanismo de composição.
Post-condition: a regra de produzir um modelo de saída válido.

Uma vez que MSR tenha sido especificada, um conjunto de *merge rules* são utilizadas para realizar a composição. Por exemplo, para compor dois *stereotypes* `Tree.Node` e `Topology.Node` de acordo com a estratégia de composição *MergeStrategy* e a *DefaultMatchStrategy*, tem-se a seguinte sintaxe de MSR:

mergeStereotype {
Node **in** Tree
Node **in** Topology
by MergeStrategy **and** DefaultMatchStrategy
}

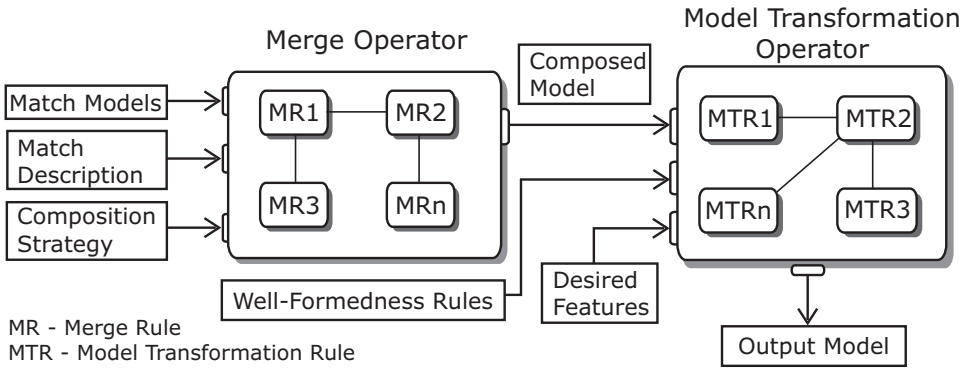

Figura 4.9: Operador de composição e de transformação

4.4.4 *Model Transformation Operator*

O *model transformation operator* trata-se de uma heurística e seu objetivo é identificar e solucionar problemas que surgem durante a composição, e assegurar que "desejadas características" estejam presentes no *output model*. Alguns dos problemas que este operador trata são: (*i*) referência a elementos que não existem; (ii) conflito de nome, ou seja, elementos com nomes iguais em uma mesma *namespace*; (iii) número elevado de elemento , por exemplo, grande quantidade de atributos; (iv) elemento em determinada *namespace*, porém fazendo referência a outra *namespace*.

Este operador tem três entradas: (i) *composed model*; (ii) *desired features*; e (iii) *well-formedness rules*. Como saída é produzido o *output model*, o resultado da composição. A Figura 4.9 mostra uma representação deste operador. Sendo assim, para produzir o *output model* o operador faz uso de *model transformation rules*. Estas *rules*

dão capacidade ao operador de manipular o modelo de entrada, *composed model*, para produzir o modelo de saída, *output model*, que representa efetivamente o resultado da composição.

Neste sentido, dado um *composed model* o operador verifica se existe algum problema e se os recursos desejados são encontrados. Para isto ele faz uso de *well-formedness rules* e *desired features*. Quando uma *well-formedness rule* ou um *desired feature* não é respeitada, isto representa uma *bad-formedness conflict*.

Para todo o problema identificado as regras de transformação devem ser aplicadas com o objetivo de solucioná-los.

Model Transformation Rules (MTR)

As *model transformation rules* criadas têm as seguintes capacidades:

- Criar e deletar modelos.
- Adicionar e remover modelos de um *namespace*.
- Renomear elementos de modelos.
- Alterar/Atualizar referências entre os modelos.

Para definir estas regras é utilizado o modelo apresentado anteriormente para definir a *merge specification rule*, como segue:

MTR1. Criar novo modelo
Name: create
Description: este regra de transformação é utilizada para criar um novo modelo. A função desta regra se assemelha com uma *factory*, tendo a capacidade de criar os modelos que podem ser inseridos nos *profiles*. Para especificar esta regra foi utilizado o princípio de reflexão definido no MOF (*CMOF::Reflection*) (OMG 2002). Para fazer uso desta regra é necessário: (*i*) fornecer o tipo do elemento que será criado; e (*ii*) os argumentos exigidos (quando necessário), os quais são definidos no metamodelo da UML. Um exemplo de propriedade que deve ser especificada no momento da criação do elemento consiste em *isAbstract* definida na metaclasse *Class*, a qual assume valor *true* ou *false*.
Context: criar novo modelo
Syntax: newElement = **create**[ProfileElement.type](arg: Argument[*]){
Factory.createElement([ProfileElement.type], arg) }
Pre-condition: checar se todos argumentos são válidos e se existem argumentos para as propriedade obrigatórias para criar o modelo.
Post-condition: a regra deve criar um modelo corretamente. Além disso, é possível fazer referências a *well-formedness rules* e *desired features* com o objetivo de verificar se

o elemento criado não possui um *bad-formedness conflict*.

MTR2. Deletar um modelo
Name: delete
Description: esta regra é aplicada quando determinado modelo deve ser deletado. Ela possui dois operandos: (i) o modelo que deve ser deletado; (ii) *namespace* de origem.
Context: deletar um modelo
Syntax:
delete[ProfileElement.type] name: ProfileElement
from owner: Namespace
Pre-condition: tanto a *Namespace* e o elemento a ser deletado devem ser elementos válidos e devem existir. Todos os operandos devem estar corretamente especificados.
Post-condition: a regra deve deletar um modelo corretamente. Além disso, é possível fazer referência a *well-formedness rules* e *desired features* a fim de verificar se *bad-formedness conflict* são criados após a deleção do modelo. Por exemplo, na Figura 4.11, se a *enumeration* `TreeTopology.StateKind` é deletada, então surge um "conflito de referência", pois o *stereotype* faz uma referência para a mesma, representado por `TreeTopology.Root.state`. Na Figura 4.11, deletar o *stereotype* `TreeTopology.Node` levaria ao surgimento de um "conflito de referência", pois os *stereotypes* `TreeTopology.Leaf`, `TreeTopology.Root` e `TreeTopology.MainNode` fazem referência a ele.

MTR3. Inserir um modelo em uma Namespace
Name: insert
Description: todo modelo deve ter uma *namespace*. Portanto, o objetivo da regra é inserir um modelo a uma *Namespace*. Para isto são necessários dois operandos: (*i*) o modelo (por exemplo um *stereotype*) que será inserido; e a (*ii*) *Namespace* especificando onde o modelo deve ser inserido.
Context: inserir um modelo em um *Namespace*.
Syntax:
insert[ProfileElement.type] name: ProfileElement
into owner: Namespace
Pre-condition: checar se a *Namespace* existe e verificar o nome do operador. Todos os operandos devem estar corretamente especificado.
Post-condition: o modelo de ser inserido corretamente na *Namespace*.

MTR4. Renomear um modelo em uma *Namespace*
Name: rename
Description: o objetivo da regra é renomear o modelo, a fim de evitar conflito de nome. Por exemplo, duas *tagged values* de diferentes *stereotypes* têm nomes iguais e devem ser

inseridos no *resulting profile*. Logo, configura um *name conflict*. Portanto, uma das *tagged values* deve ser renomeada. Esta regra tem três operandos: o modelo (por exemplo *stereotype*, *Operation*, etc.) que deve ser renomeado; (ii) especificação da *namespace*; e (iii) o novo nome que será atribuído ao modelo.
Context: renomear um modelo em uma *Namespace*
Syntax:
rename[ProfileElement.type] name: ProfileElement
from owner: Namespace
to newName: ProfileElement.name
Pre-condition: o modelo deve estar definido em uma *Namespace*.
Post-condition: a regra deve inserir o modelo corretamente.

MTR5. Criar uma associação entre dois modelos
Name: createAssociation
Context: criar uma associação entre dois modelos.
Description: esta regra é utilizada para criar associação de dois elementos de um *profile*. O primeiro operando cria uma associação com o segundo operando, sendo os mesmos de tipos iguais. Por exemplo, não é possível cria uma herança entre uma *stereotype* e uma *enumeration*. Para especificar a regra foi utilizado o princípio de reflexão definido no MOF (*CMOF::Reflection*) (OMG 2002). Sendo assim, é criado a associação utilizando uma *Factory* entre os dois modelos.
Syntax:
createAssociation(rt: RelationshipType , arg1: AssociationEnd, arg2: AssociationEnd){ Factory.createLink(rt, arg1, arg2) }
Pre-condition: os argumentos devem estar corretamente especificados.
Post-condition: a associação entre os modelos devem ser corretamente criada.

MTR6. Alterar referência entre modelos
Name: replaceReferences
Context: alterar referência entre os modelos .
Description: um modelo pode fazer referência a um outro modelo que não existe, logo originando um conflito de referência (*reference conflict*). Sendo assim, é aplicado esta regra. Esta regra faz uso da *replaceReferences directive* definida em (Straw et al. 2004) para resolver o conflito.
Syntax:
replaceReferences originalName: Name
with replacementName: Name **in** owner: Namespace
Pre-condition: os argumentos devem ser especificados.
Post-condition: a referência entre os modelos deve ser alterada corretamente.

4.5 Guia para Composição de Modelos

Na comunidade de Engenharia de Software, não existe um consenso sobre as atividades e as particularidades inerentes a composição de modelos (Bézivin et al. 2006), muito menos uma descrição ou esboço das características de um soluções em composição de UML *profiles*. Talvez isto seja motivado pelo fato de ser uma área de pesquisa nova ou devido à complexidade do tema. Como conseqüência, todos trabalhos inicializados que abordem o tema precisa de um grande investimento de tempo e esforço.

Sendo assim, com o objetivo de modificar esta realidade, foi analisado um conjunto de trabalhos visando fazer um levantamento dos requisitos básicos, das atividades e das necessidades pertinentes à composição de modelos. Feito isto, foi elaborado um guia de composição de modelos que tem como principal objetivo auxiliar o desenvolvimento da composição. Anteriormente, foram identificadas e delegadas algumas atividades relacionadas à composição de *profiles* aos operadores de composição. Como os operadores não trabalham isoladamente, então uma visão integrada das atividades desenvolvidas pelos mesmos é necessária. Desse modo, o guia de composição elaborado especifica o fluxo de atividades dos operadores a fim de obter uma completa interação e alinhamento entre as atividades desenvolvidas por eles.

Além disso, o guia visa representar boas práticas e tornar tão compreensível quanto possível o papel dos operadores na composição. Este guia é mostrado na Figura 4.10, o qual estende (Oliveira & Oliveira 2007*a*) por introduzir a especificação do fluxo de atividades dos operadores, refinar a forma de comparação entre os modelos, por fornecer uma nova forma de especificar a composição e transformação.

A fim de ter uma visão clara do mecanismo de composição de modelos, a abordagem de composição possui quatro fases:

Inicial Phase. Esta fase é iniciada quando o *match operator* recebe dois *profiles* de entrada. Então, este operador analisa os modelos de entrada com o objetivo de conhecer e ter informações sobre a natureza dos mesmos. O conteúdo dos *profiles* de entrada são separados e agrupados de acordo com seus tipos. Por exemplo, *Stereotype* e *Association* são identificados e agrupados de acordos com seus tipos.

Comparison Phase. Baseado na análise dos modelos de entrada, a assinatura de todos os tipos de elementos dos *profile* é definida. O objetivo desta fase é especificar quais elementos dos *profiles* são equivalentes. Para isto, *match operator* define o grau de similaridade ($\mathcal{S}$) para todos os *receiving profile* e *merged profile*, e baseado no *threshold* (t), finalmente determina quais elemento dos profiles são equivalentes. Esta fase é finalizada assim que a descrição de equivalência, os modelos equivalentes e os não equivalentes são definidos.

Merge Phase. O objetivo desta fase é produzir o *composed model* (ver Figura 4.10). Para isto, o *composition strategy operator* define a estratégia de composição que será usada pelo *merge operator*. Uma vez que a descrição de equivalência, os modelos equivalentes e não equivalentes são definidos, o *merge operator* realiza a composição dos modelos equivalentes. Esta fase é finalizada assim que o *composed model* é criado.

Post-Composition Phase. O objetivo desta fase compreende nos objetivos do *model transformation operator* definido anteriormente. Baseado em *desired features* e *well-formedness features*, este operador verifica se o modelo possui alguma má formação. Sendo esta má formação representada como conflitos. Quando conflitos são encontrados, o operador utiliza *model transformation rules* a fim de solucionar estes conflitos. O próximo passo é executar a transformação. O modelo de saída desta fase é produzido assim que nenhum conflito é identificado. The *output model* representa o resultado da composição.

4.6 Exemplo de Composição de UML *Profiles*

Baseado nas seções anteriores, agora é apresentado um exemplo de composição de *profiles*, o qual é ilustrado na Figura 4.11. A composição será realizada entre dois *profiles*: `Tree` e `Topology`. Para realizar a composição, duas características devem se encontradas no *resulting profile*: (i) um `Leaf` deve ter `Node` como superclasse da forma que `Node` o faz.

Desse modo, a fim de especificar e compor `Tree` e `Topology` (ver 4.11(a)) são utilizados a regra de especificação de composição e o guia definido anteriormente é utilizado, como segue:

1. Definição da composição usando a regra de especificação de composição, tem-se:
   mergeProfile {
   Tree **in** DefaultNamespace
   Topology **in** DefaultNamespace
   by MergeStrategy **and** DefaultMatchStrategy
   }

2. **Initial Phase.** Os modelos de entradas são analisados e conteúdo deles são tratados de acordo com os seus tipos. For exemplo, uma vez que `Topology.Edge`, `Topology.LocalEdge` e `Tree.Edge` são de tipos iguais e são aplicados a `Association`, então são "agrupados".

3. **Comparison Phase.** É definido a *default match strategy* como a *match strategy* que deve ser seguida na composição; o *threshold* usado para definir a equivalência

entre o modelos será $t = 0.7$. Baseado nisto, é calculado o grau de similaridade ($\mathcal{S}$) para todo o possível par (r,m), onde r são *receiving elements* e m são os *merged elements*. O resultado da comparação é mostrado na tabela de similaridade ilustrada na Figura 4.8. Para $t = 0.7$ os modelos equivalentes são: `Tree.Node`, `Tree.Leaf`, `Tree.Statekind`, `Topology.Node`, `Topology.EndNode`, `Topology.StateKind`. Sendo assim, a descrição de equivalência definida é: (`Tree.Node`→`Topology.Node`), (`Tree.Leaf`→`Topology.EndNode`) and (`Tree.StateKind`→`Topology.StateKind`).

4. **Merge Phase.** Neste exemplo, a estratégia de composição utilizada é a *merge strategy*. Portanto, os elementos dos *profiles* que são equivalentes deverão aparecer apenas uma vez no *composed model* (mostra uma visão integrada dos modelos de entrada). A partir dos modelos equivalentes e da descrição de equivalência entre os mesmos, os modelos composto produzidos são: (`TreeTopology.Node`), (`TreeTopology.Leaf`) and (`TreeTopology.StateKind`) (ver Figura 4.11(c)).

5. **Post-Composition Phase.** Existem duas características que não são satisfeitas, e `TreeTopology.MainNode` faz referência a `Topo- logy.StateKind` que não existe, representando um conflito de identificado como, *conflict reference*. Com o objetivo de solucionar estes problemas o *model transformation operator* deve realizar algumas atividades, como segue:

   - **Passo 01:** toda referência para `Topology.StateKind` deve ser alterada para `TreeTopology.StateKind`, sendo assim tem-se:
     replaceReferences `Topology.StateKind`
     with `TreeTopology.StateKind` **in** `TreeTopology`

   - **Passo 02**: o *stereotype* `TreeTopology.Leaf` deve estender `TreeTopology.Root`, assim, tem-se:
     createAssociation(inheritance, TreeTopology.Leaf,
     TreeTopology.Node){
     Factory.createLink(inheritance, TreeTopology.Leaf, TreeTopology.Node) }

   - **Passo 3**: a propriedade `leaf:Leaf` deve ser criada e inserida em `TreeTopology` profile:
     leaf = **createProperty**(Property, leaf, Leaf){
     Factory.createElement(Property, leaf, Leaf) }
     insertProperty leaf
     into TreeTopology
     Por fim, o *resulting profile* produzido é mostrado na Figure 4.11(d).

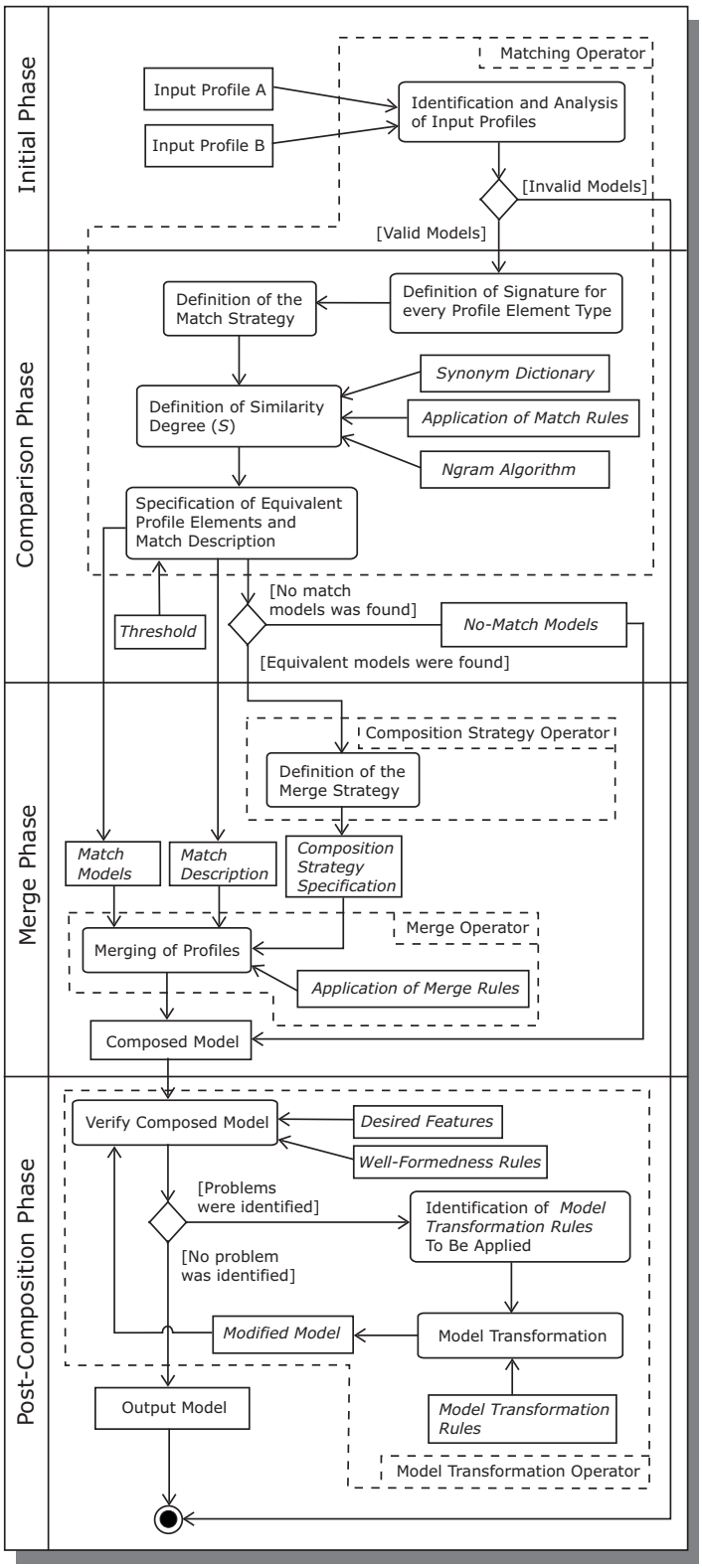

Figura 4.10: Guia de composição de modelos

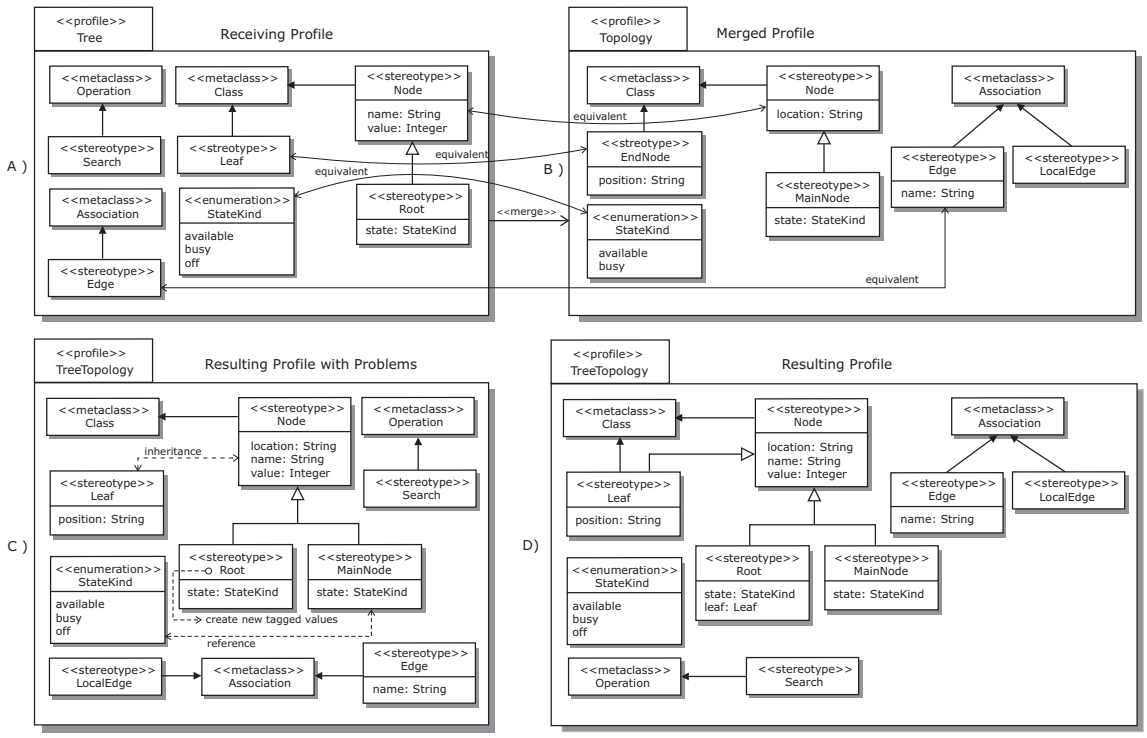

Figura 4.11: Exemplo ilustrativo de composição de *profiles*

Capítulo 5

Estratégias de Composição

Neste capítulo são descritas as estratégias de composição especificadas nesta dissertação. O objetivo destas estratégias é dar flexibilidade à composição através da apresentação de diferentes maneiras de realizar a composição de modelos. Ao longo do capítulo são apresentadas três seções:

- *Override Strategy:* nesta seção é apresentada a descrição da estratégia *override*, do seu cenário de aplicação e de sua semântica.
- *Union Strategy:* nesta seção é apresentada a descrição da estratégia *union*, do seu cenário de aplicação e de sua semântica.
- *Merge Strategy:* nesta seção é apresentada a descrição da estratégia *merge*, do seu cenário de aplicação e de sua semântica.

5.1 *Override Strategy*

O relacionamento de composição baseado na *override strategy* especifica que os elementos do *receiving profile* devem sobrescrever os elementos equivalentes do *merged profile* originando um *resulting profile*, o qual possui todos elementos de *receiving profile* e todos os elementos de *merged profile*, exceto aqueles equivalentes aos do *receiving profile*. Todas as partes que não são equivalentes do *merged profile* em relação ao *receiving profile* são apenas copiadas para o *composed profile*.

Nesta seção será detalhada a semântica de composição baseado na composição baseado na *Override Strategy*, para isto tem-se as seguintes Subseções contendo:

- possíveis cenários de aplicações do relacionamento de composição baseado na *override strategy*;
- a descrição da semântica de composição de *override strategy*.

5.1.1 Cenários de Aplicação

Override Strategy é aplicada quando um *profile* precisa ter alguns dos elementos modificados, ou quando é necessário adicionar novos elementos. Por exemplo, a construção de um *profile* é realizada tendo como base um metamodelo, o qual define os conceitos específicos do domínio que devem ser especificados no *profile*. Dentro do cenário de evolução de software, os domínios das aplicações estão constantemente evoluindo, sendo esta evolução caracterizada, ou pelo incremento de novos requisitos, ou pela alteração dos já existentes. Com isso, o metamodelo é alterado a fim de refletir as alterações do domínio. Da mesma forma, o *profile* específico do domínio deve ser alterado para refletir as alterações ou a incremento de novos conceitos no metamodelo. Esta alteração no *profile* pode ser realizada através da composição baseada na *Override Strategy*.

Outro cenário de aplicação para *Override Strategy* é no contexto de GSD (*Global Software Development*) . Onde equipes distribuídas trabalham na especificação de uma *profile*. Estando este *profile* em um repositório, é possível que alguma parte da especificação do *profile* desenvolvida por uma equipe venha a ser sobrescrita ou tenha novos elementos desenvolvidos por outra equipe. Isto pode ser implementado usando composição de *profile* baseada na *Override Strategy*. Por exemplo, qualquer alteração do *Business Metamodel* (ver Figura 1.1) implica em realizar alteração do *Java Profile* e realizar a composição dos *Web, Java e Oracle Profiles*, novamente.

Definir a estratégia de composição a qual relacionamento de composição deve seguir, implica em especificar a forma como os modelos devem ser compostos. Compor seguindo a *override strategy* consiste em definir que as *overlapping parts* do *receiving profile* deve sobrescrever as *overlapping parts* do *merged profile*.

Quando o relacionamento de composição tem *override strategy* como estratégia de composição, esta composição passa a ter a mesma semântica do relacionamento de herança. Um exemplo de composição seguindo esta estratégia de composição é mostrado na Figura 5.1. A superclasse (*merged profile*) terá o seu conteúdo (*merged elements*) que é equivalente ao da subclasse (*receiving elements*) sobrescritos. Ou seja, todo o *receiving element* que tiver algum *merged element* equivalente, sobrescreverá o *merged element*, dessa forma, sendo inserido no *resulting profile*.

5.1.2 Semântica de *Override Strategy*

Nesta Subseção é discutida a semântica que o relacionamento de composição assume quando é definido *override strategy* como a estretégia de composição.

Conforme anteriormente mencionado, *override strategy* é aplicada quando o *profile* precisa ter alguns dos elementos modificados, ou quando é necessário adicionar novos elementos, ou seja, especializá-lo. O relacionamento de composição baseado na *override strategy* especifica que os elementos de *receiving profile* deve sobrescrever os elementos

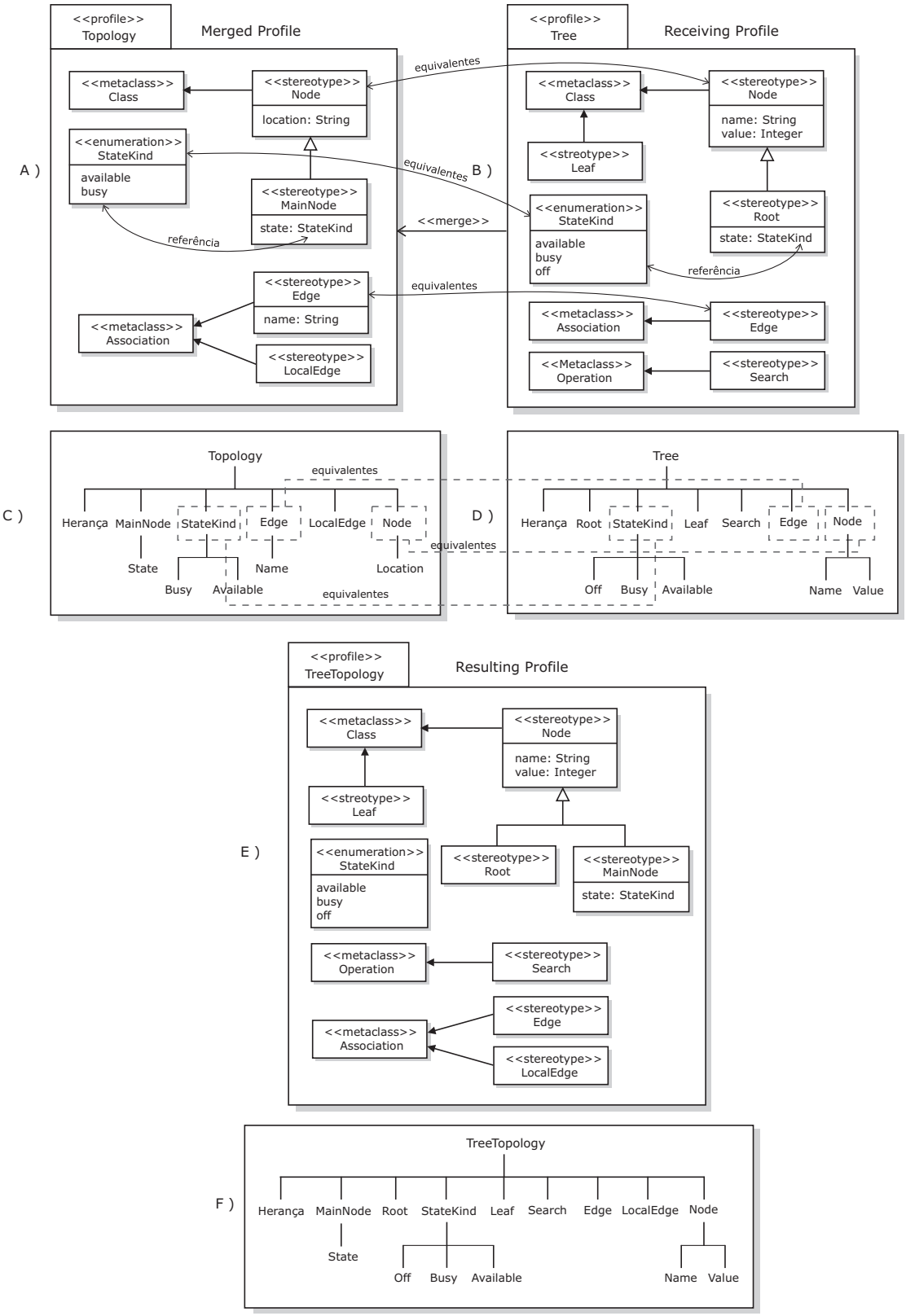

Figura 5.1: Exemplo de composição seguindo a *override strategy*

equivalentes do *merged profile* originando um *resulting profile* que possui todos elementos de *receiving profile* e todos os elementos de *merged profile*, exceto aqueles equivalentes aos do *receiving profile*, como mencionado anteriormente.

A maneira de identificar os elementos equivalentes, as *overlapping parts*, é especificada no Capítulo 4. A composição seguindo a semântica de composição *override strategy* deve respeitar as seguintes regras, como segue:

Regra 01: para todo *receiving element* que possua um *merged element* equivalente implica que *receiving element* deve sobrescrevê-lo. A aplicação desta regra é ilustrada na Figura 5.1, onde é considerado a *default match strategy* como a base para a realização da definição da equivalência. Desse modo, observa-se três casos, como segue:

1. o *stereotype* `Tree.Node` sobrescreve o *stereotype* `Topology.Node` tendo `TopologyTree.Node` como resultado.
2. o *enumeration* `Tree.StateKind` sobrescreve o *enumeration* `Topology.StateKind` tendo `TopologyTree.StateKind` como resultado.
3. o *stereotype* `Tree.Edge` sobrescreve o *stereotype* `Topology.Edge` tendo `TopologyTree.Edge` como resultado.

Regra 02: para todo *merged element* que não possua um *receiving element* equivalente, o mesmo deverá ser inserido no *resulting profile* sem alterações. Na Figura 5.1, é apresentado o uso desta regra, onde os *stereotype* `Topology.LocalEdge` e `Topology.MainNode` são inseridos no *resulting profile* sem alterações.

Regra 03: para todo *receiving element* que não possua um *merged element* equivalente, o mesmo deve ser inserido no *resulting profile* sem alterações. Na Figura 5.1, é apresentado o uso desta regra, onde os *stereotype* `Tree.Leaf`, `Tree.Root` e `Tree.Search` são inseridos no *resulting profile* sem alterações.

Regra 04: o *resulting profile* não deve apresentar problemas, como por exemplo conflitos de nomes ou referência elementos que não existem. Caso exista, é necessário realizar algumas transformações. A aplicação desta regra é ilustrada na Figura 5.1, onde *stereotype* `Tree.MainNode` possui uma referência à *enumeration* `StateKind`. As *enumeration* `Topology.StateKind` e `Tree.StateKind` são equivalentes, o que implica em `Tree.StateKind` sobrescrever `Topology.StateKind`. Desse modo, surge um *conference conflict*, haja vista `TreeTopology.MainNode` faz referência à *enumeration* `Topology.StateKind` que não existe. Este conflito é solucionado com a mudança de referência da *enumeration* `Topology.StateKind` para `TreeTopology.StateKind`.

Regra 05: o *resulting profile* deve respeitar as restrições e as especificações do metamodelo da UML (OMG 2007*c*).

5.2 *Union Strategy*

O relacionamento de composição baseado na *union strategy* especifica que todos os *receiving elements* e todos os *merged elements* devem ser adicionados ao *resulting profile*, o qual passa a ter todo o conteúdo do *receiving profile* e do *merged profile*.

As partes que são equivalentes do *merged profile* em relação ao *receiving profile* são copiadas para o *resulting profile*, porém algumas transformações são necessárias de serem realizadas. Por outro lado, as partes que não são equivalentes entre o *merged profile* e o *receiving profile* são adicionadas ao *resulting profile* sem alterações.

Todas as partes que não são equivalentes do *merged profile* em relação ao *receiving profile* são apenas copiada para o *resulting profile*. Porém, ao adicionar as partes equivalentes entre o *merged profile* e o *receiving profile* ao *resulting profile*, surgem conflitos de nome, haja vista terão *resulting elements* com assinaturas iguais em um mesma *namespace*, sendo necessário, com isso, realizar alguns transformações.

Estabelecer o relacionamento de composição entre dois *profiles* de entrada seguindo a semântica de composição definida para a *union strategy* implica em uma composição conservativa. Haja vista, os *profiles* são compostos seguindo um princípio de conservação dos conceitos dos domínios que os mesmos representam. Sendo assim, todos os conceitos presentes nos *profiles* de entrada terão que estar presentes no *resulting profile*. Para conservar os conceitos é necessário realizar algumas transformações nos elementos dos *profiles* que serão inseridos no modelo de saída para não originar conflitos. Na Figura 5.2 é mostrado um exemplo de composição seguindo a *union strategy*, onde se tem na parte (E) da figura `Tree.Node`, `Tree.StateKind`, `Topology.StateKind` e `Topology.Node` que mostram a inserção do nome do *profiles* de origem `Topology` e `Tree`.

A semântica de composição baseada na *union strategy* é descrita nesta seção. Para isto, tem-se as seguintes Subseções contendo:

- possíveis cenários de aplicações do relacionamento de composição baseado na *union strategy*;
- a descrição da semântica de composição de *union strategy*.

5.2.1 Cenário de Aplicação

Quando é estabelecido um relacionamento de composição entre dois *profiles* aplicando *union strategy* como a estragégia de composição, isto implica em uma tentativa de conservar o conteúdo dos *profiles* de entrada. Por exemplo, tendo-se os *profiles* dos *framework* JSF

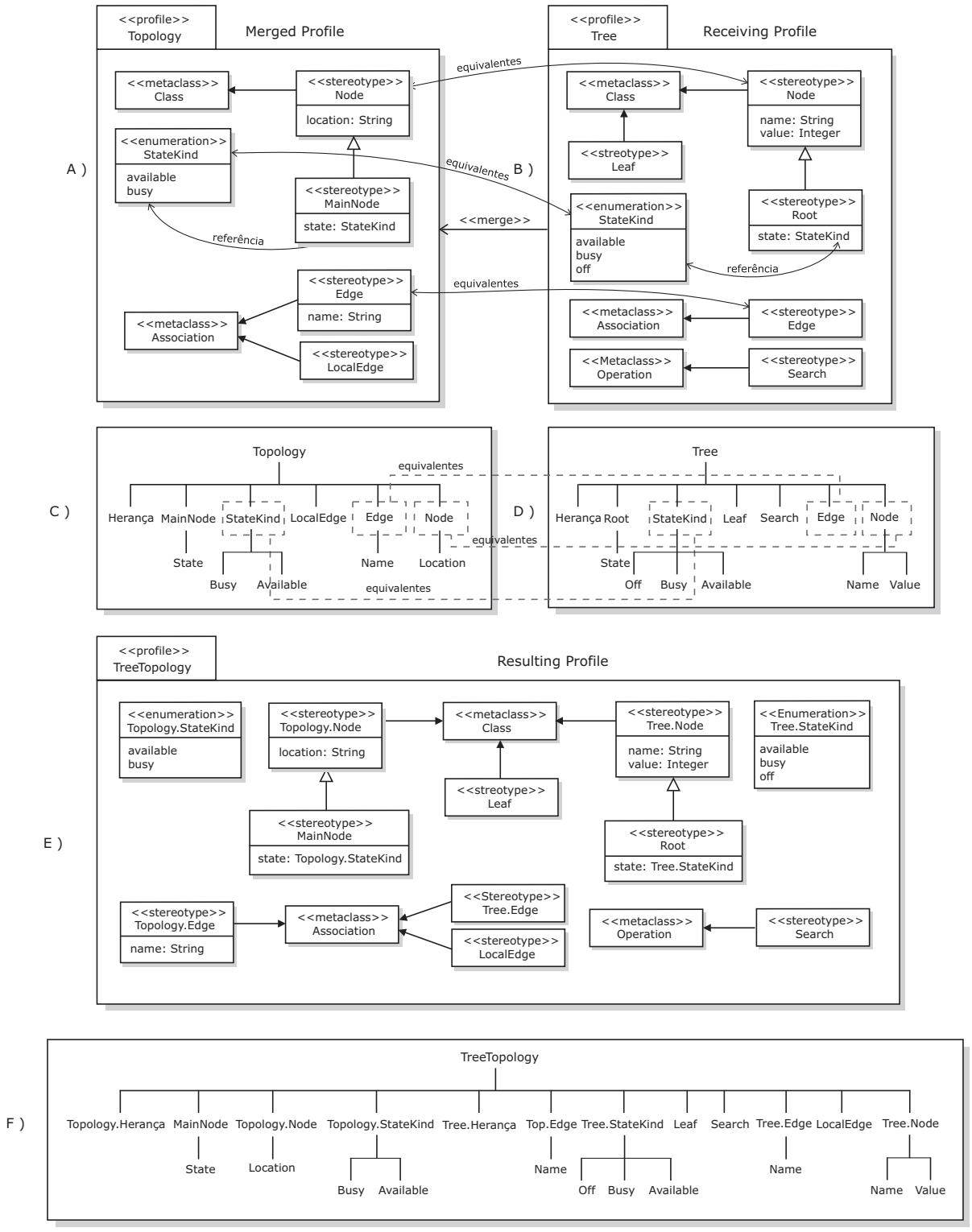

Figura 5.2: Exemplo de composição seguindo a *union strategy*

(*Java Serve Face*) e Struts utilizados para modelar aplicações Web, busca-se ter um *profile* único que possa representar estes dois *profiles*. Para isto, é necessário apresentar a capacidade de representar todos os conceitos de ambos os *frameworks*. Sendo assim, é possível fazer uso da *union strategy* para originar um *resulting profile* que possua todos os conceitos de ambos os *frameworks*.

Definir o relacionamento de composição seguindo a *union strategy* consiste em definir que as partes que são equivalentes (*overlapping parts*) do *receiving profile* e do *merged profile* sofram transformações a fim de que todas as partes que são equivalentes estejam presentes no *resulting profile*, enquanto que as partes não equivalentes são apenas adicionadas no *resulting profile*.

5.2.2 Semântica de *Union Strategy*

Nesta Subseção é discutida a semântica que o relacionamento de composição assume quando é definido *union strategy* como a estretégia de composição. *Union strategy* é aplicada quando existe a necessidade de obter um *profile* único contendo todo o conteúdo dos *profiles* de entrada, como já mencionado.

O relacionamento de composição baseado na *union strategy* especifica que os elementos de *receiving profile* deve "reunir" os elementos equivalentes de *merged profile* originando um *resulting profile*, o qual possui todos os *receiving elements* e *merged elements* equivalentes combinados. A forma de identificar os elementos que são equivalentes, as *overlapping parts*, é a mesma especificada no Capítulo 4.

A composição seguindo a semântica de composição *union strategy* deve respeitar as seguintes regras, como segue:

Regra 01: para todo *receiving element* que possua um *merged element* equivalente, isto implica que os mesmos deverão ser inseridos no *resulting profile* sendo necessário aplicar algumas transformações, tais como: atualizar *namespace* e renomear os elementos equivalentes. A aplicação desta regra é ilustrada na Figura 5.2, onde é considerado a *default match strategy* como a base para definir as *overlapping parts*. Desse modo, observa-se três *overlapping parts*, o surgimento de alguns conflitos e a realizações de algumas transformações, como segue:

1. os *stereotypes* `Tree.Node` e `Topology.Node` são equivalentes e, uma vez que sejam inseridos em *TreeTopology* surge um conflito de nome, haja vista têm-se dois *stereotypes* com nomes iguais em uma mesma *namespace*. Logo, é necessário realizar uma transformação que consiste, basicamente, em renomeá-los e verificar o que esta modificação produz. Por exemplo, um vez que um elemento teve seu nome alterado é possível que surjam conflitos de referência, ou seja, elementos fazendo referência a um elemento que não existe, devido à

alteração do nome. Outro exemplo seria, o conflito de referência do relacionamento de herança (ver Figura 5.2 na página 83) entre `TreeTopology.MainNode` e `TreeTopology.Topology.Node`, a qual faz referência ao `Topology.Node` e não ao `TreeTopology.Topology.Node` que é o correto.

2. o *enumeration* `Topology.StateKind` e `Tree.StateKind` são equivalentes e devem ser inseridas no *resulting profile* `TreeTopology`. Porém, surgem conflitos de nome, haja vista têm-se dois *stereotypes* com nomes iguais em uma mesma *namespace*. Com isso, é realizado uma transformação que consiste em renomeá-los, obtendo `TreeTopology.Topology.StateKind` e `TreeTopology.Tree.StateKind`. Isto tem um impacto nos elementos que fazem referência a eles, surgindo conflitos de referência, por exemplo `TreeTopology.MainNode` e `TreeTopology.Root` fazendo referência à `Topology.StateKind` e `Tree.StateKind`, respectivamente, sendo necessário atualizar estas referências.
3. os *stereotypes* `Tree.Edge` e `Topology.Edge` são equivalentes, sendo necessário renomear estes elementos, a fim de inseri-los no *resulting profile* `TreeTopology`. Como resultado tem-se `TreeTopology.Tree.Edge` e `TreeTopology.Topology.Edge`.

Regra 02: para todo *merged element* que não possua um *receiving element* equivalente, o mesmo deverá ser inserido no *resulting profile* sem alterações. Na Figura 5.2, é ilustrado o uso desta regra, onde os *stereotype* `Topology.LocalEdge` e `Topology.MainNode` são inseridos no *resulting profile* sem alterações.

Regra 03: para todo *receiving element* que não possua um *merged element* equivalente, o mesmo deve ser inserido no *resulting element* sem alterações. Na Figura 5.2, é apresentado o uso desta regra, onde os *stereotype* `Tree.Leaf`, `Tree.Root` e `Tree.Search` são inseridos no *resulting profile* sem alterações.

Regra 04: o *resulting profile* não deve apresentar conflitos. Caso exista, é necessário aplicar as *model transformation rules*, anteriormente definidas. Um exemplo de aplicação destas regras é ilustrada na Figura 5.2, onde *stereotype* `Tree.MainNode` possui uma referência à *enumeration* `StateKind`. As *enumeration* `Topology.StateKind` e `Tree.StateKind` são equivalentes, o que implica que `Tree.StateKind` é composto com `Topology.StateKind`. Desse modo, surge um *reference conflict*, haja vista `TreeTopology.MainNode` faz referência à *enumeration* `Topology.StateKind` que não existe mais. Este conflito é solucionado com a mudança de referência da *enumeration* `Topology.StateKind` para `TreeTopology.StateKind` através da aplicação da regra de transformação MTR6 definida na Seção 4.4.4 na página 70.

Regra 05: o *resulting profile* deve respeitar as restrições e as especificações do metamodelo da UML (OMG 2007*c*).

5.3 *Merge Strategy*

O relacionamento de composição baseado na *merge strategy* especifica que os elementos do *receiving profile* devem ser combinados com os elementos equivalentes do *merged profile* originando um *resulting profile*, o qual possui o resultado da combinação de todos os elementos do *receiving profile* com os do *merged profile*.

Todas as partes que não são equivalentes de *merged profile* em relação ao *receiving profile* são apenas copiadas para o *resulting profile*. Por outro lado, as partes equivalentes entre o *merged profile* e o *receiving profile* são combinadas, como anteriormente mencionado.

Nesta Seção será detalhada a semântica de composição baseado na composição baseado na *merge strategy*, para isto tem-se as seguintes Subseções contendo:

- possíveis cenários de aplicações do relacionamento de composição baseada na *merge strategy*;
- a descrição da semântica de composição de *merge strategy*.

5.3.1 Cenário de Aplicação

Merge strategy é aplicada ao relacionamento de composição quando é nececessário obter uma visão integrada dos *profiles*. Por exemplo, no contexto de desenvolvimento distribuído de software , a etapa de modelagem dos domínios da aplicação pode ser dividida entre os grupos de desenvolvimento a fim de que o esforço seja dividido. Desse modo, tem-se equipes diferentes trabalhando na especificação de domínios onde é necessário representar seus conceitos específicos, os quais a UML não dê suporte. Assim, são criadas linguagens específicas de domínio para representar os conceitos do domínio, sendo usado UML *Profiles* para isto. Porém, em um determinado momento é necessário ter uma visão integrada dos domínios da aplicação e, conseqüentemente, dos *profiles*. Esta integração dos *profiles* pode ser realizada através da *merge strategy*. Na Figura 5.3 é mostrado um exemplo de composição seguindo esta estratégia.

Para o desenvolvimento de uma aplicação Web, por exemplo, divide-se a aplicação em três partes: visão, dados e negócio (ver Figura 1.1. Onde cada parte representa um domínio da aplicação, os quais apresentam conceitos específicos. A fim de modelar esses três domínios, são desenvolvidos três *profiles* um para cada domínio. Para se obter um *profile* único da aplicação Web é possível combinar os três *profiles* estabelecendo um relacionamento de composição usando *merge strategy*.

Definir o relacionamento de composição seguindo a *merge strategy* consiste em definir que as *overlapping parts* do *receiving profile* e do *merged profile* devem ser combinadas a fim de produzir uma *overlapping parts* integrada, como já mencionado.

5.3.2 Semântica de *Merge Strategy*

Nesta Subseção é discutido a semântica que o relacionamento de composição assume quando é definido *merge strategy* como a estratégia de composição. *Merge strategy* é aplicada quando existe a necessidade de obter um *profile* único, sendo resultado da composição de dois ou mais *profiles*.

O relacionamento de composição baseado na *merge strategy* especifica que os elementos de *receiving profile* deve combinar os elementos equivalentes de *merged profile* originando um *resulting profile* que possui todos os *receiving elements* e *merged element* equivalentes combinados. A maneira de identificar os elementos equivalentes, as *overlapping parts*, é a mesma forma utilizada nas outras estratégia. A composição seguindo a semântica de composição *merged strategy* deve respeitar as seguintes regras, como segue:

Regra 01: para todo *receiving element* que possua um *merged element* equivalente isto implica que os mesmos deverão ser combinados. A aplicação desta regra é ilustrada na Figura 5.3, onde é considerado a *default match strategy* como a base para a realização da definição da equivalência. Desse modo, observa-se três casos, como segue:

1. o *stereotype* `Tree.Node` é composta com o *stereotype* `Topology.Node`, tendo `TreeTopology.Node` como resultado.
2. a *enumeration* `Tree.StateKind` é composta com a *enumeration* `Topology.StateKind` tendo `TreeTopology.StateKind` como resultado, o qual tem nome `StateKind`, comum as duas *enumeration*, e os *EnumerationLiteral* `available` e `busy` pertencente a ambas *enumeration*, além disso, o *EnumerationLiteral* `off` originado de `Tree.StateKind`.
3. o *stereotype* `Tree.Edge` é composto com o *stereotype* `Topology.Edge`, tendo `TreeTopology.Edge` como resultado.

Regra 02: para todo *merged element* que não possua um *receiving element* equivalente, o mesmo deverá ser inserido no *resulting profile* sem alterações. Na Figura 5.3, é ilustrado o uso desta regra, onde os *stereotype* `Topology.LocalEdge` e `Topology.MainNode` são inseridos no *resulting profile* sem alterações.

Regra 03: para todo *receiving element* que não possua um *merged element* equivalente, o mesmo deve ser inserido no *resulting element* sem alterações. Na Figura 5.3, é apre-

sentado o uso desta regra, onde os *stereotype* `Tree.Leaf`, `Tree.Root` e `Tree.Search` são inseridos no *resulting profile* sem alterações.

Regra 04: o *resulting profile* não deve apresentar nenhum conflito. Caso exista, é necessário aplicar as *model transformation rules* definidas na Seção 4.4.4. A aplicação desta regra é ilustrada na Figura 5.3, onde o *stereotype* `Tree.MainNode` possui uma referência à *enumeration* `StateKind`. As *enumeration* `Topology.StateKind` e `Tree.StateKind` são equivalentes, o que implica que `Tree.StateKind` é composto com `Topology.StateKind`. Desse modo, surge um *reference conflict*, haja vista `TreeTopology.MainNode` faz referência à *enumeration* `Topology.StateKind` que não existe. Este conflito é solucionado com a mudança de referência da *enumeration* `Topology.StateKind` para `TreeTopology.StateKind` através da aplicação da regra de transformação MTR6 definida na Seção 4.4.4 na página 70.

Regra 05: o *resulting profile* deve respeitar as restrições e as especificações do metamodelo da UML (OMG 2007*c*).

5.4 Semelhança Entre as Estratégias

Estabelecer um relacionamento de composição consiste, basicamente, em: (i) especificar dois modelos de entrada, o *merged profile* e o *receiving profile*; (ii) definir a estratégia de composição a ser seguida; (iii) definir a estratégia de comparação a ser seguida; (iv) obter um modelo de saída, *resulting profile*, como resultado da composição. A partir da definição da estratégia de comparação, a qual define a base para definir a equivalência entre eles, foi observado durante o desenvolvimento deste trabalho que o estabelecimento de um relacionamento de composição pode originar modelos de saída iguais, independente da estratégia de composição adotada.

Diante deste fato, esta seção tem como objetivo realizar uma análise dos casos em que os modelos de saída são iguais. Para isto é mostrado o relacionamento entre os modelos de entrada, as estratégias de composição e as *overlapping parts*.

A partir dos modelos de entrada *merged profile* e *receiving profile* é possível obter *resulting profile* iguais independente da estretágia de composição especificada. Ou seja, tendo um *profile A* e um *profile B* independente da estratégia de composição a ser seguida, sempre originará um *profile AB*.

Sendo assim, a comparação entre dois modelos de entrada pode originar um relacionamento entre o conteúdo os quais são: *total*, *partial*, *subset* e *empty*, sendo estes descritos as seguir:

- *Total Overlapping Parts:* representa que o *receiving profile* e *merged profile* possuem conteúdos iguais. Ou seja, tudo que é encontrado em *receiving profile* também é

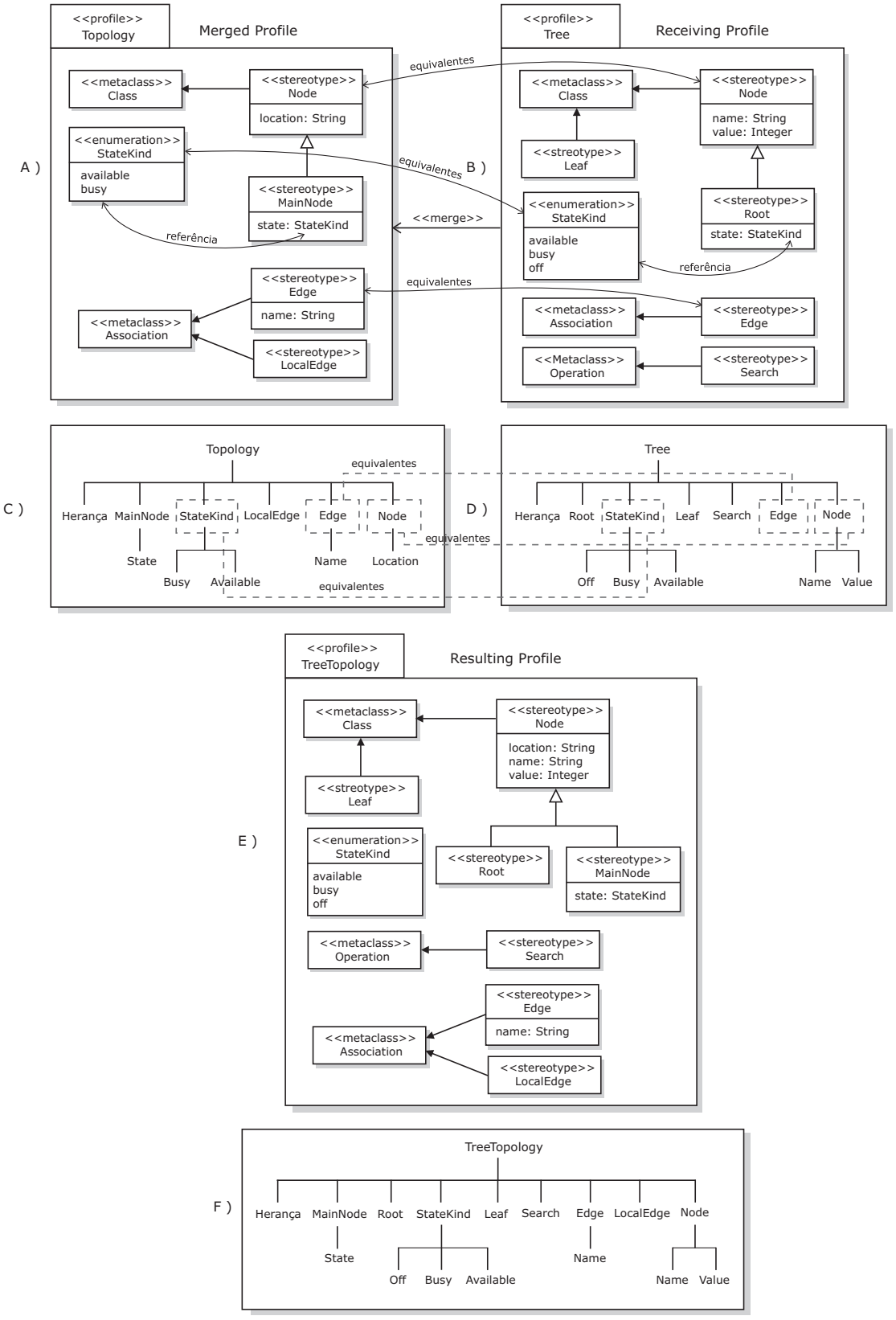

Figura 5.3: Exemplo de composição seguindo a *merge strategy*

encontrado em *merged profile*. Assim, os *receiving element* e os *merged element* são iguais.

- *Partial Overlapping Parts: receiving profile* e *merged profile* possuem parte dos seus conteúdos equivalentes. Ou seja, parte do que é encontrado no *receiving profile* também é encontrado no *merged profile*.

- *Subset Overlapping Parts: receiving profile* possui todo o conteúdo de *merged profile*, ou seja, os *merged elements* fazem parte dos *receiving elements*. Logo, os *merged elements* são um subconjunto do *receiving element*.

- *Empty Overlapping Parts: receiving profile* não possui nenhum conteúdo do *merged profile*. Ou seja, *merged elements* e *receiving element* são disjuntos.

A fim de definir os termos anteriormente especificados seguindo a definição da teoria dos conjuntos, tem-se:

Receiving Profile: representa uma coleção de possíveis elementos, sendo estes elementos definidos na Figura 4.4 na página 52. Estes elementos são representados pela letra R.

Merged Profile: representa uma coleção de elementos, sendo estes elementos ilustrados na Figura 4.4 na página 52. Estes elementos são representados pela letra M.

Total Overlapping Parts: representa a coleção dos elementos de R e M que são iguais. Isto implica que: $R \cap M = R$ e $R \cap M = M$.

Partial Overlapping Parts: representa a coleção dos elementos de R e M que são parcialmente iguais. Isto implica que: $R \cap M = \{x: x \in A \text{ e } x \in B\}$

Subset Overlapping Parts: representa que M é um subconjunto de R. Isto implica que: $M \subset R$ que equivale a $R \cup M = R$ e $R \cap M = M$.

Empty Overlapping Parts: representa que M e R são disjuntos. Isto implica que: a $R \cup M = \varnothing$

A Figura 5.4 mostra uma análise da composição de dois *profiles* tendo como base da comparação a *complete match strategy*. Desse modo, observa-se que:

Caso 01: o modelo de saída é igual ao *receiving profile* quando a estratégia de composição adotada é *override strategy* ou *merged strategy*, e tem-se *total overlapping parts* ou *subset overlapping parts*.

Caso 02: o modelo de saída é igual a $A \cup B$ quando adota-se *override strategy* ou *merge strategy* como estratégia de composição, tendo *partial overlapping parts*; e quando tem-se *merge strategy* como estratégia de composição e tem-se *empty overlapping parts*.

Caso 03: o modelo de saída é igual a `A` ∪ (`B` - (`A` ∩ `B`)): quando adota-se *union strategy* como estratégia de composição, tendo *total overlapping parts*, *partial overlapping parts*, *subset overlapping parts* e *empty overlapping parts*; e quando tem-se *override strategy* como estratégia de composição e tem-se *empty overlapping parts*.

	Receiving Profile (A) and Merged Profile (B)											
Override Strategy	X	X	X	X								
Union Strategy					X	X	X	X				
Merge Strategy									X	X	X	X
Total Overlapping Parts	X				X				X			
Partial Overlapping Parts		X				X				X		
Subset Overlapping Parts			X				X				X	
Empty Overlapping Parts				X				X				X
Output Model	A	*	A	**	**	**	**	**	A	*	A	**

* A U (B-(A ∩ B))
** A U B

Figura 5.4: Diferentes saídas a partir da mudança da estratégia de composição

Capítulo 6

Extensão do Metamodelo da UML

Com o objetivo de tornar a UML capaz de expressar o mecanismo de composição proposto, foi criado uma extensão do seu metamodelo. Sendo assim, a UML é adaptada ao domínio de composição de modelos e a abordagem proposta passa a ter uma forma de representação. Dessa forma, com a finalidade de realizar a composição dos *profiles*, este trabalho apresenta uma abordagem que é baseada na composição de modelos de entrada produzindo modelos de saída, como citado anteriormente. A composição é realizada levando em consideração: estratégias de comparação, estratégias de composição, os modelos de entrada que apresentam equivalentes, regras de transformação, regras de composição e regras de equivalência, como mencionado nos capítulos anteriores. Desse modo, criar uma representação deste mecanismo no contexto da UML implica em tornar possível representar este conteúdo.

O mecanismo de composição especificado e desenvolvido é representado através de um relacionamento de composição, identificado como *composition relationship*. Porém, este relacionamento apresenta característica (por exemplo as *merge rules*, *composition strategy*, e entre outras) que não podem ser representadas com a UML. Sendo assim, é proposto uma extensão do metamodelo da UML para torná-la capaz de representar este relacionamento.

Este capítulo apresenta uma proposta de extensão do metamodelo da UML para representar o relacionamento de composição. O Capítulo possui com as seguintes seções:

- *Metamodelo da UML*: esta seção apresenta considerações sobre o metamodelo da UML.

- *Definição do Relacionamento de Composição:* esta seção se detém na definição de uma extensão do metamodelo da UML. Este metamodelo é responsável por especificar o relacionamento de composição.

- *Regras de Boa Formação:* nesta seção são apresentadas regras de boa formação, a fim de representar informações semânticas adicionais e garantir correta construção

do relacionamento definido na extensão do metamodelo da UML.

6.1 Metamodelo da UML

A OMG definiu a especificação da UML seguindo a abordagem da metamodelagem. O metamodelo da UML especifica a sintaxe e a semântica da linguagem, sendo projetado seguindo os princípios de: (i) modularidade; (ii) *layering*; (iii) *partitioning*; (iv) extensibilidade; (v) reuso. O metamodelo da UML é descrito usando:

- **sintaxe abstrata:** representada pelos diagramas UML mostrando as metaclasses que definem os construtores da linguagem (como por exemplo: *Association*, *Operation*, *Property*, *Parameter*, entre outros) e seus relacionamentos. Uma descrição informal em linguagem natural que descreve cada um destes construtores e seus relacionamentos.

- **regras de boa formação:** trata-se de um conjunto de regras de boa formação que são descritas usando OCL e linguagem natural.

- **semântica:** para cada construtor da linguagem é associado um significado, para isto é usado linguagem natural.

O uso da linguagem natural para especificar os elementos que compõem a linguagem, como a semântica, as regras de boa formação e sintaxe abstrata citados anteriormente, favorece o surgimento de ambigüidade. Conseqüentemente, implica em dificuldades em estender o metamodelo da linguagem. O trabalho não se deterá em apresentar uma especificação formal da extensão da UML, e sim uma descrição do relacionamento de composição seguindo as recomendações de extensão do metamodelo apresentada da especificação da linguagem. Porém, evitando ambigüidade e inconsistência.

6.2 Definição do Relacionamento de Composição

Para especificar os tipos de elementos que podem participar do relacionamento de composição foi levado em consideração o modo como a UML define os participantes do relacionamento de herança, do relacionamento de composição (*PackageMerge*), e o padrão de projeto *composite*. Na especificação da UML, são definidos dois construtores abstratos: (i) *Classifier*, todo os elementos que podem participar de um relacionamento de herança é filho do mesmo; (ii) *PackageElement*, todos os elementos que podem participar de um relacionamento de composição de pacote deve herdá-lo.

Com isso, são inseridos dois conceitos no metamodelo da UML:

ComposableElement: trata-se dos modelos de entradas do relacionamento de composição, ou seja, os elementos que podem ser compostos. *ComposableElement* é uma classe abstrata e um *NamedElement* (definido na UML e mostrado na Figura A.9 no Anexo A, na página 146).

CompositeElement: é um *ComposableElement* que é composto por outros *ComposableElement* ou *ComposableElement*. Cada elemento que faz parte de um *CompositeElement* é tratado separadamente durante a composição. *CompositeElement* é uma classe abstrata que funciona como um contêiner.

A introdução destes dois conceitos no metamodelo da UML é representado através da sintaxe abstrata ilustrada na Figura 6.1, a qual define os tipos de modelos que podem participar do relacionamento de composição através da estrutura do padrão de projeto *composite* (Gamma et al. 1995). Isto implica em algumas conseqüências, tais como:

1. uma hierarquia de modelos que consiste de elementos "primitivos" e "compostos" (*CompositeElement*). Desse modo, *ComposableElement* como *Property* pode fazer parte de *CompositeElement* que funcionam como um contêiner. A distribuição dos elementos "primitivos" nos compostos se configura de acordo com o apresentado na Figura 4.4 e de acordo com a definição do metamodelo da UML.

2. torna fácil acrescentar novos modelos a serem compostos, sendo necessário apenas extender *ComposableElement* ou *CompositeElement*, dependendo da natureza do modelo;

3. o acréscimo excessivo de novos elementos poderá configurar uma desvantagem, algo natural do padrão *composite*. Haja vista, a dificuldade de restringir a qual elemento "composto" os novos elementos poderiam fazer parte. Isto é solucionado através das restrições na especificação da UML, a qual define o que cada um tipo de modelo pode ter, e em qual modelo o mesmo pode fazer parte. A distribuição dos elementos "primitivos" nos compostos se configura de acordo com apresentado na Figura 4.4 e de acordo com a definição do metamodelo da UML.

Com o objetivo de expressar o relacionamento de composição no contexto do metamodelo da UML, foi criado uma extensão do metamodelo, o qual é ilustrado na Figura 6.2, e tem as seguintes características:

- o relacionamento de composição é representado por *CompositionRelationship*. Ao passo que *CompositionRelationship* faz reuso de *DirectedRelationship*, o mesmo disponibiliza uma especialização de *DirectedRelationship* no contexto de composição de modelos;

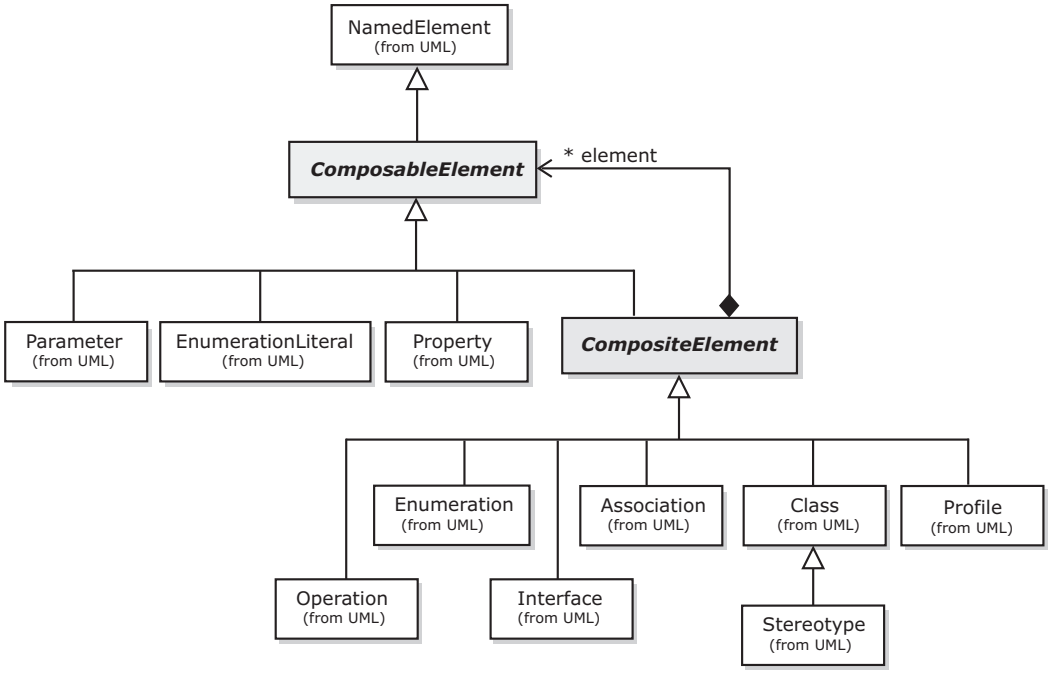

Figura 6.1: Especificação dos elementos que podem ser compostos no mecanismo

- apenas é especificado entre elementos que são passíveis de serem compostos. Este elementos são definidos na Seção 4.2, "Especificando os Modelos de Entrada" na página 51;

- o relacionamento de composição faz uso de regras as quais precisam ser representadas, sendo idealizadas dentro da extensão baseada na estrutura de *Constraint* definida na UML.

- o relacionamento de composição é caracterizado pelo suporte a diferentes formas de composição, isto é verificado através da definição da *enumeration CompositionStrategyKind*.

- o relacionamento de composição faz uso de regras para determinar a equivalência entre os modelos (*Match Rules*), para realizar transformações dos modelos (*Model Transformation Rules*) e para executar a composição dos modelos (*Merge Rules*), como já descrito. *Rule* foi definida para representar estas regras.

- os operadores do mecanismo de composição são representado por: *Match*, *CompositionStrategy*, *Merge*, e *Transformation*.

O metamodelo definido na Figura 6.2 define a sintaxe abstrata do relacionamento de composição. Sendo assim, com o objetivo de deixar claro o papel de cada elemento, é realizado uma descrição, como segue:

CompositionRelationship: especifica um relacionamento semântico entre dois *CompositeElement*. Este construtor relaciona um *receiving* a um *merged*, representando um relacionamento navegável do *receiving* ao *merged*. *CompositionRelationship* possui um conjunto de tuplas cujo valores especificam a instância do relacionamento. A instância do *CompositionRelationship* é chamada de *merge*.

Generalização:

- *CompositionRelationship* estende *DirectedRelationship* definido no metamodelo da UML.

Atributos:

- *resulting: CompositeElement*, especifica o resultado da composição entre dos modelos de entrada. Trata-se de um subconjunto de *Namespace::ownedMember* definido no metamodelo da UML.

Associações:

- *merged:* trata-se de um *CompositeElement* que será compostos com *receiving*. A cardinalidade "1" representa que todo o relacionamento de composição deve ter um *CompositeElement* (*profile*) no papel de *merged profile* para ser válido. Representa um subconjunto de *DirectedRelationship::target* do metamodelo da UML;
- *receiving:* trata-se do *CompositeElement* que terá seu conteúdo estendido pelo *merged*. A cardinalidade "1" representa que todo o relacionamento de composição deve ter um *CompositeElement* (*profile*) no papel de *receiving profile* para ser válido.
- *strategy:* especifica a estratégia que deve ser aplicada à composição. A cardinalidade "1" representa que todo o relacionamento de composição deve ter uma estratégia de composição associado a ele;
- *transformation:* especifica que o relacionamento de composição pode ter transformações associadas ao mesmo, sendo isto representado através da cardinalidade "0..1".
- *match:* especifica qual estratégia de comparação que deve ser seguida durante o relacionamento de composição. A cardinalidade "1" representa que todo relacionamento de composição deve ter uma *Match Strategy* associada.
- *merge:* especifica qual estratégia de composição deve ser seguida durante o relacionamento de composição. A cardinalidade "1" representa que todo relacionamento de composição deve ter uma *Merge Strategy* associada.

Constraint:

- O relacionamento de composição deve ser estabelecido entre apenas dois *profiles*. Isto é expressado em OCL como segue:
  context *CompositionRelationship*
  self.receiving.oclIsKindOf(Profile) and
  self.merged.oclIsKindOf(Profile) and
  (self.receiving.lowerBound() = 1 and self.receiving.upperBound() = 1)
  (self.merged.lowerBound() = 1 and self.merged.upperBound() = 1)
- A visibilidade do *CompositionRelationship* é público ou privado.
  self.visibility = #public or self.visibility = #private

Operações Adicionais:

- A operação *getCompositionStrategy()* retorna o tipo da *composition strategy* que é especificada para o relacionamento de composição. Isto é expressado em OCL como segue:
  CompositionRelationship::getCompositionStrategy(): CompositionStrategyKind
  getCompositionStrategy = **if** self.strategy->notEmpty()
  then self.strategy.kind
- A operação *getMatch()* retorna o tipo da *match strategy* que é especificada para o relacionamento de composição. Isto é expressado em OCL como segue:
  CompositionRelationship::getMatch(): MatchKind
  getMatch = **if** self.match->notEmpty
  then self.match.kind
- A operação *getMatchThreshold()* retorna o limite do grau de similaridade que deve ser considerado no relacionamento de composição. Isto é expressado em OCL como segue:
  CompositionRelationship::getMatchThreshold(): String
  getMatchThreshold = **if** self.match->notEmpty
  then self.match.threshold

Semântica:

Um relacionamento de composição estabelece um relacionamento de composição entre *receiving* e *merged*, onde o principal objetivo é representar a integração do conteúdo dos mesmo, produzindo um *CompositeElement* que representa o resultado da composição. A instância de *CompositionRelationship* é chamada de *merge*. A navegabilidade do relacionamento de composição significa que o *merge* deve ser estabelecido em um determinado sentido. O estabelecimento da origem e destino

do relacionamento define o papel dos *CompositeElements* na instância do relacionamento de composição. A origem representa o *receiving* e o destino o *merged* (*receiving* → *merged*)

Notação:

Um relacionamento de composição é representado por um senta de linha sólida, partindo do *receiving* ao *merged*.

ComposableElement: trata-se de uma metaclasse abstrata a qual representa os elementos que podem ser composto. Estes elementos são operandos na definição das regras de *match, merge*, e *model transformation*.

CompositeElement: trata-se de uma metaclasse abstrata a qual representa os elementos que é estabelecido o relacionamento de composição. O relacionamento de composição especificado nesta proposta é definido entre dois *profiles*.

CompositionStrategy: é responsável pela associação de uma estratégia de composição a um relacionamento de composição. Faz o papel do *Composition Strategy Operator*.

Atributos:

- *kind: CompositionStrategyKind*, representa a estratégia de composição de fato. Os possíveis valores de *kind* são definidos na *enumeration CompositionStrategyKind*.

Transformation: é responsável pela representação da resolução de problemas através de *model transformation rules*. Com esta metaclasse, a UML passa a ser capaz de expressar *Model Transformation Operator*. Como uma composição pode surgir conflitos/problemas, ou não. A cardinalidade "0..1" indica que um relacionamento de composição pode ter *model transformation rules* associado, ou não.

Associações:

- *ownedTransformationRule: Rule [*]*, consiste de uma coleção de *Rule* (*model transformation rules*). A cardinalidade "*" especifica que é possível ter uma coleção de regras, ou nenhuma.

Atributos:

- *isRequired: Boolean*, quando o seu valor é igual `true` indica que as especificações das *model transformation rules* devem ser aplicadas ao relacionamento de composição. Os possíveis valores de *kind* são os definidos na *enumeration CompositionStrategyKind*.

- *conflictDescription: String*, trata-se da descrição dos conflitos ao qual as *model transformation rules* são aplicadas.

Match: é responsável pela representação da *match strategy* aplicada ao relacionamento de composição. Trata-se de um *NamedElement*.

Associações:

- *ownedMatchRule: Rule [*]*, consiste de uma coleção de *Rule* (*match rules*). A cardinalidade "1..*" especifica que *Match* sempre terá uma referência à *Rule*, a qual define as *match rules*.

Atributos:

- *kind: MatchKind*, representa as possíveis *match strategies* que podem ser associadas ao relacionamento de composição. Os possíveis valores de *kind* são os definidos na *enumeration MatchKind*.

Merge: é responsável pela representação da composição no relacionamento de composição. Trata-se de um *NamedElement*. Esta entidade dá UML a capacidade de representar o *merge operator*.

Associações:

- *ownedMergeRule: Rule [*]*, consiste de uma coleção de *Rule* (*merge rules*). A cardinalidade "1..*" especifica que *Merge* sempre terá uma referência a *Rule*, a qual define as *merge rules*).

Rule: este construtor é usado para representar as regras usadas no relacionamento de composição. Sendo assim, as regras de comparação, composição e transformação de modelos podem ter sua sintaxe representadas fazendo uso desta.

Generalização:

- *Rule* é uma extensão de *Operation* definida no metamodelo da UML.

Associações:

- *ruleExpression: RuleExpression [*]*, consiste de uma coleção de *RuleExpression*. A cardinalidade "*" especifica que *Rule* pode ter, ou não, uma referência a *RuleExpression*. É utilizada para a construção da sintaxe das regras.
- *ruleValues: RuleValues [*]*, consiste de uma coleção de *RuleValues*. A cardinalidade "*" especifica que *Rule* pode ter, ou não, uma referência a *RuleValues*. Um regra para ser válida a mesma deve satisfazer os valores especificados em *ruleValues*.

- *ownedElement: ComposableElement [*]*, representa os operandos das regras.
- *raisedConflict: Conflict [*]*, a execução das regras podem gerar exceções durante a sua aplicação (por exemplo, caso uma *Post-Condition* não seja satisfeita gerará uma exceção) estas exceções são representadas como uma coleção de *Conflict*.
- *precondition: Constraints [*]*, especifica um conjunto de restrições que a regra deve satisfazer para ser executada. Redefinida de *Operation::precondition*
- *postcondition: Constraints [*]*, especifica um conjunto de restrições que devem encontradas após aplicação da regra. Por exemplo, não deve existir conflitos de nomes, ou seja, modelos com nomes iguais dentro de uma mesma *Namespace*. Redefinida de *Operation::postcondition*

Semântica:

Rule adiciona ao metamodelo da UML a capacidade de representar as regras definidas no mecanismo de composição. Pode ser expressada em linguagem natural ou em algum linguagem específica (por exemplo em *Alloy*, OCL, Java e etc). Para que uma regra seja executa corretamente a mesma deve satisfazer as *postcondition* definidas. Ao passo que, *precondition* deve ter sido satisfeitas.

Conflict: especifica os problemas que surgem durante a execução de uma *Rule*.

Generalização:

- *Conflict* é um *Type*, o qual é definido no metamodelo da UML.

RuleExpression: representa uma árvore estruturada de símbolos que denota um determinado valor.

Generalização:

- *RuleExpression* é um *Expression*, a qual é definido no metamodelo da UML.

Semântica:

RuleExpression especifica um *node* em uma árvore de expressão. Dado um *node*, este será um operando ou um operador. Se for uma operando será considerado um símbolo terminal. Caso contrário será um operador que atuará sobre alguns operandos. É utilizada para construir as regras utilizadas no relacionamentos de composição, as *match rules*, *merge rules*, e as *model transformation rules*.

Notação:

Uma *RuleExpression* com operandos é representada com um símbolo e seus parâmetros representando os operandos da mesma. Por exemplo, a expressão da *match rule* para *enumeration*, MatchEnumeration(Enumeration rcv, Enumeration mrgd).

RuleValues: é a especificação e representação de um conjunto de instâncias.

Generalização:

- *RuleValues* é uma *ValueSpecification*, que é definido no metamodelo da UML.

Semântica:

RuleValues é necessário que os valores atribuídos sejam valores válidos ao contexto da composição, ao qual o mesmo é aplicado. Representam os valores que são manipulados pelas regras.

MatchKind: este construtor é uma *Enumeration* que define literais para determinar a estratégia de comparação a ser seguida durante a composição.

Descrição:
MatchKind possui os seguintes literais:

- *default*, usado para representa a *default match strategy.*
- *partial*, usado para representa a *partial match strategy.*
- *complete*, usado para representa a *complete match strategy.*

Semântica:

MatchKind é usado para especificar os tipos de *match strategy* suportadas no relacionamento de composição. Sendo assim, os possíveis valores para *match strategy* passado como parâmetro para o relacionamento de composição são definidos neste construtor. O significado de cada literal (*default, partial, complete*) é definido nas estratégias de comparação especificadas no Capitulo 4.

CompositionStrategyKind: este construtor é uma *Enumeration* que define literais para determinar a estratégia de composição a ser seguida durante a composição.

Descrição:
CompositionStrategyKind possui os seguintes literais:

- *union*, usado para representa a *union composition strategy.*
- *override*, usado para representa a *override composition strategy.*
- *merge*, usado para representa a *merge composition strategy.*

Semântica:
CompositionStrategyKind é usado para especificar os tipos de *composition strategy* suportadas no relacionamento de composição. Sendo assim, os possíveis valores

para *composition strategy* passado como parâmetro para o relacionamento de composição são definidos neste construtor. O significado de cada literal (*union, override, merge*) é definido nas estratégias de composição especificadas no Capitulo 4.

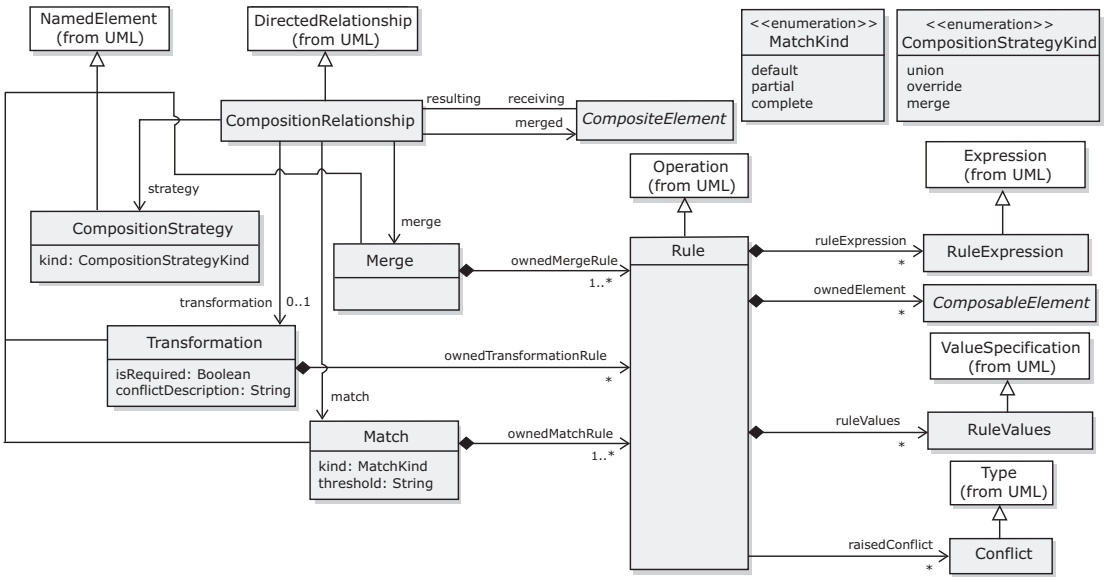

Figura 6.2: Diagrama de relacionamento de composição

6.2.1 Regras de Boa Formação

A fim de assegurar que a definição do relacionamento de composição definido seja garantida e representar informações semânticas adicionais são apresentadas algumas regras de boa formação. O modelo seguido para especificar estas regras é o apresentado na especificação da UML (OMG 2007*c*), como segue:

Regra 01 : o relacionamento de composição deve ser definido entre *CompositeElement* do tipo *Profile*.

context *CompositionRelationship*
self.receiving.oclIsKindOf(Profile) and
self.merged.oclIsKindOf(Profile)

Regra 02 : todo relacionamento de composição deve ter uma estratégia de composição.

context *CompositionRelationship*
self.strategy.oclIsKindOf(CompositionStrategy) and
(self.strategy.lowerBound() = 1 and self.strategy.upperBound() = 1)

Regra 03 : todo relacionamento de composição deve ter ao menos duas regras associado.

context *Merge*
if self.ownedMergeRule.oclIsKindOf(Rule)
then self.ownedMergeRule.lowerBound() = 1

context *Match*
if self.ownedMatchRule.oclIsKindOf(Rule)
then self.ownedMatchRule.lowerBound() = 1

Regra 04 : uma regra pode ter *Conflicts* gerados da sua execuação.

context *Rule*
self.raisedConflict.oclIsKindOf(Conflict) implies
(self.raisedConflict.lower() = 0 or self.raisedConflict.lower() = 1)

Regra 05 : o resultado da composição é um *ComposositeElement*.

context *CompositionElement*
self.resulting.oclIsKindOf(CompositeElement)

Uma vez definido a extensão, é apresentado um exemplo de uso da extensão do metamodelo o qual é ilustrado na Figura 6.3. Sendo assim, para dado um *Profile A* e um *Profile B*, deseja-se representar o relacionamento de composição entre os mesmo. Para isto é necessário definir a estratégia de comparação e a estratégia de composição que deve ser seguida. As duas estratégias para este exemplo são: *default match* e *merge strategy*. Assim, a configuração da composição fica definida: (*i*) `Profile A`, como o *receiving profile*; (*ii*) `Profile B`, como o *merged profile*; (*iii*) *default match*, como a estratégia de comparação e é representada por `DM`; (*iv*) *merge strategy*, como a estratégia de composição e é representada por `MS`.

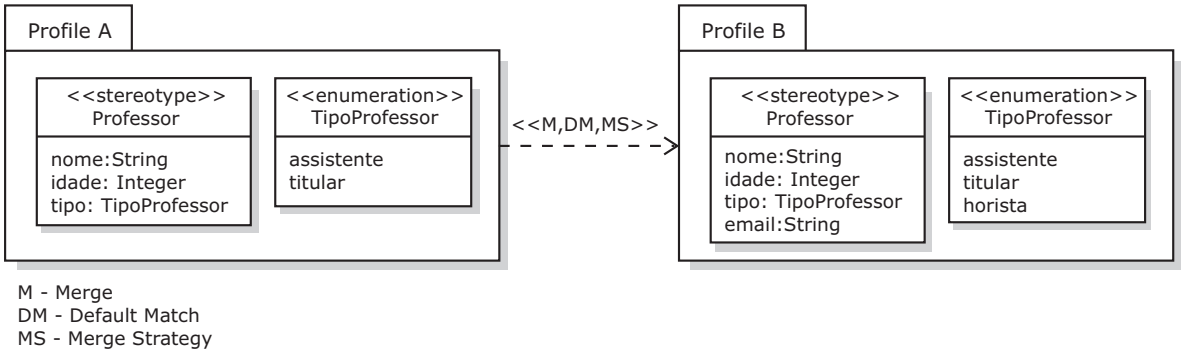

Figura 6.3: Exemplo de uso da extensão do metamodelo

Capítulo 7

Modelo Formal do Mecanismo de Composição de *Profiles*

O mecanismo de composição da UML é definido usando linguagem natural, o que proporciona o surgimento de inconsistência e ambiguidade. A fim de evitar isto, o mecanismo é modelado utilizando a linguagem formal Alloy. Com isso, o modelo formal criado é analisado de forma automática utilizado o *Alloy Analyzer*, a fim de validar o modelo.

Este capítulo tem cinco objetivos: (i) criar o modelo formal do mecanismo de composição de UML *Profiles* em Alloy; (ii) especificar o metamodelo que define os UML *profiles* em *Alloy*, com isso é criado um semântica formal da especificação dos *profiles*; (iii) realizar verificações baseadas na definição da semântica formal da especificação dos *profiles*; (iv) realizar análise automática usando o *Alloy Analyzer*; (v) verificar algumas propriedades algébricas do mecanismo.

A especificação formal e análise automática do metamodelo dos *profiles* e do mecanismo de composição envolve responder várias questões, tais como: quais critérios deve ser levado em consideração para realizar a análise? Como a análise destes critérios pode ser realizada? Uma vez que o metamodelo dos *profiles* é definido usando linguagem natural, como é possível formalizá-lo? Como identificar as partes equivalentes entre os *profiles*? Como implementar as estratégias de comparação e de composição?

As respostas destas perguntas levam a ter as seguintes contribuições:

1. *Uma semântica formal do metamodelo dos profiles.* Para isto, foi traduzido a especificação dos *profiles* para *Alloy* e uma vez que a análise é realizada a nível de metamodelo isto implica que a mesma é válida para qualquer *profile*.

2. *Formalização do mecanismo de composição.* Com a formalização em *Alloy* é possível fazer uso do *Alloy Analyzer* para verificar algumas das suas propriedades. Uma vez conhecida estas propriedades é possível fazer um melhor uso do mesmo.

Diante deste contexto, este capítulo é organizado com as seguintes seções:

- *Alloy e Alloy Analyzer:* é apresentado uma descrição da linguagem *Alloy* e do *Alloy Analyzer*, os seus conceitos e suas características.

- *Modelagem do Mecanismo de Composição de Profiles:* é apresentado uma semântica formal do metamodelo dos *profiles* e a modelagem do mecanismo de composição em *Alloy*.

- *Análise do Modelo Formal:* é realizado uma análise do modelo formal usando o *Alloy Analyzer* .

- *Propriedades Algébricas:* uma verificação de algumas propriedades algébricas do mecanismo de composição com o *Alloy Analyzer* .

7.1 *Alloy e Alloy Analyzer*

Uma forma de realizar a verificação formal do mecanismo de composição de profile é traduzindo a especificação dos UML *profiles* e a definição do mecanismo de composição em uma linguagem formal e depois realizar uma análise neste modelo formal. Sendo assim, é utilizado a *Alloy* e o *Alloy Analyzer*. A *Alloy* trata-se de uma linguagem formal baseada em teoria de conjuntos, lógica de primeira e é fortemente influenciada pela notação orientada a objeto. Modelos representados em *Alloy* são entradas para a sua ferramenta de suporte o *Alloy Analyzer*, o qual basicamente trata-se de um resolvedor de restrições.

A *Alloy* e o *Alloy Analyzer* tem sido usado com sucesso para modelar: (*i*) modelos orientados a objeto (Mostefaoui & Vachon 2007); (*ii*) composição de modelos; (iii) formalização de modelos orientados a objeto (Massoni et al. 2004, Bourdeau & Cheng 1995); (*iv*) modelagem de sistemas críticos, incluindo sistema de controle de tráfico aéreo (Dennis 2003) e em máquinas de terapias com proton (Dennis et al. 2004).

Além disto, como a linguagem é influenciada pela notações da modelagem orientada a objeto, isto torna mais fácil a classificação de objetos e suas propriedades. Por estas razões, foi escolhido usar *Alloy* e acredita-se que a linguagem e sua ferramenta de suporte sejam bastante úteis e ofereçam similar benefícios na realização de análise automática do mecanismo de composição de *profiles* desta dissertação.

Um modelo representado em *Alloy* consiste basicamente de *signatures*, *facts*, *predicates* e *functions*, os quais são descritos como segue (Jackson 2002, Jackson 2006):

- *signatures:* são semelhantes as classes representadas nas linguagens orientada a objeto. As *signatures* possuem *fields* (os atributos nas classes) e podem ser estendidas por outras *signatures* (similar ao mecanismo de herança em OO). Além disso, é possível gerar instância de *signatures*, sendo estas instâncias o objeto da análise.

- *facts:* representa restrições semânticas que são aplicadas às *signatures*. Os *facts* são definidos usando lógica de predicado. Sendo assim, usam conectivos lógicos (por exemplo, && (and), || (or), ⇒ (implica), entre outros), quantificadores (como, *all* (para todo), *some* (existe)) e operadores (como, + (operador união), = (operador igualdade) e ^ (operador que representa transitividade));

- *functions:* comparando com o paradigma orientado a objeto, as *functions* seriam os métodos das classes ou as *operation* definidas no metamodelo da UML;

- *predicates:* é um tipo especial de *functions* que retorna apenas valores booleanos. A estrutura dos *predicates* é igual à apresentada nos *facts* e nas *functions*.

- *modules*: assim como a UML, *Alloy* foi idealizada com o princípio da modularidade em mente. Desta forma, a linguagem é organizada em módulos (os pacotes em UML). Cada modelo pode ter *signatures*, *functions*, *predicates* e *facts*. Da mesma forma que os pacotes em UML podem ter acesso, por exemplo, ao conteúdo de outros pacotes através do relacionamento de *«import»*, os *modules* em *Alloy* também podem ter acesso ao conteúdo de outros *modules*.

A fim de fazer uso destes conceitos, ambientar o leitor aos conceitos da linguagem e facilitar o entendimento, é apresentado um *profile* na Figura 7.1, o qual é modelado em *Alloy*. Este *profile* em *Alloy* é apresentado no Código 7.1, tendo:

1. as *signatures* `Edge` e `Node` estendem a *signature* Stereotype; as *signatures* `Leaf` e `Root` estende a *signature* `Node`; a *signature* `Tree` estende a *signature* `Profile`, assim como a herança em orientação a objeto;

2. a *signature* `Edge` possui três *fields*: `specification`, `sourceNode` e `endNode` com multiplicidade 1 e todos do tipo `String`.

3. a *signature* `Profile` possui apenas um *field*, `stereotype` com multiplicidade 0..* (representado por `set`) do tipo `Stereotype`;

4. a *signature* `Node` possui dois *fields*, `location` e `name` com multiplicidade 1 e ambos do tipo `String`;

5. a *signature* `Leaf` possui um *field*, `value` de multiplicidade 1 e de tipo inteiro.

6. a *signature* `Roof` possui quatro *fields*: `name` do tipo `String`, `value` do tipo inteiro, `leftNode` e `rightNode` ambos do tipo `Node`;

7. a *signature* `Roof` possui quatro *fields*: `edge` do tipo tipo `Edge`, `node` do tipo `Node`, `leaf` do tipo `Leaf` e `root` do tipo `Root`, todos com multiplicidade 1;

8. o *predicate* `checkNode` tem dois `Node` como parâmetros e retorna `true` se eles possuem nomes iguais;

9. o *predicate* `leafEqual` tem dois `Leaf` como parâmetros e retorna `true` se eles possuem valores iguais;

10. o *fact* `RootFact` adiciona uma restrição a *signature* `Root`, se todos os *fields* são iguais implica que os `Root` são equivalentes.

11. todas *signatures*, *functions*, *predicates* e *facts* são definidas no `module Tree`.

Código 7.1: Um *profile* representado em *Alloy*

```
1  module Tree
2  sig String{}
3  sig Stereotype {}
4  sig Profile {                    sig Edge extend Stereotype {
5     stereotype: set Stereotype       specification: String,
6  }                                   sourceNode: String,
7                                      endNode: String
8                                   }
9
10 sig Node extend Stereotype{      sig Leaf extend Node {
11    location: String,                value: Int
12    name: String                  }
13 }
14
15 sig Root extend Node {           sig Tree extend Profile {
16    name: String,                    edge: Edge,
17    value: Int,                      node: Node,
18    leftNode: Node,                  leaf: Leaf,
19    rightNode: Node                  root: Root
20 }                                }
21
22 pred checkNode(n1: Node, n2: Node) {
23    n1.name = n2.name
24 }
25
26 pred leafEqual(m1: Leat, m2: Leaf) {
27    m1.value = m2.value
28 }
```

```
29
30 fact RootFact {
31     all r1, r2: Root |
32     (r1.name = r2.name && r1.value = r2.value &&
33     r1.leftNode.location = r2.leftNode.location &&
34     r1.rightNode.location = r2.rightNode.location) => r1 = r2
35 }
```

Um modelo representado em *Alloy* corresponde, possivelmente, um conjunto de instâncias válidas. Para obter um instância de um modelo é necessário, também, instanciar todos os *fields* (as *signatures*) que ele faz referência de acordo com a multiplicidade destes (Mostefaoui & Vachon 2007, Jackson 2006). Por exemplo, para criar uma instância da *signature* `Edge` é necessário criar os três *fields specification*, *sourceNode* e *endNode* e instanciar a *signature* `String`.

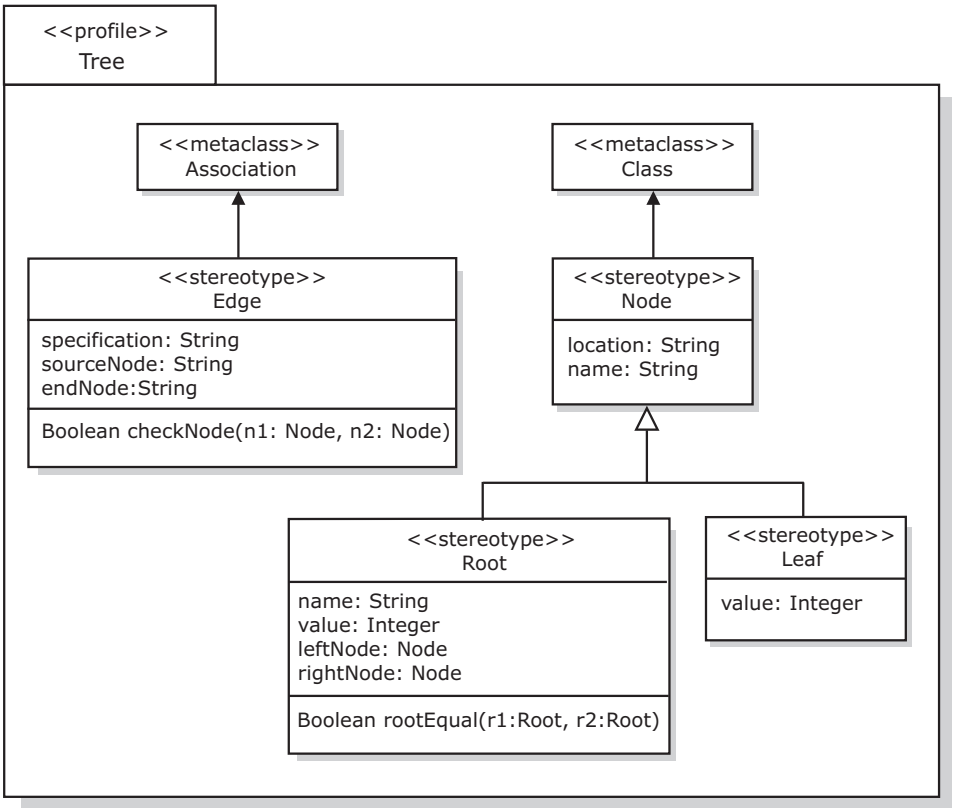

Figura 7.1: Exemplo de um *profile* em *Alloy*

Para um modelo em *Alloy* ser considerado válido, o mesmo não deve violar nenhuma restrição a qual é definida através dos *facts*. Um modelo é considerado inconsistente se não existe uma instância que satisfaça a todas as restrições definidas através dos *facts*. Sendo assim, para ter uma instância, por exemplo, de um *Stereotype* válida, é necessário que sejam satisfeitas todas as restrições relacionadas a ele.

Outro exemplo seria, uma instância de `Tree` deve respeitar as restrições de `RootFact` para se considerada válida, se não existir nenhuma instância que satisfaça as restrições será considerado um modelo inconsistente.

7.1.1 Análise com *Alloy*

O *Alloy Analyzer* pode realizar dois tipos de análise sobre o modelos de entradas: (i) checar as instâncias do modelo; (ii) verificar asserções. Com isto em mente, para realizar a análise da proposta do mecanismo de composição a verificação de asserções será levada em conta, assim como checar instâncias do modelo. Onde verificar asserções consiste em analisar se o que é definido no modelo apresentam valores verdadeiros ou falsos.

O objetivo de utilizar asserções consiste em definir uma "afirmação" e verificar se esta é válida para toda instância do modelo. Quando o *Alloy Analyzer* verifica as asserções, o mesmo retorna *true* se a asserção é válida. Caso contrário, retorna um contra-exemplo a fim de mostrar que a asserção é *false*.

A fim de verificar as asserções, o *Alloy Analyzer* analisa todas as possíveis instâncias do modelo, quando é encontrado um contra-exemplo a análise é finalizada e o contra-exemplo é mostrado. Caso o número de instância seja muito grande, ou tendendo ao infinito, a análise torna-se impraticável. Um alternativa para resolver este problema é a definição do escopo, no qual a análise deve ser realizada (Jackson 2006). Na definição de um escopo, é definido o número máximo de instâncias possíveis para as *signatures*.

O *Alloy Analyzer* tem a capacidade de considerar apenas as instâncias do modelo onde o número que define o valor máximo possível de instância das *signatures* é encontrado. Por exemplo, quando é definido o valor máximo de instâncias das *signatures* é 5, isto implica que o *Alloy Analyzer* analisará apenas os modelos que têm no máximo 5 instâncias de *signature*. A verificação de asserção da *Alloy* não é como uma prova de teorema, na qual uma vez que se tenha uma entrada e a partir desta entrada origine um resultado "'falso", implica na quebra do teorema. Em *Alloy*, se uma asserção falha, então ela é, necessariamente, falsa. Porém, se ela não falha, ela pode vir a ser falsa em um escopo maior (Jackson 2006).

Segundo (Jackson 2006), uma boa forma de iniciar uma análise de um modelo com um número elevado de *signatures* é começar com um escopo pequeno. Pois "se um asserção é inválida, ela também é inválida em um pequeno escopo". Este configura a hipótese do menor escopo apresentada em (Jackson 2006).

7.2 Modelagem do Mecanismo de Composição de *Profiles*

Nesta seção é apresentado o modelo formal do mecanismo de composição de UML *Profiles* em Alloy e especificado o metamodelo que define os UML *profiles* em *Alloy*, com isso é criado um semântica formal da especificação dos *profiles*;

A fim de realizar a modelagem do mecanismo de composição em *Alloy* foram criados dois módulos:

1. *UMLProfileMetamodel module:* apresenta a modelagem do metamodelo do UML *Profile Metamodel* em *Alloy*.

2. *CompositionRelationship module:* apresenta a modelagem das operações do mecanismo de composição da abordagem proposta em *Alloy*. *CompositionRelationship module* importa o *UMLProfileMetamodel module*.

7.2.1 UML *Profile Metamodel* em *Alloy*

Para modelar o metamodelo dos *profiles* ilustrado na Figura A.2 no Anexo A (página 142) em *Alloy* é necessário também modelar parte do metamodelo da UML o qual os *profiles* fazem uso. Sendo assim, é necessário, de fato, fazer algumas considerações, como segue:

- metaclasses e meta-atributos são considerados como *signature* e *fields*, respectivamente;

- as associações entre as metaclasses são consideradas como *fields*;

- as multiplicidades 1, 0..1, 0..* são representados através das palavras reservadas `one`, `lone`, `some` e `set`, respectivamente.

- as restrições semânticas (*constraints*) definidas no metamodelo dos *profiles*/UML são representadas através de *facts*. Parte das restrições são definidas em OCL, o que facilita a representação destas restrições em *Alloy*, haja vista ambas são baseadas em lógica de predicado.

De acordo com (Jackson 2006), *Alloy* apresenta um baixa performance quando é necessário realizar análise de modelos com muitas *signatures* e *fields*, com 25 *signatures* ou mais, sendo a análise limitada com um escopo de 5–10 *signatures*. Diante disto, modelar o metamodelo dos *profiles* e, conseqüentemente, da UML que apresenta um número maior que 35 metaclasses, as quais são representadas através das *signatures*, torna-se impossível. Assim, é necessário reduzir, o quanto possível, o número de metaclasses que deve ser representada em *Alloy*. O metamodelo dos *profiles* e da UML são apresentados no módulo *UMLProfileMetamodel*.

Por outro lado, se for desconsiderado um número elevado de metaclasses isto levará a uma análise que não condiz com a realidade. Se for considerado, por exemplo, um determinado número de metaclasses, porém não for considerado as metaclasses que estas estabelece uma "meta-associação" (sendo representada como meta-atributos, como anteriormente mencionado) torna a análise não consistente. O objetivo de desconsiderar algumas metaclasses é tornar possível a realização da análise automática.

Após análises do metamodelo dos *profiles*/UML (OMG 2007*c*) e das considerações apresentadas em (Jackson 2002, Zito & Dingel 2006), são listadas algumas considerações, como segue:

- **metaclasses semelhantes:** as metaclasses *DataType*, *PrimitiveType* e *Enumeration* são semelhantes a metaclasse *Class*, haja vista todas possuem nomes, atributos, operações e superclasses. Dessa forma, será considerada apenas a *Class* e é assumido que os resultados são aplicados a todas as outras (OMG 2007*c*, Zito 2006).

- **metaclasses não necessárias:** para realizar a análise e diante das restrições para execução desta, determinadas metaclasses precisam ser desconsideradas. Tais como: *PackageImport*, *PackageMerge*, *ElementImport*, *Comment* e *Constraint* não são utilizadas durante o mecanismo de composição. Com isso, estas metaclasses são inseridas no *resulting profile* sem alterações.

- **herança múltipla:** o metamodelo da UML possibilita o uso de herança múltipla. Porém, *Alloy* não permite a implementação de herança múltipla (Jackson 2006, Jackson 2007), ou seja, não permite uma *signature* realizar herança múltipla. Logo, a fim de evitar problemas na modelagem, a herança múltipla não será implementada. A metaclasses *Property* e *Parameter* ambas herdam de *MultiplicityElement* e *TypedElement* uma saída para implementar esta herança múltipla em *Alloy* é agrupar o conteúdo de *MultiplicityElement* e de *TypedElement* em uma única metaclasse, *TypedMultiplicityElement*, com o objetivo de que esta os representem.

- **herança:** os princípios que foram levados em consideração para projetar o metamodelo da UML (como modularidade, flexibilidade, reuso, *layering* e entre outros), conduziram à formação de um hierarquia, no relacionamento de herança, com um alto grau de profundidade. Por exemplo, a metaclasse *Class* encontra-se abaixo do nível 5 dentro da hierarquia do relacionamento de herança. Desse modo, para solucionar este problema será considerada a filha com os meta-atributos (*fields*) herdados, sendo mantido apenas as superclasses abstratas *TypedElement, Element, PackageElement* e *Element* que são usadas para tornar possível o polimorfismo.

- **atributos e associações derivadas:** a UML possui várias associações e atributos derivados como, por exemplo, os atributos *isOrdered, isUnique, lower* e *upper*

todos da metaclasse *Operation*, a qual define as operações no metamodelo da UML (ver a Figura A.6 no Anexo A página 144). Estes atributos e associações não serão inseridos nos modelos representados em *Alloy*. Apenas o atributo derivado *isDerived* e a associação derivada *extension* da metaclasse *Extesion* definida no metamodelo dos *Profiles* serão preservados.

- **recursão:** para realizar a composição de um *profile*, o qual possui outro *profile* definido no seu interior, é necessário fazer o uso da recursão. Porém, para modelar isto em *Alloy* não é possível, haja vista a mesma não oferece o mecanismo de recursão. Logo, não será levado em consideração a composição de modelos de forma recursiva.

- **tipos primitivos:** tanto o metamodelo da UML quanto a abordagem de composição de modelos proposta fazem uso de tipos primitivos como, por exemplo, *String*, *Integer*, *Double*, e entre outros. Logo, para modelar os mesmos é necessário fazer uso de recursos que torne possível modelá-los. *Alloy* não apresenta todos os tipos primitivos, desta forma é necessário fazer uso de alguns módulos que acompanham a linguagem. Isto é caracterizado por `open util/boolean`, que torna possível o uso do tipo primitivo *boolean* e de algumas operações lógicas como "or" e "and". Com isso, é possível fazer atribuições de valores booleanos *true* e *false*, juntamente com a capacidade de realizar operações lógicas. Para modelar *Strings* é adicionado a *signature* `String` sem nenhum *fields*, representado por `sig String{}`. O uso de *String* no contexto do mecanismo de composição é utilizado para realizar, principalmente, a comparação entre os modelos, a fim de encontrar modelos equivalentes. Haja vista, a abordagem proposta tem seu mecanismo de comparação baseado na assinatura dos modelos. Por exemplo, para comparar duas *signature* é realizada a comparação dos nomes. Caso sejam iguais, implica que elas são equivalentes.

Os UML *profiles* são definidos de acordo com o princípio da metamodelagem, assim como a UML. Sendo assim, um metamodelo é usado para especificar sua sintaxe e sua semântica. O objetivo deste metamodelo é definir como os *profiles* devem ser instanciados. Logo, realizar a composição destes *profiles* deve considerar a especificação deste metamodelo.

Com isso, é apresentado, na Figura 7.2, uma extensão do metamodelo do *UML profiles* simplificada baseada na extensão da UML definida no Capítulo 6 e nas restrições definidas anteriormente. Desse modo, é apresentada a modelagem de `Stereotype` e `Enumeration` em *Alloy*. A modelagem completa encontra-se no Anexo C.

Modelagem de Stereotype em Alloy

O Código 7.2 representa a modelagem da metaclasse *Stereotype* em *Alloy*. Isto é representado através da *signature Stereotype*, como segue:

- `Stereotype` estende `Class`. Logo, tudo que é definido em *Class* é herdado pelo *Stereotype* (linha 1).
- Um `Stereotype` tem um relacionamento com `Image`. Isto é representado através de uma *field* (linha 2)
- Todo `Stereotype` pode apenas generalizar ou especializar outro `Stereotype`. Esta restrição definida em (OMG 2007*c*, pág. 191) é especificada na (linha 4).
- Um `Stereotype` pode ter mais de uma *Image*. Estas imagens são iguais se, e somente se, os mesmo tiverem as propriedades *content, location* e *format* iguais (linha 5–7)
- Todo `Stereotype` deve estar definido em um *Profile* (linha 8).
- Todo *Stereotype* não deve ter conflito de nome (OMG 2007*c*, pág. 191) (linha 9–10).

Código 7.2: Modelagem de *Stereotype* em *Alloy*

```
1  sig Stereotype extends Class {
2    icon: set Image
3  }{
4    all a: Stereotype | this.superClass in Stereotype
5    all i1, i2:icon | (i1.@content = i2.@content &&
6    i1.location = i2.location &&
7    i1.format = i2.format) => i1 = i2
8    one p:Profile | this in p.ownedStereotypes
9    all a: Element, b: Stereotype |
10   (b.@name = a.@name) => b = a
11 }
```

Modelagem de Enumeration em Alloy

Para representar uma *Enumeration* em *Alloy*, é necessário criar uma *signature* que estenda *CompositeElement*. O Código 7.3 representa esta modelagem e a descrição da mesma é apresentada a seguir:

- `Enumeration` extende `CompositeElement`. Logo, tudo que é definido em *CompositeElement* é herdado pelo `Enumeration` (linha 1).

- O relaciona de `Enumeration` com `EnumerationLiteral` é representado como um *field*. A cardinalidade (0..*) é representada em *Alloy* com `set` (linha 2).
- Todo `EnumerationLiteral` com mesmo nome, são considerados iguais (linha 4).
- Todo `Enumeration` deve estar definida em um *Profile* (linha 5).

Código 7.3: Modelagem da *Enumeration* em *Alloy*

```
1 sig Enumeration extends CompositeElement{
2   ownedLiterals: set EnumerationLiteral
3 }{
4   all a, b:ownedLiterals | a.@name = b.@name => a = b
5   one p: Profile | this in p.ownedMembers
6 }
```

7.2.2 Modelagem do Mecanismo de Composição em *Alloy*

O mecanismo de composição faz uso das especificações definidas no metamodelo dos UML *profiles*. Logo, para modelar o mecanismo de composição em Alloy, é necessário fazer uso da modelagem deste em *Alloy*. Sendo assim, o módulo `CompositionRelationship` que apresenta a modelagem do mecanismo de composição importa o módulo `UMLProfile-Metamodel`. A modelagem é baseada na definição do guia de composição. Porém, não é possível modelar as atividades realizadas pelo *model transformation operator*.

Sendo assim, tem-se o objetivo de: (i) verificar os modelos de entradas; (ii) verificar a correspondência entre os modelos; (iii) verificar se a composição é possível. O relacionamento de composição foi modelado como um *predicate* com 5 parâmetros, representado por `CompositionRelationship` e ilustrado no Código 7.4. É apresentado uma descrição a seguir:

- O *predicate* retorna *true* se, e somente se, o parâmetro `resulting` for o resultado da composição do conteúdo dos parâmetros `receiving` e `merged` (linha 1–4).
- Tem 3 `CompositeElement`, uma *match strategy*, representado por `matchStrategy`, e uma *composition strategy*, representado por `compositionStrategy` (linha 1-4).
- O *merge operator* recebe os parâmetros de entrada, representado pelo *predicate* `merge-Operator`, e retorna `true` se, e somente se, for possível compor o conteúdo do `receiving` e `merged`, produzindo `resulting`, de acordo com a *match strategy* e a `composition strategy` especificada. Fica a cargo do *predicate merge operator* a verificação da composição (linha 5–6).

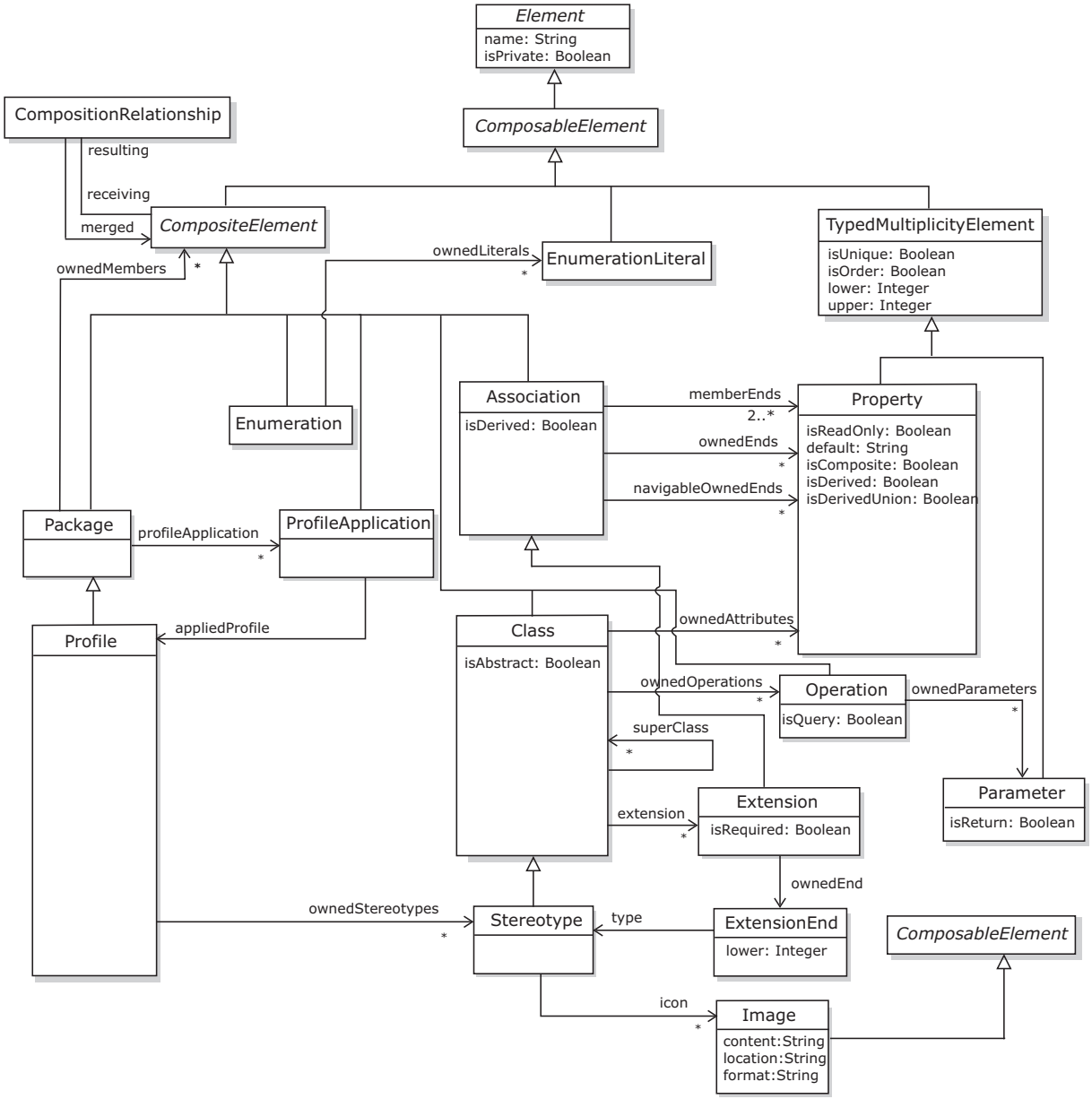

Figura 7.2: Metamodelo utilizado na modelagem em *Alloy*

Código 7.4: Representação do *predicate* `CompositionRelationship`

```
1 pred compositionRelationship(receiving: CompositeElement,
2   merged: CompositeElement, resulting: CompositeElement,
3   matchStrategy: MatchStrategy,
4   compositionStrategy: CompositionStrategy ){
5   mergeOperator[receiving, merged, resulting,
6         matchStrategy, compositionStrategy]
7 }
```

Uma vez que estes *predicates* tenham sido definidos, utiliza-se o *Alloy Analyzer* para verificar se é possível encontrar uma instância do metamodelo especificado na Figura 7.2, que satisfaça o que é especificado nestes *predicates*. O que esta verificação significa? Significa que se o *predicate compositionRelationship* é verificado e seja considerado válido, isto implica que a composição é possível de ser realizada, validando a abordagem. Caso contrário, é mostrado um contra-exemplo mostrando os valores atribuídos as instâncias das *signatures* para o qual o *predicate* é considerado inválido. Um outro exemplo simples de invalidação de *predicate* é quando existem restrições que são conflitantes, quando uma instância da *signature Stereotype* é criada tanto as suas restrições quanto as da *signature Class* devem ser respeitadas e não devem ser contraditórias. Logo, ter uma instância de um *signature Stereotype* implica que todas as restrições são respeitadas, tanto as suas quanto as das suas superclasses.

7.3 Análise do Modelo Formal

Alloy suporta dois tipos de análises automática: (i) *simulation*, usada para demonstrar a consistência de um dado predicado; e (ii) *checking*, é utilizada para validar afirmações através da tentativa de encontrar um contra-exemplo, ou seja, dado uma afirmação o *Alloy Analyzer* busca um contra-exemplo que invalide a afirmação, como mencionado anteriormente. Para realizar a análise é necessário definir o escopo da análise, este escopo delimita o tamanho da instância do modelo que está sendo explorada para validar as afirmações. Quando é encontrado um contra-exemplo a afirmação é necessariamente inválida, por outro lado, quando nenhum contra-exemplo é encontra pode-se afirma que a afirmação é válida dentro do escopo definido.

Sendo assim, se o relacionamento de composição é visto como uma operação tendo algumas entradas e produzindo uma saída, torna-se interessante e pertinente realizar um verificação se este relacionamento é válido para algumas propriedades algébricas . Esta análise visa conhecer as propriedades do relacionamento e verificar algumas propriedades que relacionamento de composição necessariamente precisa ter. Além disso, é possível verificar a semântica do relacionamento.

Em (Brunet et al. 2006), é apresentado um conjunto de propriedades algébricas as quais os operadores de composição deve satisfazer algumas delas. Sendo assim, foi escolhido quatro propriedades com o objetivo de avaliar os operadores que representam o relacionamento de composição, tais como: *idempotency*, *commutativity*, *associativity* e *uniqueness*. Para verificar estas propriedades são criadas assertivas que verificam se estas propriedades são válidas. A seguir é apresentado um descrição destas propriedades:

Idempotency: esta propriedade verifica se a composição de um modelo com ele mesmo, produzirá o mesmo como saída, dado uma estratégia de comparação e de composição. Se S é um conjunto com uma operação binária f definida neste conjunto, então f é *idempotency* se, para todo s definido em S, $f(s,s) = s$. Por exemplo, as operações de união e intersecção de conjuntos podem ser consideradas ambas *idempotency*. De uma maneira similar, a operação de composição pode ser vista como uma operação binária que tem dois modelos de entradas a fim de produzir um modelo de saída, isto pode ser expressado como $merge(m_a,m_a) = m_a$. Em nossa abordagem esta propriedade é expressada em lógica de predicado como:

∀ receiving, merged, resulting: Profile,
∀ match: MatchStrategy,
∀ strategy: CompositionStategy
merge(receiving, merged, resulting, match, strategy) =
merge(receiving, merged, resulting, match, strategy)

Uniqueness: esta propriedade verifica se dado dois modelos de entrada, um estratégia de comparação e um estratégia de composição é produzido um único e possível modelo de saída. Esta propriedade é extremamente importante, pois caso o mecanismo de composição não a atenda, isto significa que existe ambiguidade e a ausência de algum regra de composição. Aplicando esta propriedade é possível verificar se há algum problema no modelo formal. Se algum problema é encontrado, isto é refletido com um contra-exemplo. Em nossa abordagem esta propriedade é expressada em lógica de predicado como:

∀ receiving, merged, resultingA, resultingB: Profile,
∀ match: MatchStrategy,
∀ strategy: CompositionStategy
merge(receiving, merged, resultingA, match, strategy) ∧
merge(receiving, merged, resultingA, match, strategy) →
match(resultingA, resultingB, match)

Commutativity: este propriedade é largamente utilizada na matemática com o objetivo de representar a habilidade de mudança da ordem dos operandos sem alterar o

resultado. O mecanismo de composição respeita esta propriedade se, e somente se, a composição de um modelo A e um modelo B seja igual a composição de B com A. Para uma função binária $f:D\times D \rightarrow K$ é comutativa se, e somente se, $f(x,y) = f(y,x)$ para todo $x, y \in D$. Esta propriedade é expressada em lógica de predicado como:

∀ receiving, merged, resulting: Profile,
∀ match: MatchStrategy,
∀ strategy: CompositionStategy
merge(receiving, merged, resulting, match, strategy) =
merge(merged, receiving, resulting, match, strategy)

Associativity: como no relacionamento de composição não é especificado a ordem da composição, esta propriedade é importante para verificar se ordem interfere na composição. Um operação binária f em um conjunto D é associativa se, e somente se, ela satisfaz a regra de associatividade: $f(f(x,y),z) = f(x,f(y,z))$ para todo $x, y, z \in D$. Esta propriedade é expressada em lógica de predicado como:

∀ receiving, merged, mergedA, resulting,
resultingA, resultingB : Profile,
∀ match: MatchStrategy, ∀ strategy: CompositionStategy
merge(merge(receiving, merged, resultingA, match, strategy), mergedA, resulting, match, strategy) =
merge(receiving, merge(merged, mergedA, resultingB, match, strategy), resulting, match, strategy)

O relacionamento de composição é representado através de um *predicate* sendo ilustrado no Código 7.5 e descrito como segue:

- tem cinco parâmetros de entrada: três *CompositeElement* (*receiving, merged* e *resulting*), uma estratégia de comparação (*matchStrategy*) e uma estratégia de composição (*compositionStrategy*) (linhas 1–4).
- uma vez definido os parâmetros de entrada do relacionamento de composição, estes parâmetro são passados para o *predicate mergeOperator*, o qual tem a função de verificar se a composição é possível de ser realizada, ou seja, verifica o papel do operador de composição da abordagem proposta (linhas 5–6).

Código 7.5: *Predicate* que representa o relacionamento de composição

```
1 pred compositionRelationship(receiving:
2 CompositeElement, merged: CompositeElement, resulting:
```

```
3 CompositeElement , matchStrategy: MatchStrategy ,
4 compositionStrategy:CompositionStrategy ){
5       mergeOperator[receiving , merged , resulting ,
6       matchStrategy , compositionStrategy]
7 }
```

O próximo passo é representar as propriedades algébricas em *Alloy*, para uma dada estratégia de comparação e uma dada estratégia de composição, aplicada ao operador de composição citado anteriormente. Sendo assim, o Código 7.6 representa a formalização da propriedade *Idempotency* através de uma assertiva em *Alloy* para a estratégia de comparação *default* e a estratégia de composição *override*. Sendo assim, tem-se:

- Definição do nome da assertiva (linha 1).
- Declaração dos *profiles*, da estratégia de comparação e da estratégia de composição (linhas 2–3).
- A composição de $\boldsymbol{a}$ com ele mesmo, deve produzir o mesmo como saída (onde $\boldsymbol{c}$ representa o resultado da composição e necessariamente $\boldsymbol{c}$=$\boldsymbol{a}$) (linhas 4–6).

Código 7.6: Assertiva que verifica a propriedade *idempotency* em *Alloy*

```
1    all a, c : Profile | all match: DefaultMatchStrategy |
2    all strategy: OverrideStrategy |
3    mergeOperator[a, a, c, match, strategy] =>
4    mergeOperator[a, a, c, match, strategy]
5 }
```

O Código 7.7 representa a formalização da propriedade *Uniqueness* através de uma assertiva em *Alloy* para a estratégia de comparação *default* e a estratégia de composição *override*. Um descrição é realizada a seguir:

- Definição do nome da assertiva (linha 1).
- Declaração dos *profiles*, da estratégia de comparação e da estratégia de composição (linhas 2–4).
- A composição de *receiving* e *merged* deve resultar em *resultingA* e *resultingB*, os quais devem se iguais seguindo a *default match rule* (linhas 5–9).

Código 7.7: Assertiva que verifica a propriedade *uniqueness* em *Alloy*

```
6   all receiving , merged , resultingA , resultingB: Profile |
7   all match: DefaultMatchStrategy |
```

```
8     all strategy: OverrideStrategy |
9     mergeOperator[receiving, merged, resultingA,
10      match, strategy] &&
11    mergeOperator[receiving, merged, resultingB,
12      match, strategy] =>
13    matchOperator[resultingA, resultingB, match]
14  }
```

Sendo assim, o Código 7.8 representa a formalização da propriedade *Commutativity* através de uma assertiva em *Alloy* para a estratégia de comparação *complete* e a estratégia de composição *override*. Um descrição é realizada a seguir:

- Definição do nome da assertiva (linha 1).
- Declaração dos *profiles*, da estratégia de comparação e da estratégia de composição (linhas 2–4).
- A composição de *receiving* e *merged* deve resultar em *resultingA* e *resultingB*, os quais devem se iguais seguindo a *default match rule* (linhas 4–6).

Código 7.8: Assertiva representada em *Alloy* que verifica a propriedade *commutativity*

```
1  assert mergeOperatorCommutative {
2     all a, b, c : Profile |
3     all match: CompleteMatchStrategy |
4     all strategy: OverrideStrategy |
5     mergeOperator[a, b, c, match, strategy] =>
6     mergeOperator[b, a, c, match, strategy]
7  }
```

O Código 7.9 representa a formalização da propriedade *Associativity* através de uma assertiva em *Alloy* para a estratégia de comparação *complete* e a estratégia de composição *merge*. Um descrição é realizada a seguir:

- Definição do nome da assertiva (linha 1).
- Declaração dos *profiles*, da estratégia de comparação e da estratégia de composição (linhas 2–5).
- A verificação da propriedade consiste na representação da expressão *merge((merge(m_a,m_b,matchStrategy,mergeStategy),m_c,matchStrategy,mergeStategy) = merge(m_a,(merge(m_b,m_c,matchStrategy,mergeStategy),matchStrategy,mergeStategy)* em *Alloy* (linhas 6–13).

Código 7.9: Assertiva representada em *Alloy* que verifica a propriedade *associativity*

```
1  assert mergeOperatorIsAssociative {
2    all receiving, mergedA, mergedB, resulting,
3      intermediateA, intermediateB: Profile |
4    all match: CompleteMatchStrategy |
5    all strategy: MergeStrategy |
6    mergeOperator[receiving, mergedA, intermediateA,
7      match, strategy] &&
8    mergeOperator[intermediateA, mergedB, resulting,
9      match, strategy] &&
10   mergeOperator[receiving, mergedB, intermediateB,
11     match, strategy] =>
12   mergeOperator[intermediate2, merged1, resulting,
13     match, strategy]
14 }
```

A execução das assertivas modeladas em *Alloy* foi feita em uma máquina com *Windows Home*, 1 GB de memória e processador centrino de 1,6 GHz. Para isto, foi utilizado a versão 4.0 do *Alloy Analyzer*. Além desta assertivas descritas anteriormente, foi analisado a composição de *profiles* para as estratégias de *override* e *merge*, para as três estratégias de comparação: *default*, *partial* e *complete*. Os resultados da análise da composição seguindo a estratégia *override* e *merge* são mostrados na Figura 7.3 e Figura 7.4, respectivamente. Nas figuras, é apresentado o tempo de execução, o escopo e se a propriedade é valida, ou não. Com isso, pode se concluir que relacionamento de composição satisfaz as quatro propriedades levando em consideração o escopo 2.

		Match Strategy								
		Default			Partial			Complete		
		Verify	Scope	Time	Verify	Scope	Time	Verify	Scope	Time
Properties	Idempotency	Sim	2	38m27s	Sim	2	39m40s	Sim	2	39m02s
	Uniqueness	Sim	2	15m11s	Sim	2	16m15s	Sim	2	16m29s
	Commutativity	Sim	2	42m16s	Sim	2	47m11s	Sim	2	45m17s
	Associativity	Sim	2	51m43s	Sim	2	51m40s	Sim	2	53m47s

Figura 7.3: Resultado da análise da composição de *profiles* seguindo a estratégia de composição *override*

		Match Strategy								
		Default			Partial			Complete		
		Verify	Scope	Time	Verify	Scope	Time	Verify	Scope	Time
Properties	Idempotency	Sim	2	27m03s	Sim	2	30m18s	Sim	2	31m07s
	Uniqueness	Sim	2	12m05s	Sim	2	13m07s	Sim	2	14m41s
	Commutativity	Sim	2	22m15s	Sim	2	24m01s	Sim	2	21m09s
	Associativity	Sim	2	45m27s	Sim	2	40m05s	Sim	2	39m17s

Figura 7.4: Resultado da análise da composição de *profiles* seguindo a estratégia de composição *merge*

Capítulo 8

Implementação do Mecanismo de Composição

Com o objetivo de automatizar e colocar em prática o mecanismo de composição apresentado no Capítulo 4, foi desenvolvido uma ferramenta de composição de modelos chamada MoCoTo. A implementação do mecanismo de composição torna possível a automatização na obtenção do *resulting profile*.

Este capítulo possui as seguintes seções:

- *Tecnologias Utilizadas*: nesta seção são descritas as tecnologias utilizadas para o desenvolvimento da ferramenta.
- *MoCoTo – Visão Geral:* nesta seção são apresentadas as características, um cenário de aplicação da ferramenta e uma descrição da ferramenta.
- *Arquitetura da MoCoTo:* nesta seção é apresentado a arquitetura da ferramenta juntamente com a descrição da mesma.
- *Funcionalidades:* nesta seção são apresentadas as funcionalidades da ferramenta.
- *Interfaces da Ferramenta:* com o objetivo de ilustrar como o modelador interage com a ferramenta são apresentadas algumas *interfaces*.

8.1 Tecnologias Utilizadas

Esta seção tem o objetivo de descrever as tecnologias que foram utilizadas no desenvolvimento do *protótipo*:

8.1.1 *Eclipse Modeling Framework*

O EMF (Eclipse Project 2007) trata-se de um *framework* de modelagem do *Eclipse* usado freqüentemente para gerar código a partir de modelos. Dado os modelos de entradas, o

EMF gera um completa API para criar, rodar, salvar e manipular instâncias dos modelos. O modelos de entrada do EMF são feitos baseados no ECore. O ECore consiste do metamodelo utilizado para definir a sintaxe e a semântica do modelos manipulados pelo EMF. Resumidamente, o ECore é semelhante ao MOF, isto implica que o ECore é capaz de construir elementos como, por exemplo, classes, interfaces e associações (Budinsky et al. 2003). Porém, não é capaz de criar máquina de estados, diagrama de seqüencia e entre outros. Basicamente, os modelos são formados por EClasses (elementos equivalentes às classes definidas no MOF) as quais podem ter EAttributes (elementos equivalentes as propriedades definidas no MOF) e é possível estabelecer relacionamento entre as EClasses (estes relacionamentos são equivalentes à associações definidas no MOF).

A API gerada pelo EMF inclui uma interface e implementação para toda classe no modelo. Para elemento do modelo é fornecido métodos *get* e *set* para todos os atributos e associações, e um fábrica para criar instâncias das classes. Além disso, toda classe gerada fornecida pelo EMF implementa a interface *EObject* (Budinsky et al. 2003).

8.2 MoCoTo – Visão Geral

As mais populares ferramentas de modelagem (tais como, Jude, Visual Paradigma, Poseidon e entre outras) permitem a construção de modelos, porém não são capazes de realizar a composição dos mesmos. Compor modelos manualmente consiste em uma tarefa árdua. Pois, modelos tipicamente possuem dezenas ou até centenas de elementos, os quais possuem vários tipos de relacionamento, sendo um deles, possivelmente, o relacionamento de composição. Além disto, o tamanho, a complexidade e a falta de conhecimento dos modelos por parte do modeladores em equipes, possivelmente, distribuídas aumenta, sem dúvida, a dificuldade.

Sendo assim, a utilização de uma ferramenta que automatize parcial ou completa a composição traz significantes melhorias. Neste contexto, a ferramenta MoCoTo foi desenvolvida com o objetivo de automatizar a composição e auxiliar os modeladores no processo de composição de *profiles*. Desse modo, é ilustrado um cenário de aplicação da MoCoTo na Figura 8.1.

De acordo com (Bézivin et al. 2006), uma ferramenta de composição de modelos deve ser capaz de:

- validar e verificar a composição de modelos (por exemplo, checar tipos do modelos de entrada);
- uma *máquina virtual* para, de fato, realizar a composição;
- analisar as falhas ou as inconsistências que surgem durante o processo de composição.

- apresentar um mecanismo de serialização para carregar e salvar os modelos.

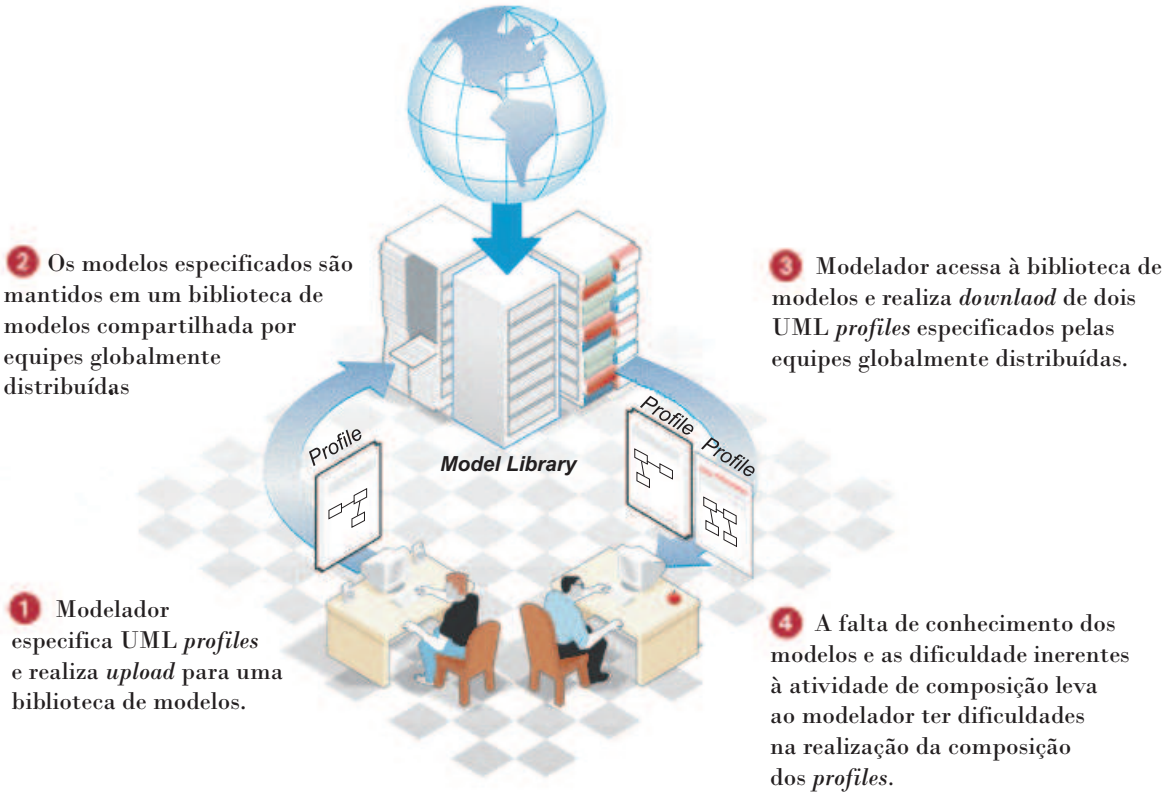

Figura 8.1: Cenário de aplicação da MoCoTo

Tendo isto em mente, foi elaborada uma ferramenta que auxilia a composição de *profiles*. De um modo geral, a ferramenta permite:

- verificar se os modelos de entrada são modelos válidos para mecanismo de composição.
- compor *profiles* de acordo com o mecanismo de composição definido no Capítulo 4;
- verificar problemas que surgem durante a composição
- importar, carregar e salvar *profiles*;

Sob o ponto de vista do modelador, a ferramenta MoCoTo interage com o usuário para conduzi-lo a um correta configuração da composição. Para isto, um assistente de composição captura os dados necessários que definem como a composição dos *profiles* deve ser realizada. Este assistência tem como base a verificação se os parâmetros de entrada da composição são válidos ou não.

A MoCoTo tem cinco parâmetros que configuram a composição:

- **Match Strategy:** parâmetro obrigatório que define como a comparação dos *profiles* deve ser realizada.
- **Composition Strategy:** parâmetro obrigatório que define como a composição dos *profiles* deve ser realizada.
- **Threshold (t):** parâmetro obrigatório que especifica qual o menor valor do grau de similaridade entre dos elementos de *profiles* para os mesmos serem considerados equivalentes. Caso não seja especificado é utilizado o valor padrão $t = 0.5$.
- **Receiving Profile:** parâmetro obrigatório que define o primeiro operando da composição. Este parâmetro trata de *profile* no formato de arquivo .uml, tipo de arquivo fornecido pelo EMF e UML2.
- **Merged Profile:** parâmetro obrigatório que define o segundo operando da composição. Este parâmetro trata de *profile* no formato de arquivo .uml, tipo de arquivo fornecido pelo EMF e UML2.

A Figura 8.2 mostra a interação da ferramenta com o usuário. Um vez definidos os parâmetros de entrada a ferramenta produz o *resulting profile* como resultado da composição.

Figura 8.2: Parâmetros de entrada necessários à composição.

8.3 Arquitetura

Para atender a funcionalidade especificada através do mecanismo de composição, bem como as características de uma ferramenta de composição de modelos citadas anteriormente, a ferramenta MoCoTo foi dividida em 5 módulos, os quais são apresentados na Figura 8.3. Estes cinco módulos formam a arquitetura da ferramenta e interagem entre se com o objetivo, basicamente, de propiciar: a verificação dos modelos de entradas; a determinação da correspondência entre os modelos de entrada; a composição dos *profiles*; e a persistência dos modelos. Este módulos são descritos como segue:

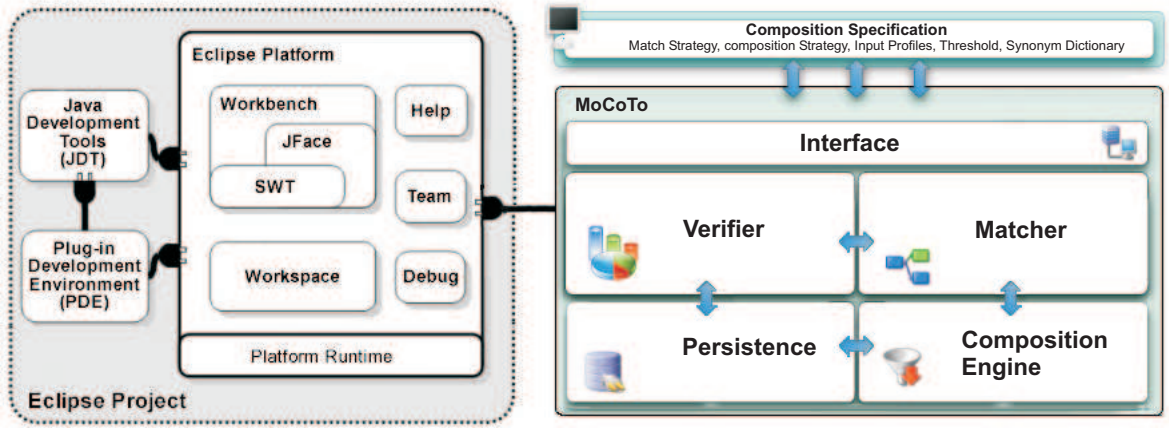

Figura 8.3: Arquitetura da ferramenta MoCoTo

Interface: a interação entre a ferramenta e o seu usuário é realizada através deste módulo. A interface da MoCoTo é baseada no conceito de *Views*, *Perspective* e *Wizard* encontrados na plataforma *Eclipse*. Este módulo será detalhada na Seção 8.5.

Verifier: tem como finalidade à validação dos parâmetros de entrada, bem como a geração da configuração de composição dos *profiles*. Sendo assim, como ilustrado na Figura 8.4, o *verifier* gera uma mensagem de erro caso não seja possível criar a configuração.

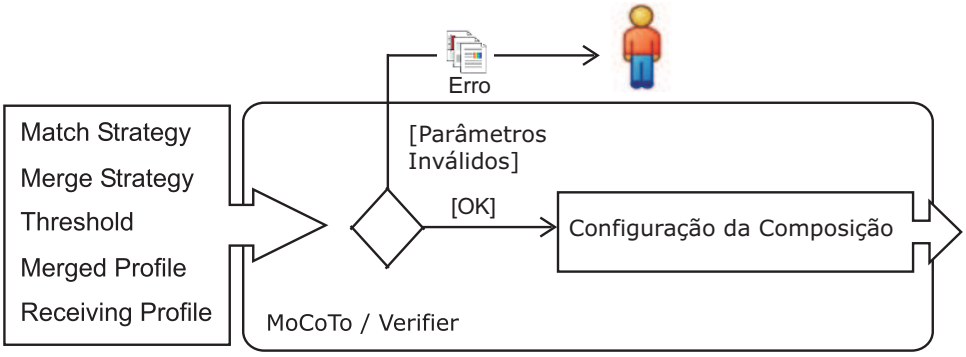

Figura 8.4: Ilustração da execução do *verifier*

Matcher: de posse da configuração da composição devidamente validado, o *Matcher* realiza a comparação dos modelos de entrada de acordo com a *match strategy* especificada. Utiliza o algoritmo NGram, a definição das assinaturas, as *match rules* gerando uma tabela de similaridade. Baseado no *threshold*, são definidos os modelos que são equivalentes e a descrição de equivalência. A Figura 8.5 ilustra o *matcher*.

Composition Engine: de posse dos elementos dos *profiles* que são equivalentes, da descrição de equivalência entre eles e da especificação da estratégia de composição,

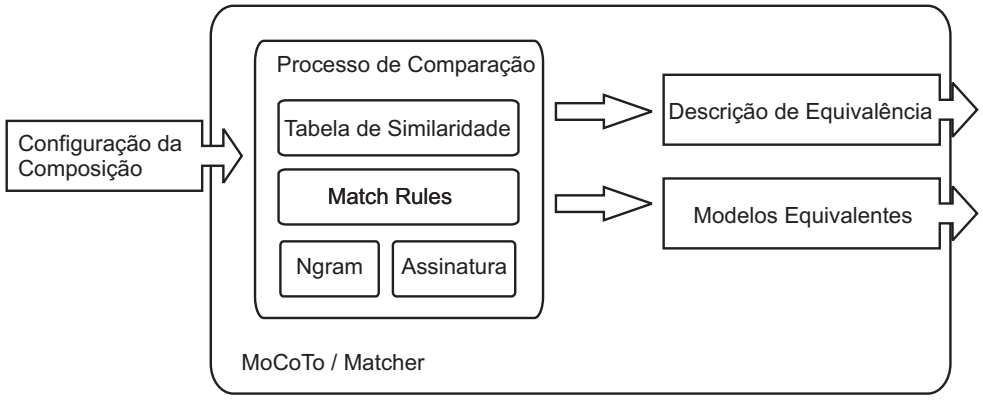

Figura 8.5: Ilustração da execução do *matcher*

o *Composition Engine* realiza, de fato, a composição dos *profiles*. Para isto, são utilizadas determinadas regras de composição de acordo com a estratégia de composição definida. Por exemplo, se a estratégia de composição definida for a *override strategy*, então as regras de composição utilizadas pelo *Composition Engine* serão as *override merge rules*. A Figura 8.6 representa a *CompositionEngine*.

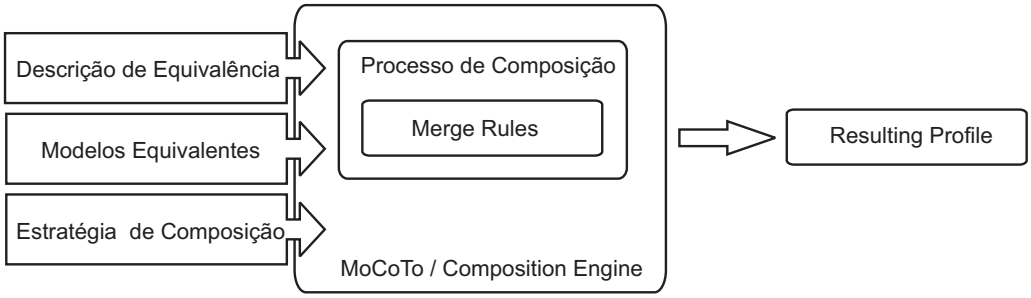

Figura 8.6: Ilustração da execução do *composition engine*

Persistence: o módulo de persistência é responsável pela gerência de todo o mecanismo de armazenamento em disco dos *profiles*. O formato de arquivo manipulado é do tipo `.uml`.

8.4 Funcionalidades

Nesta seção são apresentadas as funcionalidades básicas da ferramenta. Para especificá-las, foi tomado como partida à visão do modelador. Sendo assim, a ferramenta MoCoTo pode ser dividida em duas funcionalidades: (*i*) configurar composição; (*ii*) compor *profiles*. A Figura 8.7 ilustra estas funcionalidades.

Antes de realizar a composição, é necessário definir como a mesma deve ser executada. Sendo assim, o modelador precisa definir a configuração da composição. Para

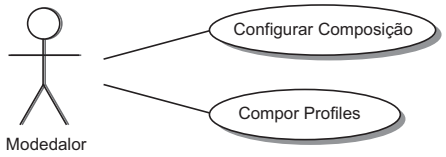

Figura 8.7: Atividades realizadas pelo modelador

construir a configuração da composição tem-se que: (*i*) especificar os parâmetros (*receiving profile*, *merged profile*, estratégia de composição, estratégia de comparação, *threshold*); (*ii*) carregar os modelos; (*iii*) validar a configuração; (*iv*) enviar mensagem ao usuário. Na Figura 8.8 é ilustrado o caso de uso e na Figura 8.9 o diagrama de atividades desta funcionalidade.

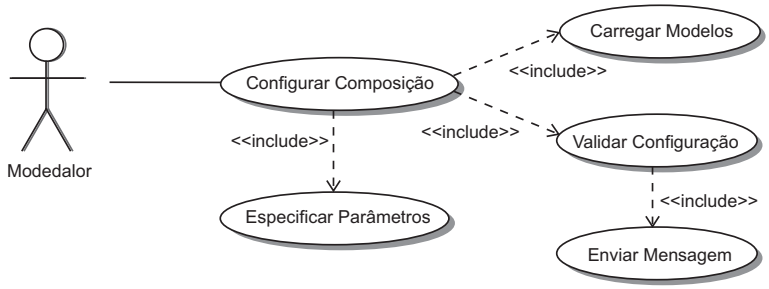

Figura 8.8: Diagrama de caso de uso relacionado à funcionalidade "Configurar Composição"

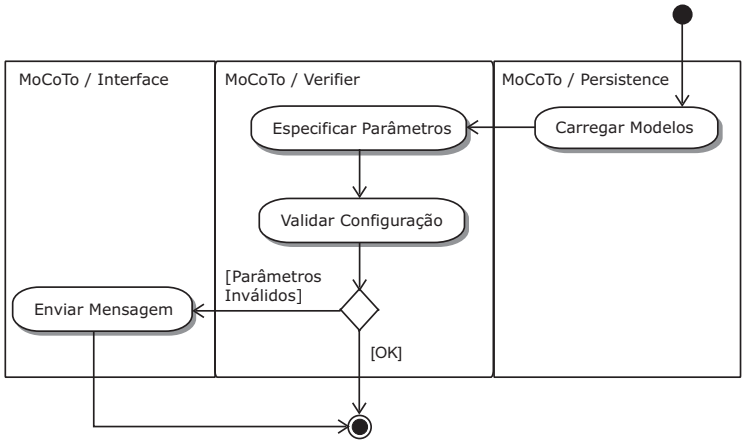

Figura 8.9: Diagrama de atividade referente à funcionalidade "Configurar Composição"

Uma vez definido a configuração da composição, é possível compor os *profiles* de entrada. Para compor é necessário: (i) comparar os *profiles*; (ii) executar efetivamente a composição; e (iii) persistir o modelo de saída. Isto é ilustrado na Figura 8.10. Para executar a composição os módulos da arquitetura envolvidos são: *matcher*, *composition machine* e *persistence*. As atividades que são executadas durante a composição são distribuída por estes módulo de acordo com a Figura 8.11

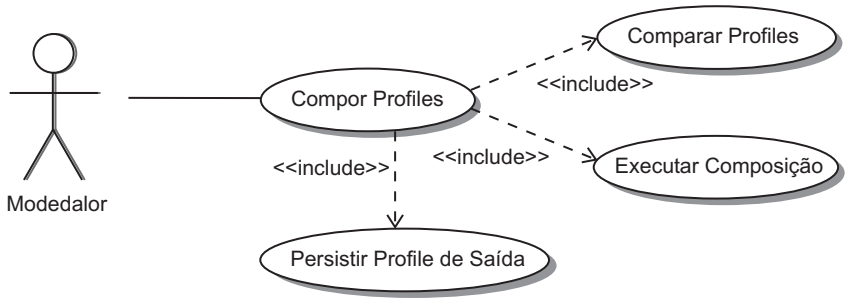

Figura 8.10: Diagrama de caso de uso referente à funcionalidade "Compor *Profiles*"

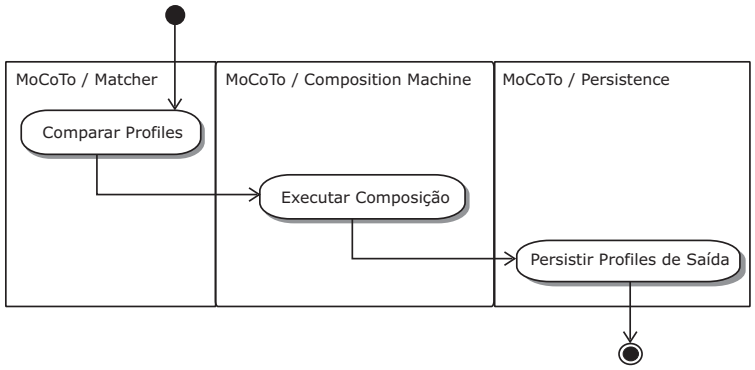

Figura 8.11: Diagrama de atividade referente à funcionalidade "Compor *Profiles*"

8.5 Interfaces da MoCoTo

Uma ponto importante do processo de composição é a interação do modelador. Esta interação tem como objetivo configurar a composição e principalmente executar a composição. Uma das formas para aumentar a compreensão desta interação (Modelador ⇔ Máquina) é através da utilização de interface (GUI) amigável. Sendo assim, como a ferramenta MoCoTo trata-se de um *plug-in* da plataforma *Eclipse*, foram utilizados os conceitos de *interfaces* apresentados nesta plataforma, tais como: *views*, *menus* e *perspective*. A *interface* definida é integrada ao módulo *interface* da arquitetura da MoCoTo.

A comunicação (Modelador ⇔ MoCoTo) é de responsabilidade deste módulo. É através da *interface* que todas as informações necessária ao processo de composição são

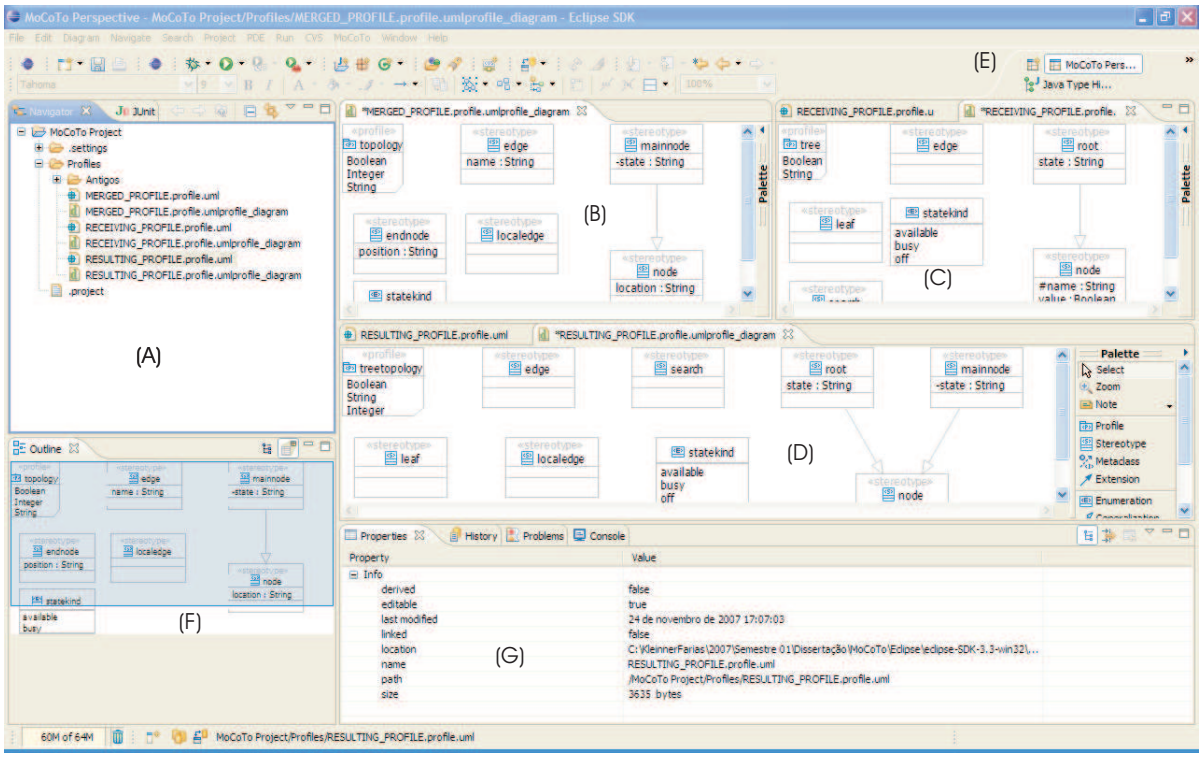

Figura 8.12: Tela principal da ferramenta.

passadas. Quando a MoCoTo é inicializada, é apresentado uma janela principal a partir da qual o modelador tem uma visão geral do ambiente. Como ilustrado na Figura 8.12, esta visão geral disponibiliza :

- em (A) é disponibilizado uma lista de *profiles* que podem participar da composição. Esta *view* permite o modelador visualizar os *profiles* disponíveis e o resultado da composição. Os *profiles* que podem ser importados com o objetivo de participar do processo de composição deve também ser visualizados nesta *view*.

- em (B), (C) e (D) é mostrado a visualização do *merge profile*, *receiving profile* e *resulting profile*, respectivamente. Ao lado direto destas *views*, é disponibilizado um paleta de componentes que permite alterar o modelo.

- em (E) é apresentada perspectiva da ferramenta (*MoCoTo Perspective*). Onde toda a interação do modelador com a ferramenta deve ser realizada através dela. Ou seja, uma vez escolhida esta perspectiva as funcionalidades da ferramenta são disponibilizadas.

- em (G) é mostrado a *view properties* que permite ver as propriedades dos modelos que estão sendo manipulados e que participam do processo de composição. Com esta *view*, o modelador pode ver detalhes dos modelos que participam da composição, visualizando detalhes do que foi modificado.

- em (F) é mostrado a *view outline*. Esta *view* permite ter uma visão geral dos modelos.

Uma vez apresentada a janela principal, é possível realizar a composição dos *profiles*. Sendo assim, a ação de compor é representa através de um submenu ilustrado da Figura 8.14. Um vez que *MergeAction* seja selecionada, é mostrado a *interface* de configuração de composição. Nesta *interface*, é possível definir os parâmetros: *match strategy*, *merge strategy*, *threshold*, *receiving profile* e *merged profile*. Finalmente, a composição é executada através do botão *merge*.

É apresentado uma *overview* da ferramenta com o objetivo de aumentar sua usabilidade por parte do usuário. Isto é ilustrado na Figura 8.13. Além disto, é fornecido um link para o site www.umlprofile.org, onde será hospedado informações sobre a ferramenta.

Figura 8.13: Tela de *overview* da ferramenta.

Com o objetivo de permitir o modelador fazer uso de *profiles* previamente definidos, é possível importá-los e especificá-los para a composição. Para isto, foi criado um extensão do mecanismo de *import* oferecido na plataforma *Eclipse* o qual é mostrado na Figura 8.15. Com isso, é possível importa arquivos do tipo *.uml*. Para uma eventual evolução da ferramenta, o mecanismo de *import* consiste em um importante recurso.

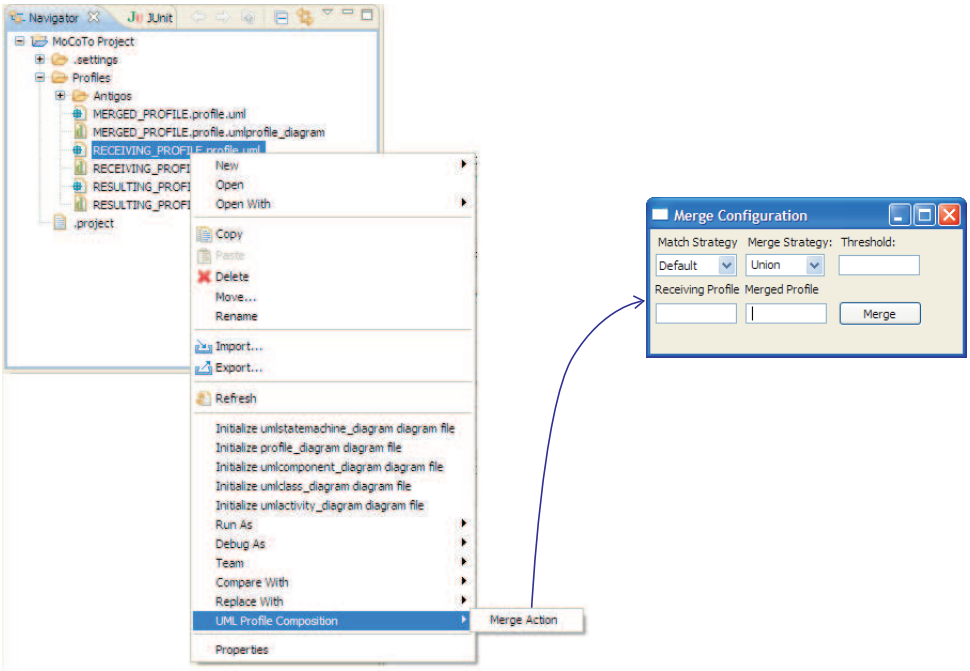

Figura 8.14: Tela de configuração da composição.

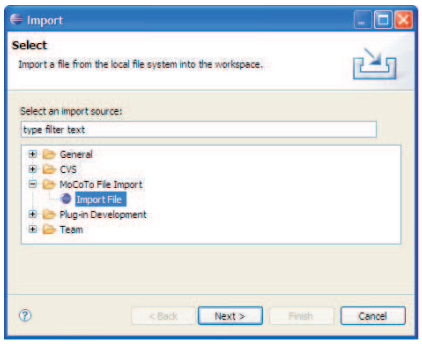

Figura 8.15: Importando *profile*

Capítulo 9

Conclusão

Com a MDD surgiram novos desafios dentre eles a necessidade de compor modelos e construir linguagem de modelagem especifica de domínio a fim de modelar domínios específicos. Enquanto a UML fornece um mecanismo de extensão que permite a criação de DSML através de *profiles*, o seu mecanismo de composição não é capaz de compor os mesmos. Neste contexto, foi proposto um mecanismo de composição de composição de *profiles*.

O mecanismo proposto é fundamentado em quatro operadores de composição, estratégia de composição, estratégia de comparação, regras de composição, regras de comparação, regras de transformação e em um guia de composição de modelos que auxilia a composição. Além disso, a construção de uma extensão da UML permitiu criar uma representação do mecanismo de composição. Com o objetivo de avaliar a proposta, foi elaboração uma ferramenta de composição e uma verificação formal da abordagem usando Alloy.

Após um levantamento das referências relevantes relacionadas ao tema da pesquisa, verificou-se a ausência de investigações na solução dos problemas citados nesta dissertação. Sendo assim, a solução destes problemas configura as principais contribuições da pesquisa, sendo algumas destas contribuições publicadas em eventos nacionais e internacionais. Por fim, espera-se com este trabalho ter gerado conhecimento para a evolução da área de composição de modelos.

9.1 Limitações do Estudo

A seguir são apresentadas algumas limitações deste trabalho:

1. Durante a avaliação da proposta não foi realizada verificações da composição de *profiles* de grande porte usados na industria.

2. A abordagem não oferece uma solução para a composição de *constraints* associadas

aos elementos dos *profiles* considerando o seu valor semântico.

3. As considerações feitas referente ao valor semântico dos modelos no momento da comparação dos *profiles* é limitada. Esta limitação pode dificultar e comprometer o processo de composição.

9.2 Trabalhos Futuros

Nesta dissertação foram respondidas algumas questões importantes relacionadas à composições de modelos, especificamente sobre a composição de *profiles*. Embora estas respostas sejam boas contribuições, a abordagem precisa ser melhorada e ter maiores avaliações com o objetivo de ser consolidada. Apesar de avaliações iniciais terem demonstrado a aplicabilidade e corretude da abordagem, maiores investigações necessitam ser desenvolvidas quanto a aplicabilidade em *profiles* de grande porte.

Sendo assim, são listados alguns trabalhos a serem realizados futuramente, como segue:

Operadores de Composição

- **Aplicação dos operadores em contextos diferentes**
  Utilização dos operações de composição para compor diagramas da UML, por exemplo, diagrama de atividades e diagrama de estado. Observações iniciais tem apontado ser factível, porém sendo necessário algumas adaptações para atender a natureza dos diagramas.

- **Definição de um linguagem de composição de modelos**
  Embora tenham sido definidos operadores de composição, os quais são responsáveis por realizar a composição, não há um base formal para garantir a correta aplicação destes operadores. Diante disto, a definição de uma linguagem de composição de modelos permitirá e disponibilizará um correta definição e especificação da composição.

Ferramenta de Composição de Modelos

- **Análise de desempenho da MoCoTo**
  Embora tenham sido realizados alguns testes através da composição de alguns *profiles*, é necessário realizar algumas avaliações com *profiles* de grande porte utilizados na industria a fim de verificar o comportamento da ferramenta. Dependendo da quantidade de elementos que façam parte de um determinado *profile*, pode implicar alguma queda de desempenho da ferramenta. Desse modo, um avaliação neste sentido é importante e representa um relevante e provável futuro investimento. O resultado desta avaliação permitirá prover melhorias na ferramenta.

- **Suporte a um linguagem de composição de modelos**
  Um dos trabalhos futuros é a definição de uma linguagem de composição para especificar a composição de modelos. Um vez cumprido esta etapa, torna-se necessário um ambiente que dê suporte a utilização desta linguagem. Sendo assim, a evolução da ferramenta neste sentido é extremamente pertinente.

Guia de composição de modelos

- **Aplicabilidade do guia de composição**
  A aplicação do guia de composição definido neste trabalho em diferentes contextos (por exemplo, na composição de diagrama de seqüência) a fim de realizar composição de outros tipos de modelos é indispensável para avaliação de sua aplicabilidade, consolidação e evolução.

Análise formal em Alloy

- **Verificar novas propriedades**
  A análise formal do mecanismo permitiu a validação do seu funcionamento e a verificação de algumas propriedades. Porém, novas avaliação devem ser feitas considerando escopos maiores e analisando outras propriedades. Uma vez que já existem resultados neste sentido, a consolidação desta nova atividade tende a se tornar realidade em um futuro próximo.

Referências Bibliográficas

Atkinson, C. & Kuhne, T. (2003), 'Model-Driven Development: a Metamodeling Foundation', *IEEE Software* **20**(5), 36–41.

Batini, C., Lenzerini, M. & Navathe, S. (1986*a*), 'A Comparative Analysis of Methodologies for Database Schema Integration', *ACM Computing Surveys* **18**(4), 323–364.

Batini, C., Lenzerini, M. & Navathe, S. B. (1986*b*), 'A Comparative Analysis of Metodologies for Database Schema Integration', *ACM Computing Surveys* **18**(4), 323–364.

Baudry, B., Fleury, F., France, R. & Reddy, R. (2005), Exploring the Relationship between Model Composition and Model Transformation, *in* '7th International Workshop on Aspect-Oriented Modeling, co-located with (MODELS'05)', IEEE Computer Society, Montego Bay, Jamaica, pp. 4–12.

Bernstein, P. & Melnik, S. (2007), 'Model Management 2.0: Manipulating Richer Mappings', *Special Interest Group on Management of Data* pp. 1–12.

Bourdeau, R. & Cheng, B. (1995), 'A Formal Semantics for Object Model Diagrams', *IEEE Transactions on Software Engineering* **11**(21), 799–821.

Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N. & Sabetzadeh, M. (2006), A Manifesto for Model Merging, *in* 'International Workshop on Global Integrated Model Management (GaMMa'06)', ACM Press, Shanghai, China, pp. 5–12.

Budinsky, F., Ellersick, R., Grose, T. J., Merks, E. & Steinberg, D. (2003), *Eclipse Modeling Framework*, The Eclipse Series, Addison-Wesley Professional, USA.

Bézivin, J. (2001), From Object Composition to Model Transformation with the MDA, *in* 'IEEE - TOOLS 2001', IEEE Computer Society, Santa Barbara, USA, pp. 58–70.

Bézivin, J., Bouzitouna, S., Del Fabro, M. D., Gervais, M. P., Jouault, F., Kolovos, D., Kurtev, I. & Paige, R. (2006), A Canonical Scheme for Model Composition, *in* 'Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications- (ECMDA-FA'06)', pp. 27–35.

Clarke, S. (2001), Composition of Object-Oriented Software Design Models, PhD thesis, School of Computer Applications, Dublin City University, Dublin, Irland.

C.Parent & Spaccapietra, S. (1998), 'Issues and Approaches of Database Integration', *Communications of The ACM* **41**(5), 166–178.

Dennis, G. (2003), TSAFE: Building a Trusted Computing Base for Air Traffic Control Software, Master's thesis, MIT, Cambridge, MA.

Dennis, G., Seater, R., Rayside, R. & Jackson, D. (2004), Automating Commutativity Analysis at the Design Level, *in* 'Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis', ACM Press, USA, pp. 165–174.

Eclipse Project (2007), 'Eclipse Modeling Framework'. http://www.eclipse.org/emf, Capturado em 18 de novembro de 2007.

Estublier, J. & Ionita, A. D. (2005), Extending UML for Model Composition, *in* 'Australian Software Engineering Conference (ASWEC'05)', IEEE Computer Society, USA, pp. 31–38.

Fernández, L. & Moreno, A. (2004), 'An Introduction to UML Profiles', *The European Journal for the Informatics Professional* **5**(2), 6–13.

France, R., Ghosh, S. & Dinh Trong, T. (2006), 'Model Driven Development Using UML 2.0: Promises and Pitfalls', *IEEE Computer Society* **39**(2), 59–66.

France, R. & Rumpe, B. (2007), Model-Driven Development of Complex Software: A Research Roadmap, *in* 'Future of Software Engineering (FOSE'07) co-located with ICSE'07', IEEE Computer Society, Minnesota, EUA, pp. 37–54.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, USA.

Harrison, W. & Ossher, H. (1993), Subject Oriented Programming (a critique of pure objects), *in* 'Object Oriented Programming, Systems, Languages Applications (OOPSLA)', IEEE Computer Society, USA, pp. 80–92.

IBM (2007), 'IBM Rational Software Modeler'. http://www-306.ibm.com/software/, Capturado em 22 julho de 2007.

Jackson, D. (2002), 'Alloy: a Lightweight Object Modelling Notation', *ACM Transactions on Software Engineering and Methodology (TOSEM)* **11**(2), 256–290.

Jackson, D. (2006), *Software Abstractions: Logic, Language and Analysis*, The MIT Press, USA.

Jackson, D. (2007), 'The Alloy Analyzer'. http://alloy.mit.edu/, Capturado em 22 abril de 2007.

Manning, C. & Shütze, H. (1999), *Foundations of Statistical Natural Language Processing*, MIT Press, USA.

Massoni, T., Gheyi, R. & Borba, P. (2004), A UML Class Diagram Analyzer, *in* 'Third Workshop on Critical Systems Development with UML affiliated with UML Conference', ACM Press, USA, pp. 100–114.

Mostefaoui, F. & Vachon, J. (2007), Verification of Aspect-UML Models Using Alloy, *in* 'Proceedings of the 10th International Workshop on Aspect-Oriented Modeling', ACM Press, USA, pp. 41–48.

Muller, P., Fleury, F. & Jézéquel, J. (2005), Weaving Executability into Object Oriented Meta-Languages, *in* 'Proceedings 8th International Conference on UML Modelling Languages and Applications (MODELS/UML'05)', IEEE Computer Society, Montego Bay, Jamaica.

Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S. & Zave, P. (2007), Matching and Merging of Statecharts Specifications, *in* 'ICSE'07', IEEE Computer Society, Minnesota, EUA, pp. 54–64.

Oliveira, K. & Oliveira, T. (2007*a*), A Guidance for Model Composition, *in* 'International Conference on Software Engineering Advances (ICSEA'07)', IEEE Computer Society, France, pp. 27–32.

Oliveira, K. & Oliveira, T. (2007*b*), Composição de UML Profiles, *in* 'Workshp de Tese e Dissertações em Engenharia de Software SBES'07', João Pessoa, PB, pp. 17–23.

OMG (2001), *Common Warehouse Metamodel*, Object Management Group. Internacional Standard ISO/IEC 19501.

OMG (2002), *Meta-Object Facility Version 1.4*, Object Management Group. http://www.omg.org, Último acesso em 12 de agosto de 2007.

OMG (2003), *MDA Guide Version 1.0.1*, Object Management Group. http://www.omg.org/docs/omg/03-06-01.pdf.

OMG (2007*a*), 'Common Object Request Broker Architecture (corba)'. http://www.omg.org/corba/, Capturado em 28 maio de 2007.

OMG (2007*b*), 'Object Management Group (OMG)'. http://www.omg.org/, Capturado em 28 maio de 2007.

OMG (2007*c*), *Unified Modeling Language: Infrastructure version 2.1*, Object Management Group. http://www.omg.org/cgi-bin/apps/doc?formal/07-02-03.pdf, Capturado em 12 de abril de 2007.

Pólya, G. (2004), *How to Solve It: a New Aspect of Mathematical Method*, Princeton University Press, USA.

Reddy, R., France, R., Ghosh, S., Fleurey, F. & Baudry, B. (2005), Model Composition - A Signature Based Approach, *in* 'Aspect Oriented Modeling (AOM) Workshop', IEEE Computer Society, Montego Bay, Jamaica, pp. 12–22.

Reddy, Y., France, R., Straw, G., J. Bieman, N. M., Song, E. & Georg, G. (2006), 'Directives for Composing Aspect-Oriented Design Class Models', **1**(1), 75–105.

Rumbaugh, J., Jacobson, I. & Booch, G. (2005), *The Unified Modeling Language Reference Manual*, Second edn, Object Technology Series, Addison-Wesley, USA.

Sabetzadeh, M., Nejati, S., Easterbrook, S. & Chechik, M. (2006), TReMer: a Tool for Relationship Driven Model Merging, *in* '14th International Symposium on Formal Methods', Vol. 5, IEEE Computer Society, USA, pp. 56–73.

Selic, B. (2003), 'The Pragmatics of Model-Driven Development', *IEEE Software* **20**(5), 19–25.

Simons, A. (2004), 'The Theory of Classification Part 17: Multiple Inheritance and the Resolution of Inheritance Conflicts.', *Journal of Object Technology* **4**(2), 15–26.

Straw, G., Georg, G., Song, E., Ghosh, S., France, R. & Bieman, J. (2004), Model Composition Directives, *in* 'Proceedings 7th International Conference on UML Modelling Languages and Applications', IEEE Computer Society, USA, pp. 84–97.

Stroustrup, B. (2000), *The C++ Programming Language*, Addison-Wesley Professional, USA.

Sun (2007*a*), 'Java Platform Enterprise Edition 5 (Java EE 5)'. Sun Microsystems, http://java.sun.com/javaee/, Capturado em 30 maio de 2007.

Sun (2007*b*), 'Java Platform Micro Edition (Java 5 ME)'. http://java.sun.com/javame/, Capturado em 28 maio de 2007.

Tempero, E. & Biddle, R. (2000), 'Simulating Multiple Inheritance in Java', *Journal of Systems and Software* **55**(1), 87–100.

Triskell (2005), 'The KerMeta Project Home Page'. http://www.kermeta.org/, Capturado em 28 maio de 2007.

Zito, A. (2006), UML's Package Extension Mechanism: Taking a Closer Look at Package Merge, Master's thesis, School of Computing, Quenn's University Kingston, Ontario, Canada.

Zito, A. & Dingel, J. (2006), Modeling UML 2 Package Merge With Alloy, *in* 'Proceedings of the 1st Alloy Workshop (Alloy'06)', IEEE Computer Society, Portland, Oregon, USA.

Apêndice A

Metamodelo da UML

Neste Anexo é apresentado parte do metamodelo da UML que é utilizado na dissertação.

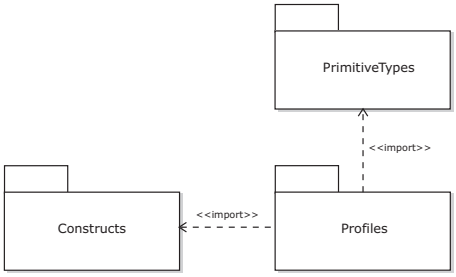

Figura A.1: Pacote *Profile* com dependência em relação a *Constructs* e *PrimitiveTypes*

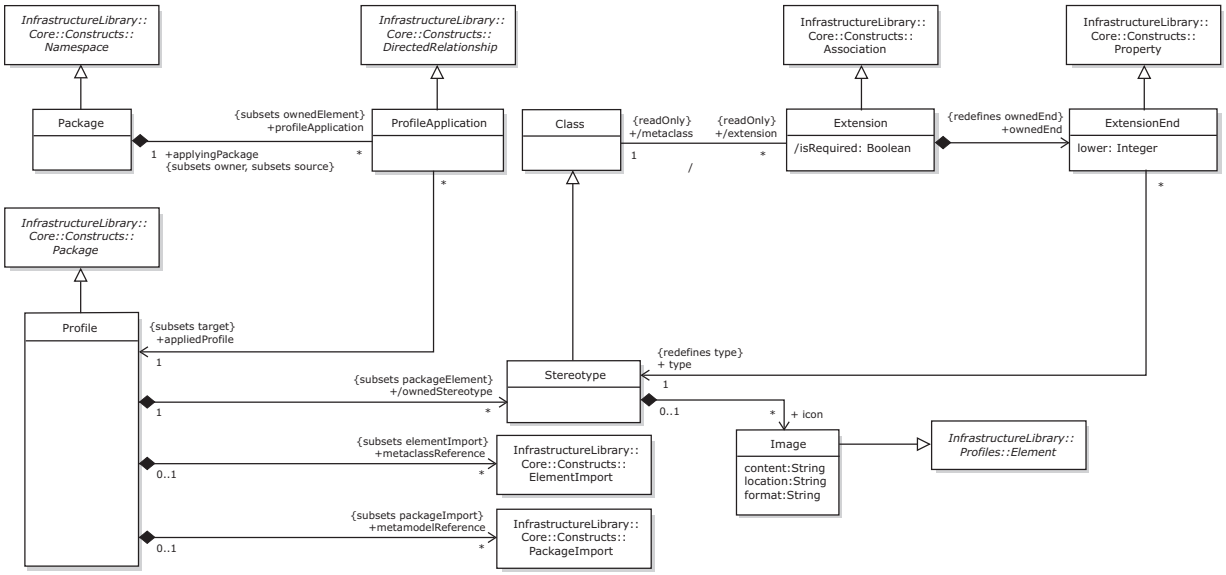

Figura A.2: Classes definidas no pacote *profiles*

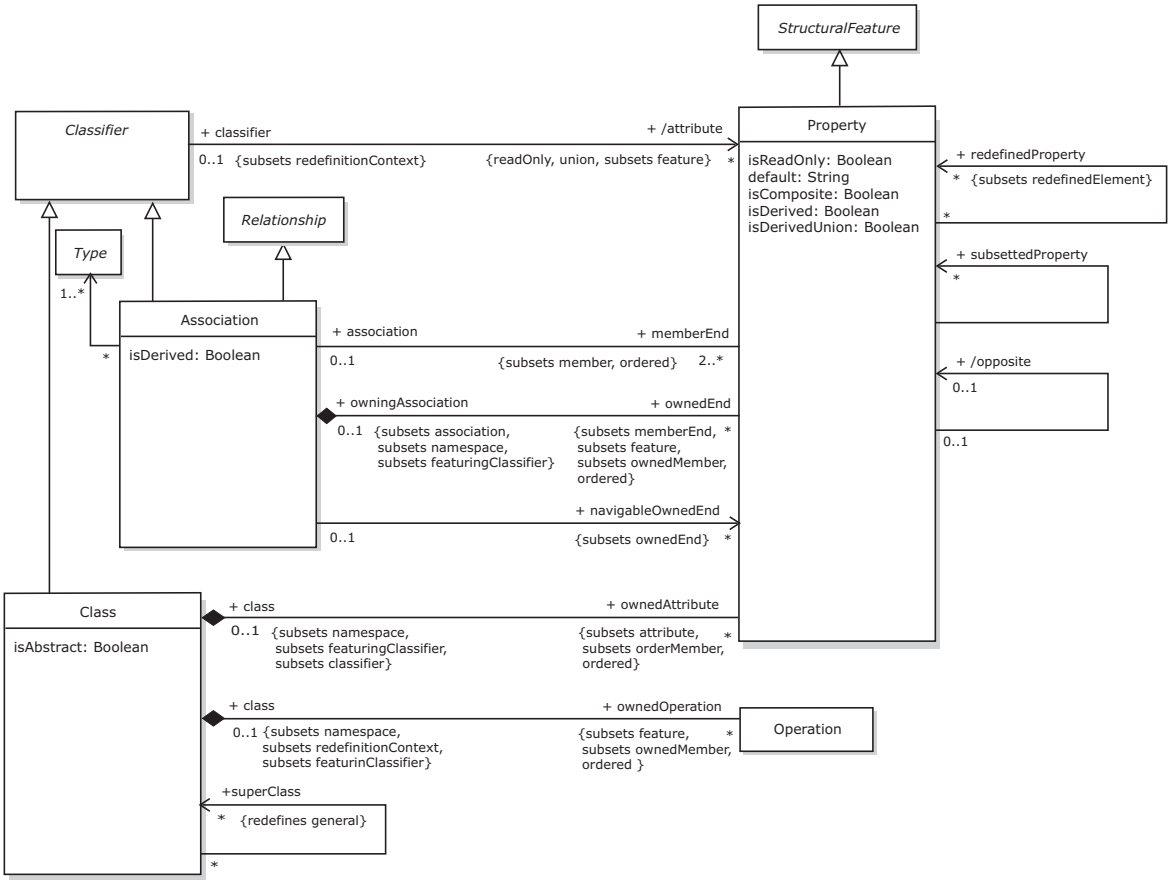

Figura A.3: Diagrama de classe do pacote *Constructs*

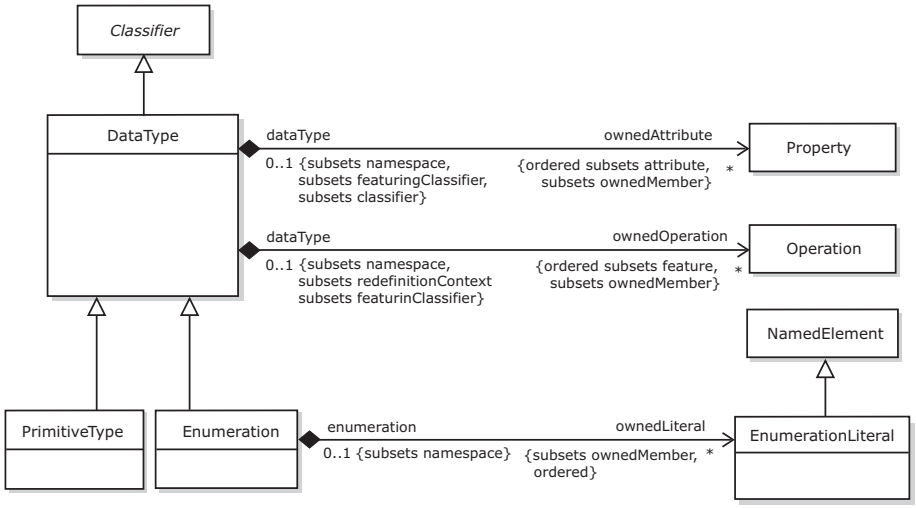

Figura A.4: Classes definidas no diagrama de *DataTypes*

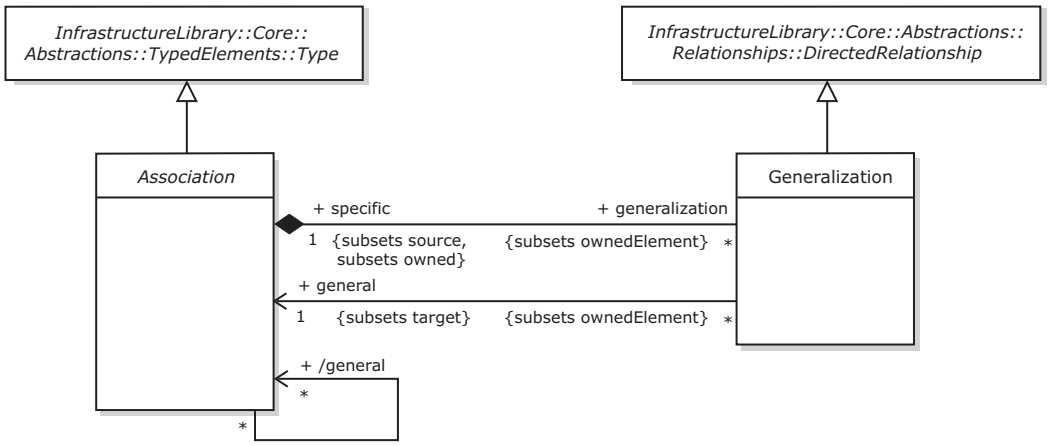

Figura A.5: Diagrama de *generalization* do pacote *constructs*

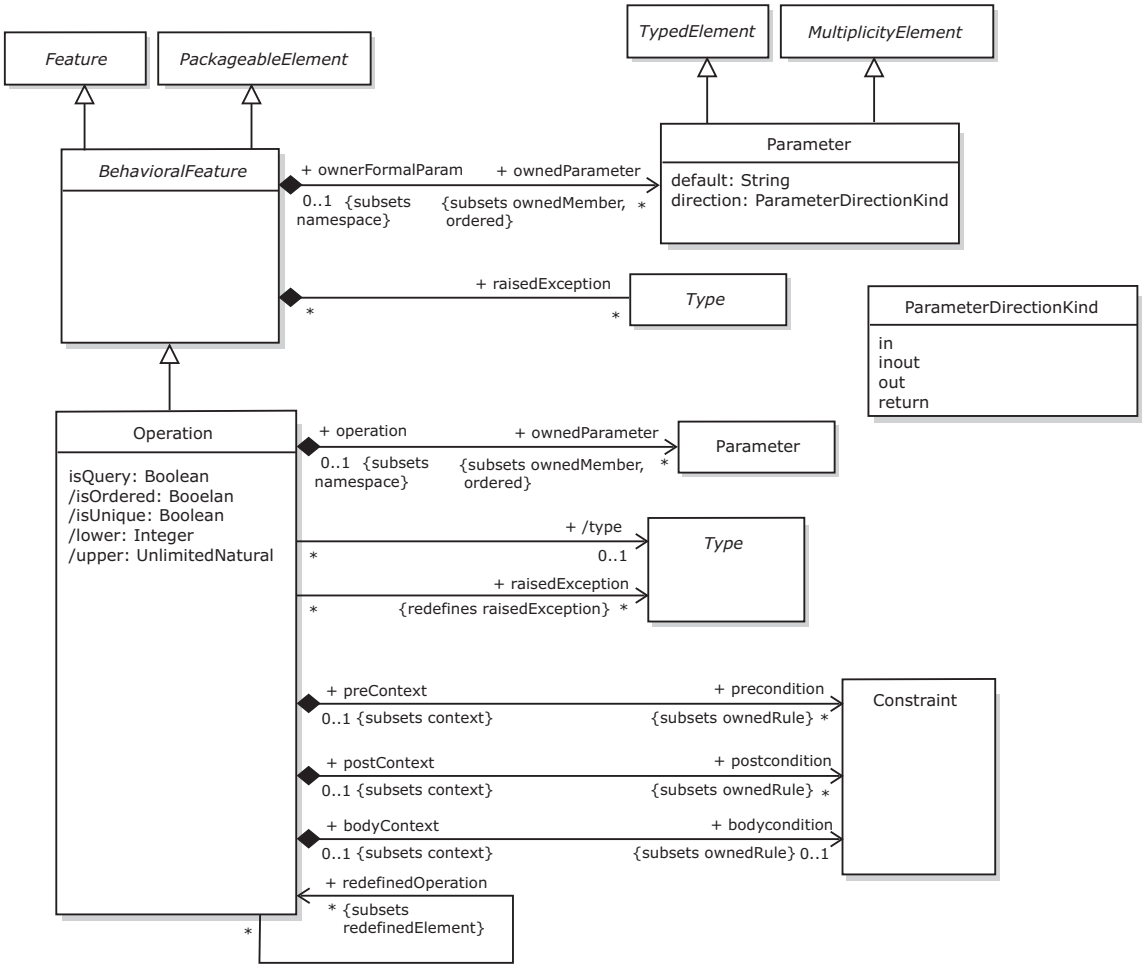

Figura A.6: Diagrama de *operations* do pacote *constructs*

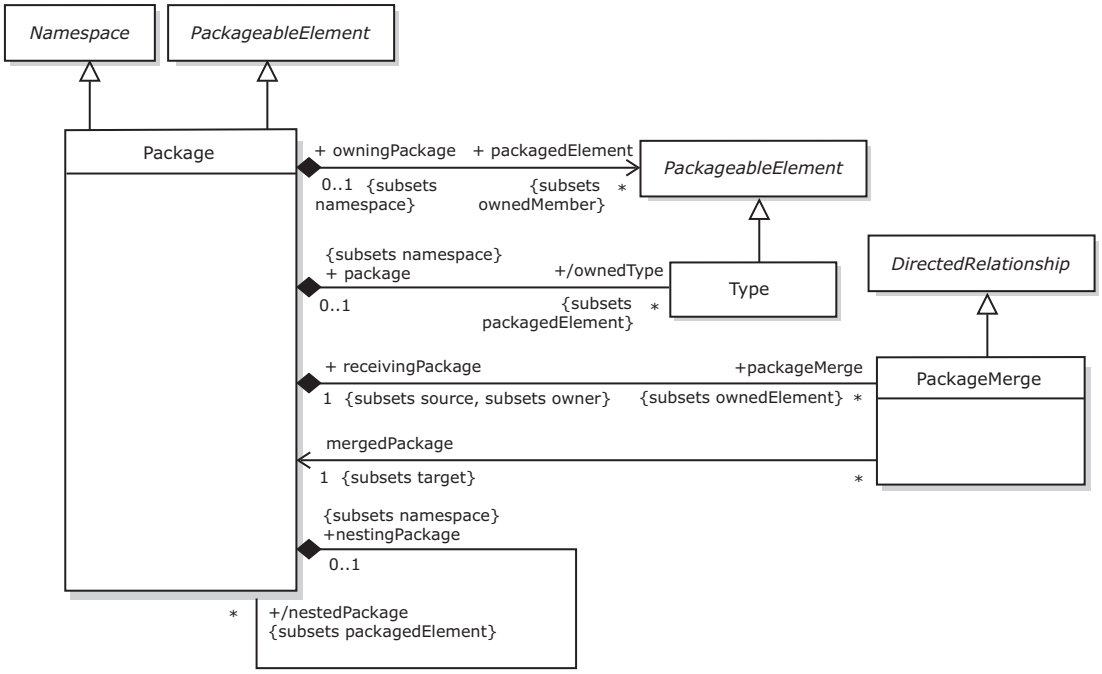

Figura A.7: Diagrama de pacote do pacote *constructs*

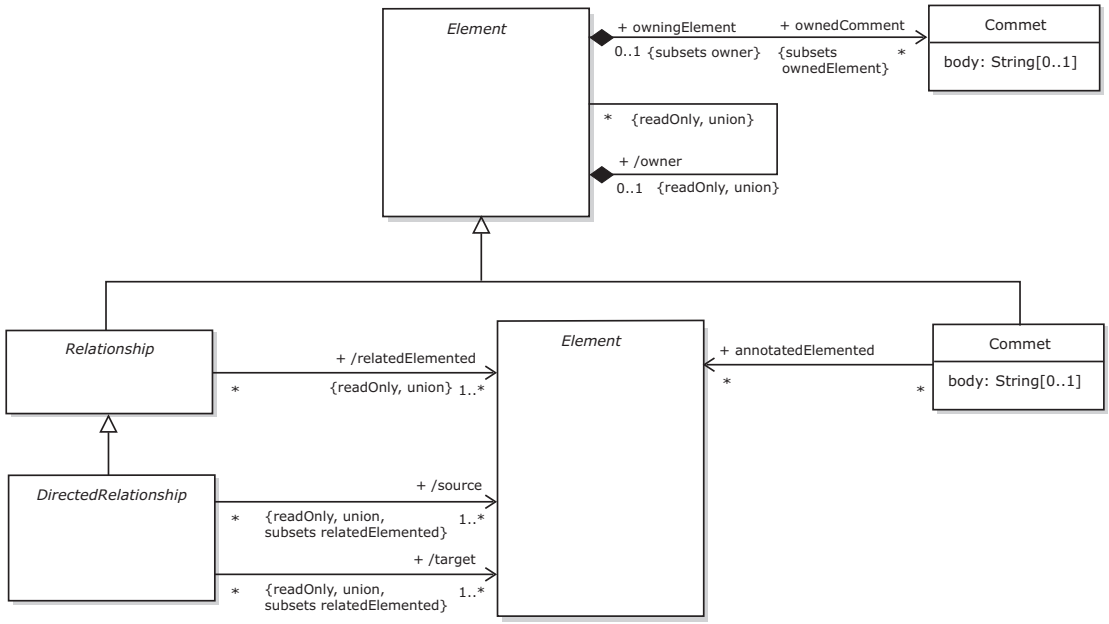

Figura A.8: Diagrama raiz do pacote *constructs*

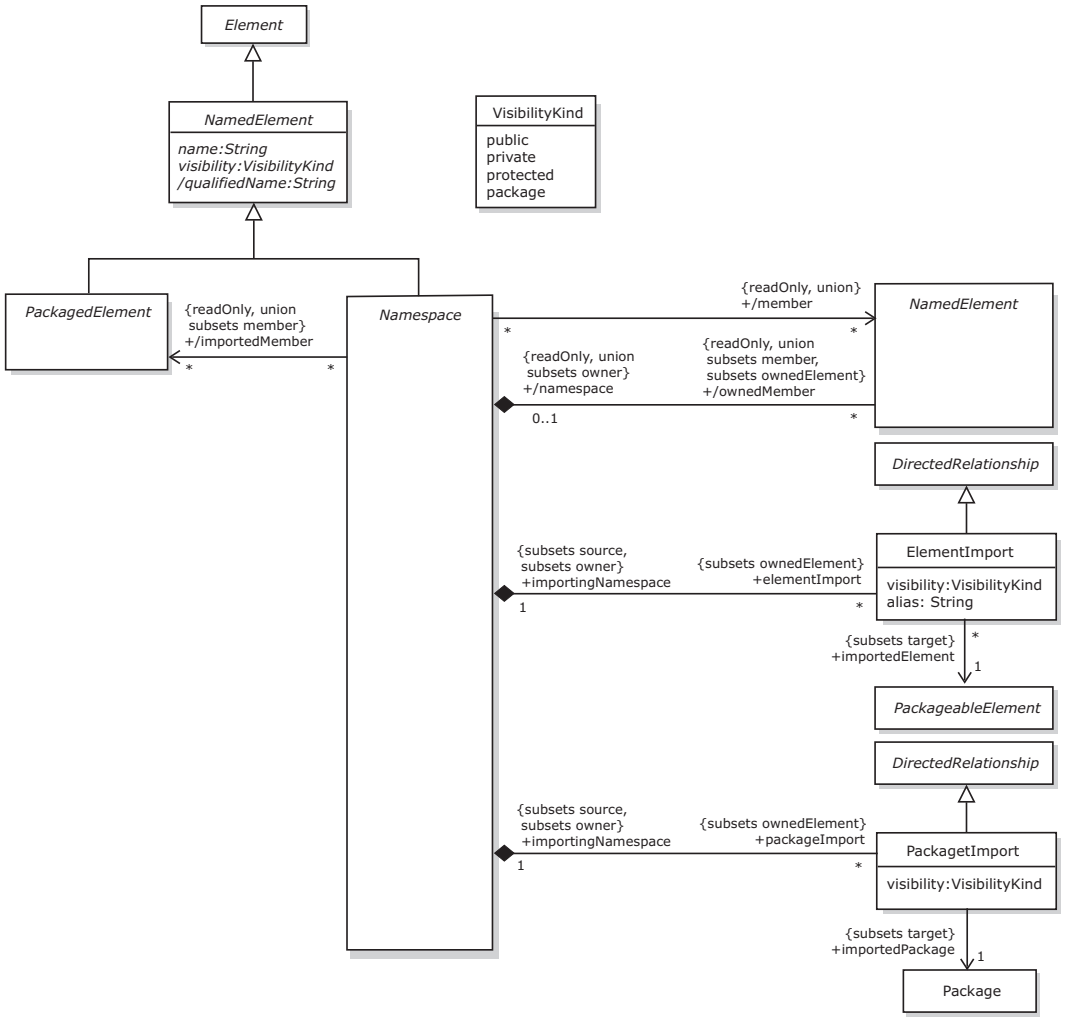

Figura A.9: Diagrama que define *namespace* do pacote *constructs*

Apêndice B

Merge Rules

Este anexo tem o objetivo de descrever as regras de composição utilizadas na abordagem proposta. A definição e a natureza das regras de composição são totalmente ligadas as estruturas dos elementos dos metamodelo da UML. Sendo assim, para realizar a composição são consideradas as propriedades sintáticas (meta-atributos) dos elementos os quais as regras os manipulam a fim de produzir a propriedade de saída.

Estrutura da Descrição das Regras de Composição

[Identificação do Elemento] Merge Rule

1. *propriedade sintática (meta-atributo) 1*: descrição como este meta-atributo 1 é composto.

2. *propriedade sintática (meta-atributo) 2*: descrição como este meta-atributo 2 é composto.

MeR1 Comment Merge Rule

1. *ownesComment: Comment (de Element)*: todos os *ownesComment* são inseridos no elemento de saída. Por exemplo, dado o *Comment* A, com dois *ownesComment*, e *Comment* B, com três *ownesComment*, o *Comment* C que representa a composição de A e de B terá cinco *ownesComment*.

2. *body: String*: deve ser igual.

3. *annotatedElement: Element*: dois comentários para serem equivalentes devem ter necessariamente o meta-atributo *annotatedElement* iguais. Logo, dado dois *Comment* A e B de entrada, produzindo *Comment* C como resultado. Logo, A + B = C, onde C.*annotatedElement* = (A.*annotatedElement* ou B.*annotatedElement*).

MeR2 Class Merge Rule

1. *ownedComment: Comment* (de Element): composto de acordo com a MeR1

2. *name: String* (de NamedElement): para compor duas classes, elas devem ter nomes iguais. Assim, dada duas Classes A e B de entrada, produzindo C como resultado da composição. Logo, A + B = C, onde C.*name* = (A.*name* ou B.*name*).

3. *ownedRule: Constrain* (de Namespace): as *ownedRule* devem ser compostas de acordo com a MeR9.

4. *elementImport: ElementImport* (de Namespace): composto de acordo com a MeR3.

5. *packageImport: PackageImport* (de Namespace): composto de acordo com a MeR4.

6. *isAbstract: Boolean*: dado duas Class A e B de entrada, produzindo C como saída. Logo, C.*isAbstract* = A.*isAbstract* AND B.*isAbstract*.

7. *visibility: VisibilityKind* (de NamedElement): dado duas *Class* A e B de entrada, produzindo C como saída. Logo,
   If (A.visibility = private and B.visibility = private)
   then C.visibility = private
   else C.visibility = public

8. *extension: Extension*: deve ser composto por união. Dado duas classes A (estendida por E)e B (estendida por F), produzindo C (estendida por E e F) como saída.

9. *superClass: Class*: deve ser composto por união. Dado duas classes A (subclasse de E)e B (subclasse de F), produzindo C (subclasse de E e F) como saída.

10. *ownedAttribute: Property*: dado duas *Class* A e B de entrada, produzindo C como saída. Todos os atributos de A e de B devem ser compostos de acordo com a regra MeR5.

11. *ownedOperation: Operation*: composto de acordo com a MeR6.

MeR3 ElementImport Merge Rule

1. *visibility: VisibilityKind* (de NamedElement): dado dois *ElementImport* A e B de entrada, produzindo C como saída. Logo,
   If (A.visibility = private and B.visibility = private)
   then C.visibility = private
   else C.visibility = public

2. *alias: String*: deve ser igual.

3. *importedElement: PackageableElement*: deve ter elementos importados equivalentes.

4. *ownedComment: Comment* (de Element): composto de acordo com MeR1.

MeR4 PackageImport Merge Rule

1. *visibility: VisibilityKind* (de NamedElement): dado dois *ElementImport* A e B de entrada, produzindo C como saída. Logo,
   If (A.visibility = private and B.visibility = private)
   then C.visibility = private
   else C.visibility = public

2. *importedPackage: Package*: deve ser iguais.

3. *ownedComment: Comment* (de Element): composto de acordo com MeR1.

MeR5 Property Merge Rule

1. *name: String* (de NamedElement): para compor duas Propriedades , elas devem ter nomes iguais. Assim, dada duas Propriedades A e B de entrada, produzindo C como resultado da composição. Logo, A + B = C, onde C.*name* = (A.*name* ou B.*name*).

2. *lower: Integer* (de MultiplicityElement): dada duas Propriedades A e B de entrada, produzindo C como resultado da composição. Logo, A.*lower* é igual ao menor valor entre B.*lower* e C.*lower*.

3. *upper: UnlimitedNatural* (de MultiplicityElement): dada duas Propriedades A e B de entrada, produzindo C como resultado da composição. Logo, A.*upper* é igual ao maior valor entre B.*upper* e C.*upper*.

4. *type: Type* (de TypeElement): para compor dois *Type* , deve prevalecer o de mais escopo. Por exemplo, para compor *integer* + *double* = *double*.

5. *isUnique: Boolean* (de MultiplicityElement): dada duas Propriedades A e B de entrada, produzindo C como resultado da composição. Logo, A.*isUnique* = B.*isUnique* OR C.*isUnique*.

6. *isReadOnly: Boolean*: dada duas Propriedades A e B de entrada, produzindo C como resultado da composição. Logo, A.*isReadOnly* = B.*isReadOnly* AND C.*isReadOnly*.

7. *isComposite: Boolean*: dada duas Propriedades A e B de entrada, produzindo C como resultado da composição. Logo,
   If (B.*isComposite* = *true* AND
   C.*isComposite* = *true*)
   then A.*isComposite* = *true*
   else A.*isComposite* = *false*

8. *default: String*: dada duas Propriedades A e B de entrada, produzindo C como resultado da composição. A.*default* e B.*default* devem ser iguais. Logo, C.*default* = B.*default* OR B.*default*.

9. *visibility: VisibilityKind* (de NamedElement): dado dois *Property* A e B de entrada, produzindo C como saída. Logo,
   If (A.visibility = private AND B.visibility = private)
   then C.visibility = private
   else C.visibility = public

10. *isDerivedUnion: Boolean*: dada duas Propriedades A e B de entrada, produzindo C como resultado da composição. Logo, A.*isDerivedUnion* = B.*isDerivedUnion* OR C.*isDerivedUnion*.

11. *ownedComment: Comment* (de Element): composto de acordo com MeR1.

12. *isDerived: Boolean*: dada duas Propriedades A e B de entrada, produzindo C como resultado da composição. Logo, A.*isDerived* = B.*isDerived* OR C.*isDerived*.

MeR6 Operation Merge Rule

1. *name: String* (de NamedElement): para compor duas *Operation*, elas devem ter nomes iguais. Assim, dada duas *Operation* A e B de entrada, produzindo C como resultado da composição. Logo, A + B = C, onde C.*name* = (A.*name* OR B.*name*).

2. *owned: Parameter* (de Parameter): composto de acordo com a regra MeR7.

3. *ownedComment: Comment* (de Element): composto de acordo com MeR1.

4. *bodyCondition: Constraint*: devem ser compostas de acordo com a regra MeR9.

5. *elementImport: ElementImport* (de Namespace): deve ser composto de acordo com a regra MeR3.

6. *raisedException: Type* (de BehaviouralFeature): dada duas *Operation* A e B de entrada, produzindo C como resultado da composição. Logo, C possui todos *raisedException* de A e B.

7. *redefinedOperation: Operation*: dada duas *Operation* A e B de entrada, produzindo C como resultado da composição. Logo, C possui todos *redefinedOperation* de A e B.

8. *preCondition: Constraint*: devem ser compostas de acordo com a regra MeR9.

9. *visibility: VisibilityKind* (de NamedElement): dada duas *Operation* A e B de entrada, produzindo C como saída. Logo,
   If (A.visibility = private AND B.visibility = private)
   then C.visibility = private
   else C.visibility = public

10. *isQuery: Boolean*: dada duas *Operation* A e B de entrada, produzindo C como resultado da composição. C.*isQuery* = A.*isQuery*.

11. *postCondition: Constraint*: devem ser compostas de acordo com a regra MeR9.

12. *packageImport: PackageImport* (de Namespace): deve ser composto de acordo com a regra MeR4.

13. *ownedRule: Constraint* (de Namespace): as *ownedRule* devem ser compostas de acordo com a MeR9.

MeR7 Parameter Merge Rule

1. *ownedComment: Comment* (de Element): composto de acordo com MeR1.

2. *name: String* (de NamedElement): para compor dois *Parameter*, eles devem ter nomes iguais. Assim, dada dois *Parameter* A e B de entrada, produzindo C como resultado da composição. Logo, A + B = C, onde C.*name* = (A.*name* OR B.*name*).

3. *isOrdered: Boolean* (de MultiplicityElement): dado dois *Parameter* A e B de entrada, produzindo C como resultado da composição. Logo, A.*isOrdered* = B.*isOrdered* OR C.*isOrdered*.

4. *isUnique: Boolean* (de MultiplicityElement): dado dois *Parameter* A e B de entrada, produzindo C como resultado da composição. Logo, A.*isUnique* = B.*isUnique* AND C.*isUnique*.

5. *lower: Integer* (de MultiplicityElement): dado dois *Parameter* A e B de entrada, produzindo C como resultado da composição. Logo, A.*lower* é igual ao menor valor entre B.*lower* e C.*lower*.

6. *default: String*: dado dois *Parameter* A e B de entrada, produzindo C como resultado da composição. A.*default* e B.*default* devem ser iguais. Logo, C.*default* = B.*default* OR B.*default*.

7. *visibility: VisibilityKind* (de NamedElement): dado dois *Parameter* A e B de entrada, produzindo C como saída. Logo,
   If (A.visibility = private AND B.visibility = private)

then C.visibility = private
else C.visibility = public

8. *upper: UnlimitedNatural* (de MultiplicityElement): dado dois *Parameter* A e B de entrada, produzindo C como resultado da composição. Logo, A.*upper* é igual ao maior valor entre B.*upper* e C.*upper*.

9. *direction: ParameterDirectionKind*: dado dois *Parameter* A e B de entrada, produzindo C como resultado da composição. Logo,
If (A.*direction* = B.*direction*)
then C.*direction* = A.*direction*
else If (A.*direction* = in/out AND
(B.*direction* = in OR B.*direction* = out))
then C.*direction* = in/out
else if (B.*direction* = in/out AND
(A.*direction* = in OR B.*direction* = out)) then C.*direction* = in/out

MeR8 EnumerationLiteral Merge Rule

1. *name: String* (de NamedElement): para compor dois *EnumerationLiteral*, eles devem ter nomes iguais. Assim, dado dois *EnumerationLiteral* A e B de entrada, produzindo C como resultado da composição. Logo, A + B = C, onde C.*name* = (A.*name* OR B.*name*).

2. *visibility: VisibilityKind* (de NamedElement): dado dois *EnumerationLiteral* A e B de entrada, produzindo C como saída. Logo,
If (A.visibility = private AND B.visibility = private)
then C.visibility = private
else C.visibility = public

MeR9 Constraint Merge Rule

1. *name: String* (de NamedElement): para compor duas *Constraint*, elas devem ter nomes iguais. Assim, dada duas *Constraint* A e B de entrada, produzindo C como resultado da composição. Logo, A + B = C, onde C.*name* = (A.*name* OR B.*name*).

2. *ownedComment: Comment* (de Element): composto de acordo com MeR1.

3. *constrainedElement: Element*: deve ser igual.

4. *constrainedElement: Element*: deve ser igual.

5. *visibility: VisibilityKind* (de NamedElement): dada duas *Constraint* A e B de entrada, produzindo C como saída. Logo,

If (A.visibility = private AND B.visibility = private)
then C.visibility = private
else C.visibility = public

MeR10 Enumeration Merge Rule

1. *name: String* (de NamedElement): para compor duas *Enumeration*, elas devem ter nomes iguais. Assim, dada dois *Enumeration* A e B de entrada, produzindo C como resultado da composição. Logo, A + B = C, onde C.*name* = (A.*name* OR B.*name*).

2. *visibility: VisibilityKind* (de NamedElement): dado dois *EnumerationLiteral* A e B de entrada, produzindo C como saída. Logo,
   If (A.visibility = private AND B.visibility = private)
   then C.visibility = private
   else C.visibility = public

3. *ownedComment: Comment* (de Element): composto de acordo com MeR1.

4. *ownedAttribute: Property*: composto de acordo com MeR5.

5. *ownedOperation: Operation*: composto de acordo com MeR6.

6. *isAbstract: Boolean*: dado duas *Enumeration* A e B de entrada, produzindo C como saída. Logo, C.*isAbstract* = A.*isAbstract* AND B.*isAbstract*.

7. *packageImport: PackageImport* (de Namespace): deve ser composto de acordo com a regra MeR4.

8. *elementImport: ElementImport* (de Namespace): composto de acordo com a MeR3.

9. *ownedRule: Constraint* (de Namespace): as *ownedRule* devem ser compostas de acordo com a MeR9.

10. *general: Classifier*: deve ser composto por união. Dado duas *Enumeration* A (subclasse de E)e B (subclasse de F), produzindo C (subclasse de E e F) como saída.

MeR11 Stereotype Merge Rule

1. *ownedComment: Comment* (de Element): composto de acordo com a MeR1

2. *name: String* (de NamedElement): para compor dois *Stereotype*, eles devem ter nomes equivalentes. Assim, dado *Stereotype* A e B de entrada, produzindo C como resultado da composição. Logo, A + B = C, onde C.*name* = (A.*name* ou B.*name*).

3. *ownedRule: Constrain* (de Namespace): as *ownedRule* devem ser compostas de acordo com a MeR9.

4. *elementImport: ElementImport* (de Namespace): composto de acordo com a MeR3.

5. *packageImport: PackageImport* (de Namespace): composto de acordo com a MeR4.

6. *icon: Image*: dado dois *Stereotype* A e B de entrada, produzindo C como saída. Logo, C.*icon* = A.*icon* OR B.*icon*.

7. *isAbstract: Boolean*: dado dois *Stereotype* A e B de entrada, produzindo C como saída. Logo, C.*isAbstract* = A.*isAbstract* AND B.*isAbstract*.

8. *visibility: VisibilityKind* (de NamedElement): dado dois *Stereotype* A e B de entrada, produzindo C como saída. Logo,
   If (A.visibility = private and B.visibility = private)
   then C.visibility = private
   else C.visibility = public

9. *superClass: Class*: deve ser composto por união. Dado dois *Stereotype* A (subclasse de E)e B (subclasse de F), produzindo C (subclasse de E e F) como saída.

10. *ownedAttribute: Property*: dado dois *Stereotype* A e B de entrada, produzindo C como saída. Todos os atributos de A e de B devem ser compostos de acordo com a regra MeR5.

11. *ownedOperation: Operation*: composto de acordo com a MeR6.

MeR12 Association Merge Rule

1. *ownedComment: Comment* (de Element): composto de acordo com a MeR1

2. *name: String* (de NamedElement): para compor dois *Stereotype*, eles devem ter nomes equivalentes. Assim, dado *Stereotype* A e B de entrada, produzindo C como resultado da composição. Logo, A + B = C, onde C.*name* = (A.*name* ou B.*name*).

3. *isDerived: Boolean*: dada duas *Association* A e B de entrada, produzindo C como resultado da composição. Logo, A.*isDerived* = B.*isDerived* OR C.*isDerived*.

4. *ownedRule: Constrain* (de Namespace): as *ownedRule* devem ser compostas de acordo com a MeR9.

5. *elementImport: ElementImport* (de Namespace): composto de acordo com a MeR3.

6. *packageImport: PackageImport* (de Namespace): composto de acordo com a MeR4.

7. *visibility: VisibilityKind* (de NamedElement): dado dois *Stereotype* A e B de entrada, produzindo C como saída. Logo,
   If (A.visibility = private and B.visibility = private)
   then C.visibility = private
   else C.visibility = public

8. *general: Classifier*: deve ser composto por união. Dado dois *Stereotype* A (subclasse de E) e B (subclasse de F), produzindo C (subclasse de E e F) como saída.

9. *navigableOwnedEnd: Property*: deve ser composto por união. Dado dois *Stereotype* A e B, produzindo C como saída. Logo, C terá todos os *navigableOwnedEnd* de A e B.

10. *ownedEnd: Property*: deve ser igual. Para compor duas *Association* A e B, elas devem ter seus *ownedEnd* iguais. Assim, C.*ownedEnd* = A.*ownedEnd* OR B.*ownedEnd*

11. *memberEnd: Property*: deve ser igual. Para compor duas *Association* A e B, elas devem ter seus *memberEnd* iguais. Assim, C.*memberEnd* = A.*memberEnd* OR B.*memberEnd*

MeR13 Profile Merge Rule

1. *ownedComment: Comment* (de Element): composto de acordo com a MeR1

2. *name: String* (de NamedElement): para compor dois *Stereotype*, eles devem ter nomes equivalentes. Assim, dado *Stereotype* A e B de entrada, produzindo C como resultado da composição. Logo, A + B = C, onde C.*name* = (A.*name* ou B.*name*).

3. *ownedRule: Constrain* (de Namespace): as *ownedRule* devem ser compostas de acordo com a MeR9.

4. *elementImport: ElementImport* (de Namespace): composto de acordo com a MeR3.

5. *packageImport: PackageImport* (de Namespace): composto de acordo com a MeR4.

6. *visibility: VisibilityKind* (de NamedElement): dado dois *Stereotype* A e B de entrada, produzindo C como saída. Logo,
   If (A.visibility = private and B.visibility = private)
   then C.visibility = private
   else C.visibility = public

7. *metamodelReference: PackageImport*: deve ser igual.

8. *ownedStereotype: Stereotype*: deve ser composto de acordo com MeR11.

Apêndice C

Modelagem em *Alloy*

C.1 Modelagem do metamodelo da UML

Com o objetivo de modelar o metamodelo da UML em *Alloy* foi necessário fazer algumas simplificações a fim de tornar a análise possível, como já mencionado. Agora é Apresentado o metamodelo dos *profiles* definidos no metamodelo da UML.

```
module UMLProfileMetamodel

open util/boolean as Bool

sig String{}
abstract sig CompositionStrategy {}

abstract sig MatchStrategy {}
sig OverrideStrategy extends CompositionStrategy{}
sig UnionStrategy extends CompositionStrategy{}
sig MergeStrategy extends CompositionStrategy{}
sig DefaultMatchStrategy extends MatchStrategy{}
sig PartialMatchStrategy extends MatchStrategy{}
sig CompleteMatchStrategy extends MatchStrategy{}
abstract sig ComposableElement extends Element{}
abstract sig CompositeElement extends ComposableElement{}

abstract sig Element {
  name: String,
  isPrivate: Bool
}
```

```
abstract sig TypedMultiplicityElement extends ComposableElement {
  isUnique: Bool,
  isOrdered: Bool,
  lower: Int,
  upper: Int,
  type: ComposableElement
}{
  int upper > 0 && int lower >= 0 && int upper >= int lower
}


sig Package extends CompositeElement {
  ownedMembers: set CompositeElement
  profileApplication: set ProfileApplication
}{
  all a: ownedMembers | a.isPrivate = False || a.isPrivate = True
  all a1, a2:ownedMembers & Association |
     a1.@name = a2.@name => a1=a2
  all e1, e2:ownedMembers & Enumeration |
     e1.@name = e2.@name => e1=e2
  all i1, i2:ownedMembers & Interface |
     i1.@name = i2.@name => i1=i2
  all c1, c2:ownedMembers & Class | c1.@name = c2.@name => c1=c2
  all s1, s2:ownedMembers & Stereotype |
     s1.@name = s2.@name => s1=s2
}


sig Class extends CompositeElement {
  isAbstract: Bool,
  ownedAttributes: set Property,
  ownedOperations: set Operation,
  superClass: set Class
  extension: set Extension
}{
  this not in this.^@superClass
  all a1, a2:ownedAttributes | a1.@name = a2.@name => a1=a2
  all o1, o2:ownedOperations | o1.@name = o2.@name => o1=o2
```

```
  one p: Package | this in p.ownedMembers
}

sig Association extends CompositeElement {
  isDerived: Bool,
  memberEnds: set Property,
  ownedEnds: set Property,
  navigableOwnedEnds: set Property
}{
  navigableOwnedEnds in ownedEnds && ownedEnds in memberEnds
  #memberEnds >= 2
  (some end:memberEnds | end.isComposite = True) => #memberEnds = 2
  #memberEnds >2 => memberEnds = ownedEnds
  all e1, e2:memberEnds | e1.@name = e2.@name => e1=e2
}

sig Property extends TypedMultiplicityElement {
  isComposite:Bool,
  isDerived:Bool,
  isDerivedUnion:Bool,
  isReadOnly:Bool,
  default: String
}{
  one a:Association | this in a.ownedEnds ||
  one c:Class | this in c.ownedAttributes ||
  one s:Stereotype | this in s.ownedAttributes
  (isComposite = True) => ( int upper =< 1)
  isReadOnly = True => ((some a:Association | this in a.
    navigableOwnedEnds) || (some c:Class | this in c.ownedAttributes))
  isDerivedUnion = True => (isDerived = True && isReadOnly = True)
}

sig Operation extends CompositeElement {
  ownedParameters: set Parameter,
  isQuery: Bool
}{
  one c:Class | this in c.ownedOperations
  lone p:ownedParameters | p.isReturn = True
  all p1, p2: ownedParameters | p1.@name = p2.@name => p1=p2
```

```
}

sig Parameter extends TypedMultiplicityElement {
  isReturn: Bool,
  default: String
}{
  one o:Operation | this in o.ownedParameters
}

sig EnumerationLiteral extends ComposableElement { }{
  one e: Enumeration | this in e.ownedLiterals
}

sig Enumeration extends CompositeElement{
  ownedLiterals: set EnumerationLiteral
}{
  all a, b:ownedLiterals | a.@name = b.@name => a = b
  one p: Profile | this in p.ownedMembers
}

sig Image extends ComposableElement {
  content:  one String,
  location: one String,
  format:   one String
}

sig Stereotype extends Class {
  icon: set Image
}{
  this not in this.^@superClass
  all a: Stereotype | this.superClass in Stereotype
  all i1, i2:icon | (i1.@content = i2.@content &&
  i1.location = i2.location &&
  i1.format = i2.format) => i1=i2
  one p:Profile | this in p.ownedStereotypes
  all a: Element, b: Stereotype |
    (b.@name = a.@name) => b = a
  all a: Element, b: Stereotype |
    (a not in Stereotype) => (b.@name = a.@name)
```

```
}


sig ExtensionEnd extends Property {
  lower: Int,
  type: one Stereotype
}{
  one e: Extension | this in e.ownedEnd
  (int lower = 0 || int lower = 1) && int lower = 1
  #type = 1 && type in Stereotype
}

sig Extension extends Association {
  isRequired: Bool,
  ownedEnd: one ExtensionEnd
}{
  #ownedEnd = 1
  one c: Class | this in c.extension iff this.@isRequired = True
  all a: ownedEnd | a.type not in Class
  all a: memberEnds | isRequired = True => #a = 2
}

sig Profile extends Package {
  ownedStereotypes: set Stereotype
}{
  one p:ProfileApplication | this in p.appliedProfile
}

sig ProfileApplication extends ComposableElement {
  isStrict: Bool,
  appliedProfile: one Profile
}{
  (isStrict = True) => #appliedProfile = 1
}

fun min[a: Int, b: Int]: Int {
  (int a < int b) => a else b
}
```

```
fun max(a:Int, b:Int):Int {
  (int a > int b ) => a else b
}

fun conformingType(a:CompositeElement, b:CompositeElement):String {
  checkSuperType[a, b] && (a.name = b.name) => a.name
  else (isSuperType[a, b]) => b.name else a.name
}

pred checkSuperType(a:CompositeElement, b:CompositeElement){
  (a in Class && b in Class) ||
  (a in Stereotype && b in Stereotype)
}

pred isSuperType(a:CompositeElement, b:CompositeElement) {
  isSuperTypeClass[a, b] ||
  isSuperTypeStereotype[a, b]
}

pred isSuperTypeClass(a:Class, b:Class) {
  (a in Class && b in Class && a.name in (b.^superClass).name)
}

pred isSuperTypeStereotype(a:Stereotype, b:Stereotype) {
  (a in Stereotype && b in Stereotype &&
  a.name in (b.^superClass).name)
}

pred conforms(a:Class, b:Class) {
  a.name = b.name || isSuperType[a, b] || isSuperType[ b, a]
}

fun ownedAssociations(p:Package): set Association {
  p.ownedMembers & Association
}

fun ownedClasses(p:Package) : set Class {
  p.ownedMembers & Class
}
```

```
fun returnResult(o:Operation) : set Parameter {
  {result : o.ownedParameters | isTrue[result.isReturn]}
}

fun ownedStereotype(p:Profile) : set Stereotype {
  p.ownedStereotypes & Stereotype
}
```

C.2 Modelagem do mecanismo de composição

Agora é apresentado a modelagem do mecanismo de composição em Alloy.

```
module CompositionRelationship

open util/boolean as Bool

open UMLProfileMetamodel

pred compositionRelationship(receiving: CompositeElement, merged:
  CompositeElement, resulting: CompositeElement,
    matchStrategy: MatchStrategy,
    compositionStrategy: CompositionStrategy ){
    mergeOperator[receiving, merged, resulting,
      matchStrategy, compositionStrategy]
}


pred mergeOperator(receiving: CompositeElement, merged:
  CompositeElement, resulting: CompositeElement,
  match: MatchStrategy, strategy: CompositionStrategy ){

    (strategy in UnionStrategy && receiving in Profile &&
    merged in Profile && resulting in Profile)
    => strategyMergeFacade[receiving, merged,
    resulting, match, strategy] ||
```

```
    (receiving in Profile && merged in Profile && resulting
    in Profile && strategy in OverrideStrategy )
    => some a:receiving.ownedStereotypes, b:merged.ownedStereotypes,
    c:resulting.ownedStereotypes |
    strategyMergeFacade[a, b, c, match, strategy] ||
    (receiving in Profile && merged in Profile && resulting
    in Profile && strategy in MergeStrategy )
    => some a:receiving.ownedStereotypes, b:merged.ownedStereotypes,
    c:resulting.ownedStereotypes |
    strategyMergeFacade[a, b, c, match, strategy]
}


pred strategyMergeFacade(receiving:ComposableElement,
  merged:ComposableElement, resulting:ComposableElement,
  matchStrategy: MatchStrategy,
  compositionStrategy: CompositionStrategy ){
    (compositionStrategy in OverrideStrategy)
       =>  overrideStrategyMerge[receiving, merged,
       resulting, matchStrategy ] ||
    (compositionStrategy in UnionStrategy)
       =>  unionStrategyMerge[receiving, merged,
       resulting, matchStrategy]    ||
    (compositionStrategy in MergeStrategy)
       =>  mergeStrategyMerge[receiving, merged,
       resulting, matchStrategy]
}


pred overrideStrategyMerge(a:ComposableElement, b:ComposableElement,
   c:ComposableElement, d: MatchStrategy){
    (a in Class && b in Class && classOSMergeRule[a, b, c, d]) ||
    (a in Association && b in Association &&
       associationOSMergeRule[a, b, c, d])  ||
    (a in Stereotype  && b in Stereotype  &&
       stereotypeOSMergeRule[a, b, c, d])   ||
    (a in Enumeration && b in Enumeration &&
       enumerationOSMergeRule[a, b, c, d])
}
```

```
pred unionStrategyMerge(receiving:ComposableElement,
  merged:ComposableElement, resulting:ComposableElement,
  matchStrategy: MatchStrategy){
    (receiving in Profile       && merged in Profile
    && profileUSMergeRule[receiving, merged, resulting, matchStrategy])
}


pred mergeStrategyMerge(receiving:ComposableElement,
  merged:ComposableElement, resulting:ComposableElement,
  matchStrategy: MatchStrategy){
    (receiving in Class && merged in Class
       && classMSMergeRule[receiving, merged, resulting, matchStrategy])
    (receiving in Association && merged in Association
       && associationMSMergeRule[receiving, merged,
       resulting, matchStrategy]) ||
    (receiving in Stereotype  && merged in Stereotype
       && stereotypeMSMergeRule[receiving, merged,
       resulting, matchStrategy]) ||
    (receiving in Enumeration && merged in Enumeration
       && enumerationMSMergeRule[receiving, merged,
       resulting, matchStrategy])
}


pred classOSMergeRule(receiving:Class, merged:Class,
  resulting:Class, match: MatchStrategy) {
  resulting.name = receiving.name
  resulting.isAbstract = receiving.isAbstract
  resulting.isPrivate = receiving.isPrivate
  all a1:receiving.ownedAttributes | (no a2:merged.ownedAttributes |
  matchOperator[a1, a2, match]) => (some a3:resulting.ownedAttributes |
    equals[a1,a3])
  all a1:receiving.ownedAttributes | (some a2:merged.ownedAttributes |
    equals[a1, a2]) => (some
  a3:resulting.ownedAttributes | equals[a1, a3])
  all a1:receiving.ownedAttributes, a2:merged.ownedAttributes |
```

```
    matchOperator[a1, a2, match] => (some a3:resulting.ownedAttributes |
  propertyOSMergeRule[a1, a2, a3, match])
  all a1:resulting.ownedAttributes | some a2:receiving.ownedAttributes |
    matchOperator[a1, a2, match]
  all o1:receiving.ownedOperations | (no o2:merged.ownedOperations |
  matchOperator[o1, o2, match]) => (some o3:resulting.ownedOperations |
    equals[o1, o3])
  all o1:receiving.ownedOperations, o2:merged.ownedOperations |
  matchOperator[o1, o2, match]  => (some o3:resulting.ownedOperations |
    operationOSMergeRule[o1, o2, o3, match])
  all o1:receiving.ownedOperations | (some o2:merged.ownedOperations |
  equals[o1, o2]) => (some o3:resulting.ownedOperations | equals[o1, o3])
  all o1:resulting.ownedOperations | some o2:receiving.ownedOperations |
    matchOperator[o1, o2, match]
  resulting.superClass.name = receiving.superClass.name
}

pred associationOSMergeRule(receiving:Association,
  merged:Association,
  resulting: Association, match: MatchStrategy) {
  resulting.name = receiving.name
  resulting.isDerived = receiving.isDerived
  resulting.isPrivate = receiving.isPrivate
  all e1:receiving.ownedEnds, e2:merged.ownedEnds |
  matchOperator[e1, e2, match] => some e3:resulting.ownedEnds |
  propertyOSMergeRule[e1, e2, e3, match] && ((e1 in
  receiving.navigableOwnedEnds) <=> e3 in resulting.navigableOwnedEnds)

}

pred stereotypeOSMergeRule(receiving: Stereotype, merged:
  Stereotype, resulting: Stereotype, match: MatchStrategy){

  classOSMergeRule[receiving, merged, resulting, match]
  all a1:receiving.icon | (no a2:merged.icon |
   matchOperator[a1, a2, match]) =>
   some a3:resulting.icon | equals[a3, a1]
  all a1:receiving.icon | (some
   a2:merged.icon | equals[a1, a2]) => (some
```

```
   a3:resulting.icon | equals[a1, a3])
  all a1:receiving.icon, a2:merged.icon |
   matchOperator[a1, a2, match] => (some a3:resulting.icon |
   imageOSMergeRule[a1, a2, a3, match])
  all a1:resulting.icon | some a2:receiving.icon |
   matchOperator[a1, a2, match]

}

pred enumerationOSMergeRule(receiving:Enumeration,
  merged:Enumeration, resulting:Enumeration, match: MatchStrategy){

  all a1:receiving.ownedLiterals | (no a2:merged.ownedLiterals |
   matchOperator[a1, a2, match]) => (some a3:resulting.ownedLiterals |
   equals[a1,a3])
  all a1:receiving.ownedLiterals | (some
   a2:merged.ownedLiterals | equals[a1, a2]) => (some
   a3:resulting.ownedLiterals | equals[a1, a3])
  all a1:receiving.ownedLiterals, a2:merged.ownedLiterals |
   matchOperator[a1, a2, match] => (some a3:resulting.ownedLiterals |
   equals[a1, a3])
  all a1:resulting.ownedLiterals | some a2:receiving.ownedLiterals |
   matchOperator[a1, a2, match]
}

pred propertyOSMergeRule(receiving:Property, merged:Property,
  resulting:Property, match: MatchStrategy) {
  resulting.name = receiving.name
  resulting.isUnique = receiving.isUnique
  resulting.isComposite = receiving.isComposite
  resulting.isReadOnly = receiving.isReadOnly
  resulting.isOrdered = receiving.isOrdered
  resulting.lower = receiving.lower
  resulting.upper = receiving.upper
  resulting.isPrivate = receiving.isPrivate
  resulting.isDerivedUnion = receiving.isDerivedUnion
  resulting.isDerived = receiving.isDerived

}
```

```
pred operationOSMergeRule(receiving:Operation, merged:Operation,
  resulting: Operation, match: MatchStrategy) {
  resulting.name = receiving.name
  resulting.isQuery = receiving.isQuery
  resulting.isPrivate = receiving.isPrivate
  all p1:receiving.ownedParameters, p2:merged.ownedParameters |
  matchOperator[p1, p2, match] => some p3:resulting.ownedParameters |
  parameterOSMergeRule[p1, p2, p3, match]
  all p1:resulting.ownedParameters |
  some p2:receiving.ownedParameters + merged.ownedParameters |
  matchOperator[p1, p2, match]
}

pred parameterOSMergeRule(receiving:Parameter, merged:Parameter,
  resulting: Parameter, match: MatchStrategy) {

  receiving.name = resulting.name
  resulting.isReturn = receiving.isReturn
  resulting.isPrivate = receiving.isPrivate
  isTrue[receiving.isReturn] => (resulting.isUnique = receiving.isUnique
     && resulting.isOrdered = receiving.isOrdered &&
     resulting.lower = receiving.lower
     && resulting.upper = receiving.upper)
  isFalse[receiving.isReturn] => (resulting.isUnique = receiving.isUnique
    && resulting.isOrdered = receiving.isOrdered
    && resulting.lower = receiving.lower
    && resulting.upper = receiving.upper)
}

pred imageOSMergeRule(receiving:Image, merged:Image,
 resulting:Image, match: MatchStrategy){
  resulting.content = receiving.content
  resulting.location = receiving.location
  resulting.format = receiving.format
}

pred profileUSMergeRule(receiving: Profile, merged: Profile,
  resulting: Profile, match: MatchStrategy) {
```

```
mergeProfileIsValid[receiving, merged, match]
all a1:receiving.ownedMembers | (no a2:merged.ownedMembers |
matchOperator[a1, a2, match]) => (some a3:resulting.ownedMembers |
 equals[a1,a3])
all a1:merged.ownedMembers | (no a2:receiving.ownedMembers |
matchOperator[a1, a2, match]) => (some a3:resulting.ownedMembers |
 equals[a1,a3])
all a1:receiving.ownedMembers | (some
a2:merged.ownedMembers, a3:resulting.ownedMembers  |
 equals[a1, a2] => equals[a1, a3] && equals[a2, a3] )
all a1:merged.ownedMembers | (some
a2:receiving.ownedMembers,  a3:resulting.ownedMembers |
 equals[a1, a2] => equals[a1, a3] && equals[a2, a3])
all a1:receiving.ownedMembers, a2:merged.ownedMembers |
matchOperator[a1, a2, match] => (some a3:resulting.ownedMembers |
 equals[a1, a3] && equals[a2, a3])
all a1:resulting.ownedMembers |
all a2:receiving.ownedMembers + merged.ownedMembers  |
  equals[a1, a1, match]
all a1:resulting.ownedMembers | matchOperator[a1, a1, match]
all a1:receiving.ownedStereotypes | (no a2:merged.ownedStereotypes |
matchOperator[a1, a2, match]) =>
(some a3:resulting.ownedStereotypes | equals[a1,a3])
all a1:merged.ownedStereotypes | (no a2:receiving.ownedStereotypes |
  matchOperator[a1, a2, match]) => (some a3:resulting.ownedStereotypes |
  equals[a1,a3])

all a1:receiving.ownedStereotypes | (some
a2:merged.ownedStereotypes, a3:resulting.ownedStereotypes  |
 equals[a1, a2] => equals[a1, a3] && equals[a2, a3] )

all a1:merged.ownedStereotypes | (some
a2:receiving.ownedStereotypes,  a3:resulting.ownedStereotypes |
  equals[a1, a2] => equals[a1, a3] && equals[a2, a3])

all a1:receiving.ownedStereotypes, a2:merged.ownedStereotypes |
matchOperator[a1, a2, match] => (some a3:resulting.ownedStereotypes |
equals[a1, a3] && equals[a2, a3])
```

```
  all a1:resulting.ownedStereotypes |
   some a2:receiving.ownedStereotypes + merged.ownedStereotypes  |
   equals[a1, a2, match]

}

pred classMSMergeRule(receiving: Class, merged: Class, resulting:
  Class, match: MatchStrategy){

  resulting.name = receiving.name && merged.name = resulting.name
  resulting.isAbstract = And[receiving.isAbstract, merged.isAbstract]
  resulting.isPrivate = And[receiving.isPrivate, merged.isPrivate]
  all a1:receiving.ownedAttributes | (no a2:merged.ownedAttributes |
  matchOperator[a1, a2, match]) =>
  (some a3:resulting.ownedAttributes | equals[a1,a3])
  all a1:merged.ownedAttributes | (no a2:receiving.ownedAttributes |
   matchOperator[a1, a2, match]) => (some a3:resulting.ownedAttributes |
   equals[a1, a3])
  all a1:receiving.ownedAttributes | (some
  a2:merged.ownedAttributes | equals[a1, a2]) => (some
  a3:resulting.ownedAttributes | equals[a1, a3])
  all a1:receiving.ownedAttributes, a2:merged.ownedAttributes |
   matchOperator[a1, a2, match] => (some a3:resulting.ownedAttributes |
   propertyMSMergeRule[a1, a2, a3, match])
  all a1:resulting.ownedAttributes | some a2:receiving.ownedAttributes
  + merged.ownedAttributes | matchOperator[a1, a2, match]
  all o1:receiving.ownedOperations | (no o2:merged.ownedOperations |
  matchOperator[o1, o2, match]) => (some o3:resulting.ownedOperations |
   equals[o1, o3])
  all o1:merged.ownedOperations | (no o2:receiving.ownedOperations |
  matchOperator[o1, o2, match]) => (some o3:resulting.ownedOperations |
   equals[o1,o3])
  all o1:receiving.ownedOperations | (some o2:merged.ownedOperations |
   matchOperator[o1, o2, match]) => (some o3:resulting.ownedOperations |
   equals[o1,o3])
  all o1:receiving.ownedOperations, o2:merged.ownedOperations |
  matchOperator[o1, o2, match]  => (some o3:resulting.ownedOperations |
   operationMSMergeRule[o1, o2, o3, match])
  all o1:resulting.ownedOperations | some o2:merged.ownedOperations +
```

```
  receiving.ownedOperations | matchOperator[o1, o2, match]
  resulting.superClass.name = merged.superClass.name +
   receiving.superClass.name

}

pred associationMSMergeRule(receiving:Association,
merged:Association, resulting: Association, match: MatchStrategy) {
  resulting.name = receiving.name + merged.name
  resulting.isDerived = Or[receiving.isDerived, merged.isDerived]
  resulting.isPrivate = And[receiving.isPrivate, merged.isPrivate]
  all e1:receiving.ownedEnds, e2:merged.ownedEnds | equals[e1, e2] =>
   (some e3:resulting.ownedEnds |
   propertyMSMergeRule[e1, e2, e3, match])
  all e1:receiving.ownedEnds, e2:merged.ownedEnds |
  matchOperator[e1, e2, match] =>
   some e3:resulting.ownedEnds |
   propertyMSMergeRule[e1, e2, e3, match] &&
   ((e1 in receiving.navigableOwnedEnds ||
   e2 in merged.navigableOwnedEnds)
   <=> e3 in resulting.navigableOwnedEnds)
}

pred stereotypeMSMergeRule(receiving: Stereotype, merged:
 Stereotype, resulting: Stereotype, match: MatchStrategy){
  classMSMergeRule[receiving, merged, resulting, match]
  all a1:receiving.icon | (no a2:merged.icon |
  matchOperator[a1, a2, match]) => (some a3:resulting.icon |
   equals[a1,a3])
  all a1:merged.icon | (no a2:receiving.icon |
   matchOperator[a1, a2, match]) => (some a3:resulting.icon |
   equals[a1, a3])
  all a1:receiving.icon | (some
   a2:merged.icon | equals[a1, a2]) => (some
   a3:resulting.icon | equals[a1, a3])
  all a1:receiving.icon, a2:merged.icon |
   matchOperator[a1, a2, match] => (some a3:resulting.icon |
   imageMSMergeRule[a1, a2, a3, match])
  all a1:resulting.icon | some a2:receiving.icon
```

```
   + merged.icon | matchOperator[a1, a2, match]
}

pred enumerationMSMergeRule(receiving: Enumeration, merged:
 Enumeration, resulting: Enumeration, match: MatchStrategy){
  resulting.name = receiving.name && merged.name = resulting.name
  resulting.isPrivate = And[receiving.isPrivate, merged.isPrivate]
  all a1:receiving.ownedLiterals | (no a2:merged.ownedLiterals |
   matchOperator[a1, a2, match]) => (some a3:resulting.ownedLiterals |
   equals[a1,a3])
  all a1:merged.ownedLiterals | (no a2:receiving.ownedLiterals |
   matchOperator[a1, a2, match]) => (some a3:resulting.ownedLiterals |
   equals[a1, a3])
  all a1:receiving.ownedLiterals | (some
   a2:merged.ownedLiterals | equals[a1, a2]) => (some
   a3:resulting.ownedLiterals | equals[a1, a3])
  all a1:receiving.ownedLiterals, a2:merged.ownedLiterals |
   matchOperator[a1, a2, match] => (some a3:resulting.ownedLiterals |
   enumerationLiteralMSMergeRule[a1, a2, a3, match])
  all a1:resulting.ownedLiterals | some a2:receiving.ownedLiterals
  + merged.ownedLiterals | matchOperator[a1, a2, match]
}

pred propertyMSMergeRule(receiving:Property, merged:Property,
 resulting:Property, match: MatchStrategy) {
  receiving.name = resulting.name && merged.name = resulting.name
  resulting.isUnique = Or[receiving.isUnique,merged.isUnique]
  resulting.isComposite = Or[receiving.isComposite, merged.
   isComposite]
  resulting.isReadOnly = And[receiving.isReadOnly,
   merged.isReadOnly]
  resulting.isOrdered = Or[receiving.isOrdered,
   merged.isOrdered]
  resulting.lower = max[receiving.lower,
   merged.lower]
  resulting.upper = min[receiving.upper, merged.upper]
  resulting.isPrivate = And[receiving.isPrivate, merged.isPrivate]
  resulting.isDerivedUnion = Or[receiving.isDerivedUnion, merged.
   isDerivedUnion]
```

```
  resulting.isDerived = Or[receiving.isDerived,
   merged.isDerived]
}

pred operationMSMergeRule(receiving:Operation, merged:Operation,
  resulting: Operation, match: MatchStrategy) {
  resulting.name = receiving.name && resulting.name = merged.name
  resulting.isQuery = Or[receiving.isQuery, merged.isQuery]
  resulting.isPrivate = And[receiving.isPrivate, merged.isPrivate]
  all p1:receiving.ownedParameters, p2:merged.ownedParameters |
  equals[p1, p2] => some p3:resulting.ownedParameters |
   equals[p3, p1]
  all p1:receiving.ownedParameters, p2:merged.ownedParameters |
   matchOperator[p1, p2, match] => some p3:resulting.ownedParameters |
   parameterMSMergeRule[p1, p2, p3, match]
  all p1:resulting.ownedParameters |
   some p2:receiving.ownedParameters + merged.ownedParameters |
   matchOperator[p1, p2, match]
}

pred parameterMSMergeRule(receiving:Parameter, merged:Parameter,
  resulting: Parameter, match: MatchStrategy) {
  resulting.name = receiving.name && resulting.name = merged.name
  resulting.isReturn = receiving.isReturn &&
   resulting.isReturn = merged.isReturn
  resulting.isPrivate = And[receiving.isPrivate, merged.isPrivate]
  isTrue[receiving.isReturn] => (resulting.isUnique =
   Or[receiving.isUnique, merged.isUnique] &&
    resulting.isOrdered = Or[receiving.isOrdered, merged.isOrdered] &&
    resulting.lower = max[receiving.lower, merged.lower] &&
    resulting.upper = min[receiving.upper, merged.upper])
  isFalse[receiving.isReturn] => (resulting.isUnique = And[receiving.
   isUnique, merged.isUnique]
    && resulting.isOrdered = And[receiving.isOrdered, merged.isOrdered]
    && resulting.lower = min[receiving.lower, merged.lower]
    && resulting.upper = max[receiving.upper, merged.upper])
}

pred imageMSMergeRule(receiving:Image, merged: Image,
```

```
resulting:Image, match: MatchStrategy){
  resulting.content = receiving.content + merged.content
  resulting.location = receiving.location + merged.location
  resulting.format = receiving.format + merged.format
}

pred enumerationLiteralMSMergeRule(a: EnumerationLiteral, b:
  EnumerationLiteral, c: EnumerationLiteral, match: MatchStrategy) {
  c.name = a. name && c.name = b.name
  c.isPrivate = And[a.isPrivate,b.isPrivate]
}

pred mergeProfileIsValid(receiving:Profile, merged: Profile, match:
MatchStrategy) {
  all s1:ownedStereotype[receiving], s2:ownedStereotype[merged] |
  matchOperator[s1, s2, match] => stereotypeConstraints[s1, s2, match]
  all e1:ownedClasses[receiving], e2:ownedClasses[merged] |
  matchOperator[e1, e2, match] => classConstraints[e1, e2, match]
  all e1:ownedAssociations[receiving], e2:ownedAssociations[merged]
   | matchOperator[e1, e2, match] =>
   associationConstraints[e1, e2, match]
}


pred stereotypeConstraints(receiving:Stereotype, merged:Stereotype,
match: MatchStrategy) {
  all a1:receiving.ownedAttributes, a2:merged.ownedAttributes |
  matchOperator[a1, a2, match] => propertyConstraints[a1, a2]
  all o1:receiving.ownedOperations, o2:merged.ownedOperations |
  matchOperator[o1, o2, match] => operationConstraints[o1, o2]
  all i1:receiving.icon, i2:merged.icon | matchOperator[i1, i2, match]
}

pred associationConstraints(receiving:Association,
  merged:Association, match: MatchStrategy) {
  all e1:receiving.memberEnds, e2:merged.memberEnds |
   ((matchOperator[e1, e2, match] &&
  e2 in merged.ownedEnds)=> e1 in receiving.ownedEnds) &&
   (matchOperator[e1, e2, match] && e1 in receiving.ownedEnds =>
```

```
   e2 in merged.ownedEnds)
  all e1:receiving.memberEnds, e2:merged.memberEnds |
   matchOperator[e1, e2, match] => propertyConstraints[e1, e2]
}

pred operationConstraints(receiving:Operation, merged:Operation) {
  conforms[returnResult[receiving].type, returnResult[merged].type]
}

pred parameterConstraints(receiving:Parameter, merged:Parameter) {
  receiving.isUnique = merged.isUnique
}

pred classConstraints(receiving:Class, merged:Class, match:
  MatchStrategy) {
  all a1:receiving.ownedAttributes, a2:merged.ownedAttributes |
  matchOperator[a1, a2, match] => propertyConstraints[a1, a2]
  all o1:receiving.ownedOperations, o2:merged.ownedOperations |
  matchOperator[o1, o2, match] => operationConstraints[o1, o2]
}

pred propertyConstraints(receiving:Property, merged:Property) {
  conforms[receiving.type, merged.type]
  isTrue[merged.isComposite] => isTrue[receiving.isComposite]
}

pred equals(e1:Element, e2:Element) {
  (e1 in Class && e2 in Class && classMatches[e1, e2]) ||
  (e1 in Property && e2 in Property && propertyMatches[e1, e2]) ||
  (e1 in Association && e2 in Association &&
    associationMatches[e1, e2]) ||
  (e1 in Operation && e2 in Operation && operationMatches[e1, e2]) ||
  (e1 in Parameter && e2 in Parameter && parameterMatches[e1, e2])
  (e1 in Stereotype && e2 in Stereotype && stereotypeMatches[e1, e2])
  (e1 in Image && e2 in Image && imageMatches[e1, e2])
  (e1 in EnumerationLiteral && e2 in EnumerationLiteral
    && enumerationLiteralMatches[e1, e2])
}
```

```
pred matches(e1:Element, e2:Element) {
  (e1 in Class && e2 in Class && classMatches[e1, e2]) ||
  (e1 in Property && e2 in Property && propertyMatches[e1, e2]) ||
  (e1 in Association && e2 in Association &&
    associationMatches[e1, e2]) ||
  (e1 in Operation && e2 in Operation && operationMatches[e1, e2]) ||
  (e1 in Parameter && e2 in Parameter && parameterMatches[e1, e2])
  (e1 in Stereotype && e2 in Stereotype && stereotypeMatches[e1, e2])
  (e1 in Image && e2 in Image && imageMatches[e1, e2])
  (e1 in EnumerationLiteral && e2 in EnumerationLiteral &&
    enumerationLiteralMatches[e1, e2])
}

pred classMatches(c1:Class, c2:Class) {
  c1.name = c2.name
}

pred propertyMatches(p1:Property, p2:Property) {
  p1.name = p2.name
}

pred associationMatches(a1:Association, a2:Association) {
  a1.name = a2.name
  #a1.memberEnds = #a2.memberEnds
  all e1:a1.memberEnds | some e2:a2.memberEnds |
  propertyMatches[e1, e2] && conforms[e1.type, e2.type]
}

pred operationMatches(o1:Operation, o2:Operation) {
  o1.name = o2.name #o1.ownedParameters = #o2.ownedParameters
  all p1:o1.ownedParameters | some p2:o2.ownedParameters |
parameterMatches[p1, p2] }

pred parameterMatches(p1:Parameter, p2:Parameter) {
  p1.name = p2.name
  p1.isReturn = p2.isReturn
  isFalse[p1.isReturn] => (p1.type.name = p2.type.name)
}
```

```
pred enumerationLiteralMatches(a: EnumerationLiteral, b:
EnumerationLiteral) {
   a.name = b.name
}

pred stereotypeMatches(s1:Stereotype, s2:Stereotype) {
  s1.name = s2.name
}

pred imageMatches(i1:Image, i2:Image) {
  i1.name = i2.name

}

pred matchOperator(a: Element, b: Element, c: MatchStrategy){
  (c in DefaultMatchStrategy && defaultMatchStrategy[a, b]) ||
  (c in PartialMatchStrategy && partialMatchStrategy[a, b]) ||
  (c in CompleteMatchStrategy && completeMatchStrategy[a, b])
}

pred completeMatchStrategy(e1:Element, e2:Element) {
  (e1 in Class && e2 in Class && classCompleteMatchRule[e1, e2]) ||
  (e1 in Property && e2 in Property &&
    propertyCompleteMatchRule[e1, e2]) ||
  (e1 in Association && e2 in Association &&
    associationCompleteMatchRule[e1, e2]) ||
  (e1 in Operation && e2 in Operation &&
    operationCompleteMatchRule[e1, e2]) ||
  (e1 in Parameter && e2 in Parameter &&
    parameterCompleteMatchRule[e1, e2])
  (e1 in Stereotype && e2 in Stereotype &&
    stereotypeCompleteMatchRule[e1, e2])
  (e1 in Image && e2 in Image && imageCompleteMatchRule[e1, e2])
  (e1 in Enumeration && e2 in Enumeration &&
    enumerationCompleteMatchRule[e1, e2])
  (e1 in EnumerationLiteral && e2 in EnumerationLiteral &&
    enumerationLiteralCompleteMatchRule[e1, e2])
}
```

```
pred classCompleteMatchRule(a:Class, b:Class) {
  a.name = b.name
  a.isPrivate = b.isPrivate
  a.isAbstract = b.isAbstract
  a.superClass.name = b.superClass.name
  #a.ownedAttributes = #b.ownedAttributes &&
  #a.ownedOperations = #b.ownedOperations
  all p1:a.ownedAttributes | some p2:b.ownedAttributes |
    propertyCompleteMatchRule[p1, p2]
  all o1:a.ownedOperations | some o2:b.ownedOperations |
    operationCompleteMatchRule[o1, o2]
}

pred propertyCompleteMatchRule(a:Property, b:Property) {
  a.name = b.name
  a.isPrivate = b.isPrivate
  a.lower = b.lower
  a.upper = b.upper
  a.isOrdered = b.isOrdered
  a.isUnique = b.isUnique
  a.isComposite = b.isComposite
  a.isDerived = b.isDerived
  a.isDerivedUnion = b.isDerivedUnion
  a.isReadOnly = b.isReadOnly
  a.type.name = b.type.name
}

pred associationCompleteMatchRule(a:Association, b:Association) {
  a.name = b.name
  a.isDerived = b.isDerived
  a.isPrivate = b.isPrivate
  #a.memberEnds = #b.memberEnds
  #a.ownedEnds = #b.ownedEnds
  #a.navigableOwnedEnds = #b.navigableOwnedEnds
  all e1:a.memberEnds | some e2:b.memberEnds |
   propertyCompleteMatchRule[e1, e2]
  all e1:a.ownedEnds | some e2:b.memberEnds |
   propertyCompleteMatchRule[e1, e2]
  all e1:b.navigableOwnedEnds | some e2:b.navigableOwnedEnds |
```

```
    propertyCompleteMatchRule[e1, e2]
}

pred operationCompleteMatchRule(a:Operation, b:Operation) {
  a.name = b.name
  a.isQuery = b.isQuery
  a.isPrivate = b.isPrivate
  #a.ownedParameters = #b.ownedParameters
  all p1:a.ownedParameters | some p2:b.ownedParameters |
  parameterCompleteMatchRule[p1, p2]
}

pred parameterCompleteMatchRule(a:Parameter, b:Parameter) {
  a.name = b.name
  a.lower = b.lower
  a.upper = b.upper
  a.isOrdered = b.isOrdered
  a.isUnique = b.isUnique
  a.isReturn = b.isReturn
  a.isPrivate = b.isPrivate
}

pred stereotypeCompleteMatchRule(a:Stereotype, b:Stereotype) {
  a.name = b.name
  a.isPrivate = b.isPrivate
  a.isAbstract = b.isAbstract
  a.superClass.name = b.superClass.name
  #a.ownedAttributes = #b.ownedAttributes &&
  #a.ownedOperations = #b.ownedOperations
  all p1:a.ownedAttributes | some p2:b.ownedAttributes |
    propertyCompleteMatchRule[p1, p2]
  all o1:a.ownedOperations | some o2:b.ownedOperations |
    operationCompleteMatchRule[o1, o2]
  imageCompleteMatchRule[a.icon, b.icon]
}

pred imageCompleteMatchRule(a:Image, b:Image) {
  a.name = b.name
  a.location = b.location
```

```
  a.format = b.format
  a.content = b.content
}

pred enumerationCompleteMatchRule(a: Enumeration, b: Enumeration) {
  #a.ownedLiterals = #b.ownedLiterals
  all e1:a.ownedLiterals | some e2:b.ownedLiterals |
  enumerationLiteralCompleteMatchRule[e1, e2]
}

pred enumerationLiteralCompleteMatchRule(a: EnumerationLiteral, b:
EnumerationLiteral) {
   a.name = b.name
}

pred partialMatchStrategy(e1:Element, e2:Element) {
  (e1 in Class && e2 in Class &&
    classPartialMatchRule[e1, e2]) ||
  (e1 in Property && e2 in Property &&
    propertyPartialMatchRule[e1, e2]) ||
  (e1 in Association && e2 in Association &&
    associationPartialMatchRule[e1, e2]) ||
  (e1 in Operation && e2 in Operation &&
    operationPartialMatchRule[e1, e2]) ||
  (e1 in Parameter && e2 in Parameter &&
    parameterPartialMatchRule[e1, e2])
  (e1 in Stereotype && e2 in Stereotype &&
    stereotypePartialMatchRule[e1, e2])
  (e1 in Image && e2 in Image &&
    imagePartialMatchRule[e1, e2])
  (e1 in Enumeration && e2 in Enumeration &&
    enumerationPartialMatchRule[e1, e2])
  (e1 in EnumerationLiteral && e2 in EnumerationLiteral &&
    enumerationLiteralPartialMatchRule[e1, e2])
}

pred classPartialMatchRule(a:Class, b:Class) {
  a.name = b.name
  a.isPrivate = b.isPrivate
```

```
  a.isAbstract = b.isAbstract
  #a.ownedAttributes = #b.ownedAttributes &&
  #a.ownedOperations = #b.ownedOperations
  all p1:a.ownedAttributes | some p2:b.ownedAttributes |
    propertyPartialMatchRule[p1, p2]
  all o1:a.ownedOperations | some o2:b.ownedOperations |
    operationPartialMatchRule[o1, o2]
}

pred propertyPartialMatchRule(a:Property, b:Property) {
  a.name = b.name
  a.isPrivate = b.isPrivate
  a.lower = b.lower
  a.upper = b.upper
  a.isDerived = b.isDerived
  a.isDerivedUnion = b.isDerivedUnion
  a.isReadOnly = b.isReadOnly
}

pred associationPartialMatchRule(a1:Association, a2:Association) {
  a1.name = a2.name
  #a1.memberEnds = #a2.memberEnds
  all e1:a1.memberEnds | some e2:a2.memberEnds |
  propertyPartialMatchRule[e1, e2] && conforms[e1.type, e2.type]
}

pred operationPartialMatchRule(a:Operation, b:Operation) {
  a.name = b.name
  #a.ownedParameters = #b.ownedParameters
  all p1:a.ownedParameters | some p2:b.ownedParameters |
  parameterPartialMatchRule[p1, p2]
}

pred parameterPartialMatchRule(a:Parameter, b:Parameter) {
  a.name = b.name
  a.lower = b.lower
  a.upper = b.upper
  a.isPrivate = b.isPrivate
}
```

```
pred stereotypePartialMatchRule(a:Stereotype, b:Stereotype) {
  a.name = b.name
  a.isPrivate = b.isPrivate
  #a.ownedAttributes = #b.ownedAttributes &&
  #a.ownedOperations = #b.ownedOperations
  all p1:a.ownedAttributes | some p2:b.ownedAttributes |
    propertyPartialMatchRule[p1, p2]
  imagePartialMatchRule[a.icon, b.icon]
}

pred imagePartialMatchRule(a:Image, b:Image) {
  a.name = b.name
  a.location = b.location
}

pred enumerationPartialMatchRule(a: Enumeration, b: Enumeration) {
   a.name = b.name
}

pred enumerationLiteralPartialMatchRule(a: EnumerationLiteral, b:
EnumerationLiteral) {
   a.name = b.name
}

pred defaultMatchStrategy(e1:Element, e2:Element) {
  (e1 in Profile && e2 in Profile &&
    profileDefaultMatchRule[e1, e2]) ||
  (e1 in Class && e2 in Class &&
    classDefaultMatchRule[e1, e2]) ||
  (e1 in Property && e2 in Property &&
    propertyDefaultMatchRule[e1, e2]) ||
  (e1 in Association && e2 in Association &&
    associationDefaultMatchRule[e1, e2]) ||
  (e1 in Operation && e2 in Operation &&
    operationDefaultMatchRule[e1, e2]) ||
  (e1 in Parameter && e2 in Parameter &&
    parameterDefaultMatchRule[e1, e2])
  (e1 in Stereotype && e2 in Stereotype &&
```

```
    stereotypeDefaultMatchRule[e1, e2])
  (e1 in Image && e2 in Image &&
    imageDefaultMatchRule[e1, e2])
  (e1 in Enumeration && e2 in Enumeration &&
    enumerationDefaultMatchRule[e1, e2])
  (e1 in EnumerationLiteral && e2 in EnumerationLiteral &&
    enumerationLiteralDefaultMatchRule[e1, e2])
}

pred classDefaultMatchRule(c1:Class, c2:Class) {
  c1.name = c2.name
}

pred propertyDefaultMatchRule(p1:Property, p2:Property) {
  p1.name = p2.name
}

pred associationDefaultMatchRule(a1:Association, a2:Association) {
  a1.name = a2.name
  #a1.memberEnds = #a2.memberEnds
  all e1:a1.memberEnds | some e2:a2.memberEnds |
  propertyDefaultMatchRule[e1, e2] && conforms[e1.type, e2.type]
}

pred operationDefaultMatchRule(o1:Operation, o2:Operation) {
  o1.name = o2.name #o1.ownedParameters = #o2.ownedParameters
  all p1:o1.ownedParameters | some p2:o2.ownedParameters |
  parameterDefaultMatchRule[p1, p2]
}

pred parameterDefaultMatchRule(p1:Parameter, p2:Parameter) {
  p1.name = p2.name
}

pred stereotypeDefaultMatchRule(s1:Stereotype, s2:Stereotype) {
  s1.name = s2.name
}

pred imageDefaultMatchRule(i1:Image, i2:Image) {
```

```
  //i1.name = i2.name
  i1.content = i2.content
}

pred enumerationDefaultMatchRule(a: Enumeration, b: Enumeration) {
   a.name = b.name
}

pred enumerationLiteralDefaultMatchRule(a: EnumerationLiteral, b:
EnumerationLiteral) {
   a.name = b.name
}

pred profileDefaultMatchRule(a: Profile, b: Profile) {
   a.ownedStereotypes = b.ownedStereotypes
}


assert stereotypeOSMergeRuleIdenIdempotency {
   all a, b, c : Stereotype | one match: DefaultMatchStrategy |
   stereotypeOSMergeRule[a, a, c, match] =>
   stereotypeOSMergeRule[a, a, c, match]
}

assert stereotypeOSMergeRuleIdempotency {
   all a, c : Stereotype | all match: DefaultMatchStrategy |
   stereotypeOSMergeRule[a, a, c, match] =>
   stereotypeOSMergeRule[a, a, c, match]
}

assert stereotypeOSMergeRuleIdempotencyPartial {
   all a, c : Stereotype | all match: PartialMatchStrategy |
   stereotypeOSMergeRule[a, a, c, match] =>
   stereotypeOSMergeRule[a, a, c, match]
}

assert stereotypeOSMergeRuleIdempotencyComplete {
   all a, c : Stereotype | all match: CompleteMatchStrategy |
   stereotypeOSMergeRule[a, a, c, match] =>
```

```
   stereotypeOSMergeRule[a, a, c, match]
}

assert stereotypeOSMergeRuleCommutative {
   all a, b, c : Stereotype | all match: DefaultMatchStrategy |
   stereotypeOSMergeRule[a, b, c, match] =>
   stereotypeOSMergeRule[b, a, c, match]
}

assert classOSMergeRuleCommutativePartial {
   all a, b, c : Class | all match: PartialMatchStrategy |
   classOSMergeRule[a, b, c, match] =>
   classOSMergeRule[b, a, c, match]
}

assert stereotypeOSMergeRuleCommutativePartial {
   all a, b, c : Stereotype | all match: PartialMatchStrategy |
   stereotypeOSMergeRule[a, b, c, match] =>
   stereotypeOSMergeRule[b, a, c, match]
}

assert stereotypeOSMergeRuleCommutativeComplete {
   all a, b, c : Stereotype | all match: CompleteMatchStrategy |
   stereotypeOSMergeRule[a, b, c, match] =>
   stereotypeOSMergeRule[b, a, c, match]
}

assert stereotypeOSMergeRuleIsAssociativeDefaultStrategy { all
  receiving, merged1, merged2, resulting, intermediate1,
  intermediate2: Stereotype | all match: DefaultMatchStrategy |
  stereotypeOSMergeRule[receiving, merged1, intermediate1, match] &&
  stereotypeOSMergeRule[intermediate1, merged2, resulting, match] &&
  stereotypeOSMergeRule[receiving, merged2, intermediate2, match] =>
  stereotypeOSMergeRule[intermediate2, merged1, resulting, match]
}

assert stereotypeOSMergeRuleIsAssociativePartialStrategy {
  all receiving, merged1, merged2, resulting, intermediate1,
  intermediate2: Stereotype | all match: PartialMatchStrategy |
```

```
  stereotypeOSMergeRule[receiving, merged1, intermediate1, match] &&
  stereotypeOSMergeRule[intermediate1, merged2, resulting, match] &&
  stereotypeOSMergeRule[receiving, merged2, intermediate2, match] =>
  stereotypeOSMergeRule[intermediate2, merged1, resulting, match]
}

assert stereotypeOSMergeRuleIsAssociativeCompleteStrategy {
  all receiving, merged1, merged2, resulting, intermediate1,
  intermediate2: Stereotype | all match: CompleteMatchStrategy |
  stereotypeOSMergeRule[receiving, merged1, intermediate1, match] &&
  stereotypeOSMergeRule[intermediate1, merged2, resulting, match] &&
  stereotypeOSMergeRule[receiving, merged2, intermediate2, match] =>
  stereotypeOSMergeRule[intermediate2, merged1, resulting, match]
}

assert stereotypeOSMergeRuleIsUniqueDefault {
  all receiving, merged, resultingA, resultingB: Stereotype |
  all match: DefaultMatchStrategy |
  stereotypeOSMergeRule[receiving, merged, resultingA, match] &&
  stereotypeOSMergeRule[receiving, merged, resultingB, match] =>
  stereotypeDefaultMatchRule[resultingA, resultingB]
}

assert stereotypeOSMergeRuleIsUniquePartial{
  all receiving, merged, resultingA, resultingB: Stereotype |
  all match: PartialMatchStrategy |
  stereotypeOSMergeRule[receiving, merged, resultingA, match] &&
  stereotypeOSMergeRule[receiving, merged, resultingB, match] =>
  stereotypeDefaultMatchRule[resultingA, resultingB]
 }

assert stereotypeOSMergeRuleIsUniqueComplete{
  all receiving, merged, resultingA, resultingB: Stereotype |
  all match: CompleteMatchStrategy |
  stereotypeOSMergeRule[receiving, merged, resultingA, match] &&
  stereotypeOSMergeRule[receiving, merged, resultingB, match] =>
  stereotypeDefaultMatchRule[resultingA, resultingB]
 }
```

```
assert stereotypeMSMergeRuleIdempotencyDefault {
   all a, c : Stereotype | all match: DefaultMatchStrategy |
   stereotypeMSMergeRule[a, a, c, match] =>
   stereotypeMSMergeRule[a, a, c, match]
}

assert stereotypeMSMergeRuleIdempotencyPartial {
   all a, c : Stereotype | all match: PartialMatchStrategy |
   stereotypeMSMergeRule[a, a, c, match] =>
   stereotypeMSMergeRule[a, a, c, match]
}

assert stereotypeMSMergeRuleIdempotencyComplete {
   all a, c : Stereotype | all match: CompleteMatchStrategy |
   stereotypeMSMergeRule[a, a, c, match] =>
   stereotypeMSMergeRule[a, a, c, match]
}

assert stereotypeMSMergeRuleCommutativeDefault {
   all a, b, c : Stereotype | all match: DefaultMatchStrategy |
   stereotypeMSMergeRule[a, b, c, match] =>
     stereotypeMSMergeRule[b, a, c, match]
}

assert stereotypeMSMergeRuleCommutativePartial {
   all a, b, c : Stereotype | all match: PartialMatchStrategy |
   stereotypeMSMergeRule[a, b, c, match] =>
    stereotypeMSMergeRule[b, a, c, match]
}

assert stereotypeMSMergeRuleCommutativeComplete {
   all a, b, c : Stereotype | all match: CompleteMatchStrategy |
   stereotypeMSMergeRule[a, b, c, match] =>
     stereotypeMSMergeRule[b, a, c, match]
}

assert stereotypeMSMergeRuleIsAssociativeDefaultStrategy {
  all receiving, merged1, merged2, resulting, intermediate1,
  intermediate2: Stereotype | all match: DefaultMatchStrategy |
```

```
  stereotypeMSMergeRule[receiving, merged1, intermediate1, match] &&
  stereotypeMSMergeRule[intermediate1, merged2, resulting, match] &&
  stereotypeMSMergeRule[receiving, merged2, intermediate2, match] =>
  stereotypeMSMergeRule[intermediate2, merged1, resulting, match] }

assert stereotypeMSMergeRuleIsAssociativePartialStrategy {
  all receiving, merged1, merged2, resulting, intermediate1,
  intermediate2: Stereotype | all match: PartialMatchStrategy |
  stereotypeMSMergeRule[receiving, merged1, intermediate1, match] &&
  stereotypeMSMergeRule[intermediate1, merged2, resulting, match] &&
  stereotypeMSMergeRule[receiving, merged2, intermediate2, match] =>
  stereotypeMSMergeRule[intermediate2, merged1, resulting, match]
}

assert stereotypeMSMergeRuleIsAssociativeCompleteStrategy {
  all receiving, merged1, merged2, resulting, intermediate1,
  intermediate2: Stereotype | all match: CompleteMatchStrategy |
  stereotypeMSMergeRule[receiving, merged1, intermediate1, match] &&
  stereotypeMSMergeRule[intermediate1, merged2, resulting, match] &&
  stereotypeMSMergeRule[receiving, merged2, intermediate2, match] =>
  stereotypeMSMergeRule[intermediate2, merged1, resulting, match]
}

assert stereotypeMSMergeRuleIsUniqueDefault {
  all receiving, merged, resultingA, resultingB: Stereotype |
  all match: DefaultMatchStrategy |
  stereotypeMSMergeRule[receiving, merged, resultingA, match] &&
  stereotypeMSMergeRule[receiving, merged, resultingB, match] =>
  stereotypeDefaultMatchRule[resultingA, resultingB]
}

assert stereotypeMSMergeRuleIsUniquePartial {
  all receiving, merged, resultingA, resultingB: Stereotype |
  all match: PartialMatchStrategy |
  stereotypeMSMergeRule[receiving, merged, resultingA, match] &&
  stereotypeMSMergeRule[receiving, merged, resultingB, match] =>
  stereotypePartialMatchRule[resultingA, resultingB]
}
```

```
assert stereotypeMSMergeRuleIsUniqueComplete {
  all receiving, merged, resultingA, resultingB: Stereotype |
  all match: CompleteMatchStrategy |
  stereotypeMSMergeRule[receiving, merged, resultingA, match] &&
  stereotypeMSMergeRule[receiving, merged, resultingB, match] =>
  stereotypeCompleteMatchRule[resultingA, resultingB]
}

assert  profileUSMergeRuleIsUniqueDefault {
  all receiving, merged, resultingA, resultingB: Profile |
  all match: DefaultMatchStrategy |
  profileUSMergeRule[receiving, merged, resultingA, match] &&
  profileUSMergeRule[receiving, merged, resultingB, match] =>
  profileDefaultMatchRule[resultingA, resultingB]
}

assert  profileUSMergeRuleIsUniquePartial {
  all receiving, merged, resultingA, resultingB: Profile |
  all match: PartialMatchStrategy |
  profileUSMergeRule[receiving, merged, resultingA, match] &&
  profileUSMergeRule[receiving, merged, resultingB, match] =>
  profileDefaultMatchRule[resultingA, resultingB]
}

assert  profileUSMergeRuleIsUniqueComplete {
  all receiving, merged, resultingA, resultingB: Profile |
  all match: CompleteMatchStrategy |
  profileUSMergeRule[receiving, merged, resultingA, match] &&
  profileUSMergeRule[receiving, merged, resultingB, match] =>
  profileDefaultMatchRule[resultingA, resultingB]
}

assert mergeOperatorIdempotency {
  all a, c : Profile | all match: DefaultMatchStrategy |
  all strategy: OverrideStrategy |
  mergeOperator[a, a, c, match, strategy] =>
  mergeOperator[a, a, c, match, strategy]
}
```

```
assert mergeOperatorCommutativeDefault {
   all a, b, c : Profile | all match: DefaultMatchStrategy |
   all strategy: OverrideStrategy |
   mergeOperator[a, b, c, match, strategy] =>
   mergeOperator[b, a, c, match, strategy
}

assert mergeOperatorCommutativePartial {
   all a, b, c : Profile | all match: PartialMatchStrategy |
     all strategy: OverrideStrategy |
   mergeOperator[a, b, c, match, strategy] =>
   mergeOperator[b, a, c, match, strategy]
}

assert mergeOperatorCommutativeComplete {
  all a, b, c : Profile | all match: CompleteMatchStrategy |
  all strategy: OverrideStrategy |
  mergeOperator[a, b, c, match, strategy] =>
  mergeOperator[b, a, c, match, strategy]
}

assert mergeOperatorIsAssociativeDefaultStrategy {
  all receiving, merged1, merged2, resulting,
    intermediate1, intermediate2: Profile |
  all match: DefaultMatchStrategy | all strategy: OverrideStrategy |
  mergeOperator[receiving, merged1, intermediate1,
     match, strategy] &&
  mergeOperator[intermediate1, merged2, resulting,
     match, strategy] &&
  mergeOperator[receiving, merged2, intermediate2,
     match, strategy] =>
  mergeOperator[intermediate2, merged1, resulting,
     match, strategy]
}

assert  mergeOperatorIsUniqueDefault {
  all receiving, merged, resultingA, resultingB: Profile |
  all match: DefaultMatchStrategy | all strategy: OverrideStrategy |
  mergeOperator[receiving, merged, resultingA, match, strategy] &&
```

```
  mergeOperator[receiving, merged, resultingB, match, strategy] =>
  profileDefaultMatchRule[resultingA, resultingB]
}

assert  mergeOperatorIsUniqueComplete {
  all receiving, merged, resultingA, resultingB: Profile |
  all match: CompleteMatchStrategy | all strategy: OverrideStrategy |
  mergeOperator[receiving, merged, resultingA, match, strategy] &&
  mergeOperator[receiving, merged, resultingB, match, strategy] =>
  profileDefaultMatchRule[resultingA, resultingB]
}

assert mergeOperatorIdempotencyDefaultMergeStrategy {
   all a, c : Profile | all match: DefaultMatchStrategy |
   all strategy: MergeStrategy |
   mergeOperator[a, a, c, match, strategy] =>
   mergeOperator[a, a, c, match, strategy]
}

assert mergeOperatorIdempotencyPartialMergeStrategy {
   all a, c : Profile | all match: PartialMatchStrategy |
   all strategy: MergeStrategy |
   mergeOperator[a, a, c, match, strategy] =>
   mergeOperator[a, a, c, match, strategy]
}

assert mergeOperatorIdempotencyCompleteMergeStrategy {
   all a, c : Profile | all match: CompleteMatchStrategy |
   all strategy: MergeStrategy |
   mergeOperator[a, a, c, match, strategy] =>
   mergeOperator[a, a, c, match, strategy]
}

assert mergeOperatorCommutativeDefaultMerge {
   all a, b, c : Profile | all match: DefaultMatchStrategy |
   all strategy: MergeStrategy |
   mergeOperator[a, b, c, match, strategy] =>
   mergeOperator[b, a, c, match, strategy]
}
```

```
assert mergeOperatorCommutativePartialMerge {
   all a, b, c : Profile | all match: PartialMatchStrategy |
   all strategy: MergeStrategy |
   mergeOperator[a, b, c, match, strategy] =>
   mergeOperator[b, a, c, match, strategy]
}

assert mergeOperatorCommutativeCompleteMerge {
   all a, b, c : Profile | all match: CompleteMatchStrategy |
   all strategy: MergeStrategy |
   mergeOperator[a, b, c, match, strategy] =>
   mergeOperator[b, a, c, match, strategy]
}

assert mergeOperatorIsAssociativeDefaultMatchMergeStrategy {
  all receiving, merged1, merged2, resulting, intermediate1,
    intermediate2: Profile | all match: DefaultMatchStrategy |
  all strategy: MergeStrategy |
  mergeOperator[receiving, merged1, intermediate1,
    match, strategy] &&
  mergeOperator[intermediate1, merged2, resulting,
    match, strategy] &&
  mergeOperator[receiving, merged2, intermediate2,
    match, strategy] =>
  mergeOperator[intermediate2, merged1, resulting,
    match, strategy]
}

assert mergeOperatorIsAssociativeCompleteMatchMergeStrategy {
  all receiving, merged1, merged2, resulting, intermediate1,
    intermediate2: Profile |
  all match: CompleteMatchStrategy | all strategy: MergeStrategy |
  mergeOperator[receiving, merged1, intermediate1,
    match, strategy] &&
  mergeOperator[intermediate1, merged2, resulting,
    match, strategy] &&
  mergeOperator[receiving, merged2, intermediate2,
    match, strategy] =>
```

```
  mergeOperator[intermediate2, merged1, resulting,
    match, strategy]
}

assert  mergeOperatorIsUniqueDefaultMatchMergeStrategy {
  all receiving, merged, resultingA, resultingB: Profile |
  all match: DefaultMatchStrategy | all strategy: MergeStrategy |
  mergeOperator[receiving, merged, resultingA, match, strategy] &&
  mergeOperator[receiving, merged, resultingB, match, strategy] =>
  profileDefaultMatchRule[resultingA, resultingB]
}

assert  mergeOperatorIsUniqueCompleteMatchMergeStrategy {
  all receiving, merged, resultingA, resultingB: Profile |
  all match: CompleteMatchStrategy | all strategy: MergeStrategy |
  mergeOperator[receiving, merged, resultingA, match, strategy] &&
  mergeOperator[receiving, merged, resultingB, match, strategy] =>
  profileDefaultMatchRule[resultingA, resultingB]
}

assert parameterOSMergeRuleIsUnique {
  all receiving, merged, resultingA, resultingB: Parameter |
  one match: DefaultMatchStrategy |
  parameterOSMergeRule[receiving, merged, resultingA, match] &&
  parameterOSMergeRule[receiving, merged, resultingB, match] =>
  parameterDefaultMatchRule[resultingA, resultingB]
}

assert imageOSMergeRuleIsUnique {
  all receiving, merged, resultingA, resultingB: Image |
  one match: DefaultMatchStrategy |
  imageOSMergeRule[receiving, merged, resultingA, match] &&
  imageOSMergeRule[receiving, merged, resultingB, match] =>
  imageDefaultMatchRule[resultingA, resultingB]
 }

assert imageMSMergeRuleIdempotencyDefault {
   all a, c : Image | all match: DefaultMatchStrategy |
   imageMSMergeRule[a, a, c, match] =>
```

```
   imageMSMergeRule[a, a, c, match]
}

assert imageMSMergeRuleIdempotencyPartial {
   all a, c : Image | all match: PartialMatchStrategy |
   imageMSMergeRule[a, a, c, match] =>
   imageMSMergeRule[a, a, c, match]
}

assert imageMSMergeRuleIdempotencyComplete {
   all a, c : Image | all match: CompleteMatchStrategy |
   imageMSMergeRule[a, a, c, match] =>
   imageMSMergeRule[a, a, c, match]
}

assert imageMSMergeRuleCommutativeDefault {
   all a, b, c : Image | all match: DefaultMatchStrategy |
   imageMSMergeRule[a, b, c, match] =>
   imageMSMergeRule[b, a, c, match]
}
```

Apêndice D

Histórico de Trabalhos

Durante a elaboração do trabalho apresentado nesta dissertação procurou-se elaborar alguns artigos com o objetivo de amadurecer a abordagem e ter um *feedback* da comunidade científica. Este artigos foram publicados em ou submetidos para, congressos nacionais e internacionais. Nestes artigos buscou-se relatar a evolução e maturidade da abordagem. Sendo assim, estes artigos são citados a seguir:

Artigo 1 - Março / 2007
Evento: IEEE Symposium on Visual Languages and Human-Centric Computing
Título: On Model Composition
Resumo: Este artigo apresentou um levantamento das necessidades de compor modelos e dos problemas encontrados nesta atividade.

Artigo 2 - Abril / 2007
Evento: Second International Conference on Software Engineering Advances (ICSEA'07)
Título: A Guidance For Model Composition
Resumo: Neste trabalho foi especificado o guia de composição em uma versão inicial juntamente com as regras de comparação e transformação.

Artigo 3 - Maio / 2007
Evento: Workshop de Tese e Dissertação em Engenharia de Software (SBES'07)
Título: Composição de UML *Profiles*
Resumo: Neste artigo foi especificado um *overview* da abordagem com os operadores, a extensão do metamodelo, a verificação formal e um visão inicial da ferramenta.

Artigo 4 - Setembro/2007
Evento: 30th ACM/IEEE International Conference on Software Engineering (ICSE'08)
Título: Composition of UML *Profiles*
Resumo: O artigo submetido a este evento teve na sua essência a definição dos operadores de composição. Além disso, as regras de transformação juntamente com as regras de comparação foram também especificadas.

Artigo 5 - Novembro/2007

Evento: 23rd Annual ACM Symposium on Applied Computing (SAC'08)

Título: Checking Composition of UML Profile with Alloy

Resumo: Este artigo apresenta a versão inicial da modelagem formal do metamodelo dos *profiles* e do mecanismo de composição.

Artigo 6 - Dezembro/2007

Evento: 20th International Conference on Advanced Information Systems Engineering (CAiSE'08)

Título: Modeling Composition of UML Profile with Alloy

Resumo: Este artigo apresenta a verificação formal do mecanismo de composição em sua versão final. Nele foram descritos os resultados, as limitações e as contribuições referente a definição de uma semântica formal para o mecanismo de composição.