UNIVERSIDADE FEDERAL DE PERNAMBUCO – UFPE
CENTRO DE INFORMÁTICA – CIN

Trabalho de Graduação – Ciência da Computação

Evolução Automatizada de Modelos Arquiteturais Concorrentes

Camila Sá da Fonseca

Recife, 14 de dezembro de 2010

Universidade Federal de Pernambuco

Centro de Informática

Camila Sá da Fonseca

Evolução Automatizada de Modelos Arquiteturais Concorrentes

Monografia apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Augusto Cezar Alves Sampaio

Recife, 14 de dezembro de 2010

Aos meus pais, Cremildo e Kenia

Agradecimentos

“A satisfação está no esforço e não apenas na realização final”.

Mahatma Gandhi

Agradeço primeiramente a Deus, sem Ele eu não poderia ter conseguido conquistar mais um marco tão importante na minha vida. Ele, que está sempre presente não só como consolo nos momentos de dificuldades, mas como a representação da fé que me faz buscar os objetivos de maneira íntegra e digna.

Aos meus pais, Cremildo e Kenia, que dedicaram tanto de suas vidas para a minha educação, e que, por meio desse trabalho, poderão ver a filha formada. Além do apoio sempre constante e da flexibilidade em aceitar as negações de momentos de lazer por conta das atividades do CIn, agradeço pelo carinho e pelas palavras de sabedoria.

Ao meu irmão, Josué, que cedeu seu quarto com muita boa vontade, pois só um computador não foi suficiente!

A Felipe, meu namorado, que suportou os estresses, especialmente deste último período, com muita compreensão. Tentou, diversas vezes me convencer a me distrair um pouco com pretexto de que acabaria sendo mais produtivo; e, muitas vezes, teve não como resposta! Mesmo assim, sempre me apoiou e escutou as dificuldades – mesmo sem entender – e acabou ajudando, porque a explicação nos faz compreender melhor até o próprio problema.

Aos meus amigos e familiares, que muitas vezes ajudaram até sem perceber com palavras de conforto e momentos de diversão. Em especial, aos amigos do CIn são uma das coisas que eu levarei para sempre comigo: o grupo coeso: Bruno, Emanuel, Guilherme e Luís; que depois virou Synergy e aproximou outros, em especial: Caio, Lailson, Rafael; e os meninos de Engenharia. Podem ter a certeza que uma das maiores dificuldades deste trabalho é por não ser realizado em equipe e por não favorecer a troca de conhecimento entre um grupo específico; o apoio de vocês ao longo de todas as disciplinas foi indiscutível, fundamental para todo o aprendizado e o que me deu coragem para continuar.

Ao professor, Augusto Sampaio, a quem admiro não só pela inteligência, mas pelo esforço, disponibilidade e interesse em querer ajudar; agradeço pela orientação, pela compreensão e pela paciência.

A José Dihego, que teve um papel fundamental me introduzindo o tema abordado.

Não sei nem se o trabalho tem tanto conteúdo para eu dedicar a tantas pessoas, mas cada uma delas teve um papel único e indispensável e eu não poderia deixar de prestar os meus sinceros agradecimentos.

Resumo

Este trabalho é baseado na necessidade de automatização de transformações de modelos no contexto de MDA (*Model Driven Architecture*). Assim, explora os conceitos dessa metodologia de desenvolvimento, juntamente com uma análise de sistemas distribuídos, pois as transformações são baseadas no *profile* UML-RT, uma extensão de UML com construções que suportam concorrência.

Para a metodologia de desenvolvimento de software MDA a busca por padrões arquiteturais é fundamental para verificar se um modelo pode ser transformado em outro, mantendo o comportamento, mas com uma estrutura mais adequada. Esse cenário executa refatorações (*refactorings*) no modelo e promove a evolução do sistema modelado.

Para mecanizar este tipo de refatoração, há algumas ferramentas disponíveis, em especial as linguagens de transformação. Para concretizar este trabalho será realizada a implementação de algumas das de Leis de Transformações para o profile UML-RT definidas formalmente por (Ramos, 2005).

A implementação se deu com a linguagem/ferramenta de transformação de programas Stratego/XT.

Palavras-chave: Transformação de Modelos, MDA, UML-RT, Stratego/XT, *refactoring*

Conteúdo

Lista de Figuras

Lista de Tabelas

1. Introdução

Contemporaneamente, o desenvolvimento de softwares mais elaborados é indispensável para suportar as demandas tanto de empresas como de usuários comuns que necessitam de sistemas complexos para melhor conduzir e controlar os negócios e para oferecer suporte a atividades pessoais.

O desenvolvimento de novas tecnologias pode exigir grandes investimentos, entretanto, o objetivo é que se consiga fazer mais com menos; ou seja, que se consiga produzir sistemas mais ricos com uma menor quantidade de recursos (tempo, financeiro e humano), exigindo, assim, um aumento da produtividade ao longo do desenvolvimento.

Acompanhando a tendência da elevação da complexidade dos programas, estão os Sistemas Distribuídos (ou SD). Segundo (Guimarães, 2008), o espectro da utilização de tecnologias emergentes é altamente extenso e, em conjunto com modernas infra-estruturas de redes de computadores, têm possibilitado o desenvolvimento de sofisticadas aplicações que exploram as potencialidades e os benefícios de ambientes computacionais de natureza distribuída.

Os sistemas distribuídos são inerentemente complexos porque exigem elevado grau de interoperabilidade muitas vezes entre ambientes completamente distintos. Entretanto, proporcionam diversas vantagens como elevação do desempenho, afinal as funcionalidades estão distribuídas; compartilhamento de recursos; maior tolerância ao erro e elevada escalabilidade.

Para acompanhar essa necessidade por sistemas mais robustos, a área da engenharia de software tem evoluído bastante, contudo ainda enfrenta uma série de desafios; a constante necessidade de adaptação de um sistema às novas tecnologias e às mudanças em seus requisitos são exemplos dessas dificuldades. Numa perspectiva mais ampla, os principais problemas enfrentados são: baixa produtividade, dificuldade na implementação de sistemas com portabilidade, dificuldade para manutenção, falta de interoperabilidade e documentação inconsistente.

Esses problemas ocorrem porque arquiteturas de software são modificadas no decorrer do tempo à medida que os sistemas são construídos. Isto pode fazer com que a arquitetura conceitual (ou seja, a arquitetura planejada) comece a divergir da arquitetura emergente (que representa o que se encontra efetivamente implementado em código fonte) (Schots, Marcelo. Murta, Leonardo. Werner, Cláudia., 2009).

Para tentar atender a esta demanda por sistemas mais complexos - e os sistemas concorrentes e distribuídos enquadram-se nesse espectro de complexidade e são necessários para boa parte dos sistemas a serem criados - é necessário buscar - e encontrar! - alternativas que consigam reduzir os recursos para o desenvolvimento desses sistemas.

Neste cenário novas metodologias surgem para tentar solucionar os problemas. Em geral, as melhorias estão relacionadas ao aumento do nível de abstração necessário para projetar e implementar o software.

Uma das técnicas para otimizar o processo de desenvolvimento está relacionada à criação e à evolução de modelos do sistema, é o que se chama de Engenharia Dirigida por Modelos (MDE – *Model Driven Engineering*). Estudar este contexto é fundamental para compreender uma das metodologias mais difundidas atualmente.

Na MDE os modelos – elemento que ajudam as pessoas a compreender e comunicar idéias complexas, (Brown, 2004) – são a base para a implementação e evolução dos softwares. Diversos engenhos apóiam a automação das transformações destes modelos. Com o objetivo de oferecer mais flexibilidade à manutenção dos processos de software que usam a abordagem MDE, linguagens para programar regras de transformação foram propostas. (Pellegrini, F.; Silva, F.; Silva, B.; Maciel, S., 2010)

A linguagem para construção de modelos de software mais difundida é a UML (*Unified Modeling Language*), que é mantida pela OMG (*Object Management Group*). Essa mesma organização mantém a instância de MDE com maior aceitação, é a MDA (*Model Driven Architecture*): um framework conceitual que separa decisões orientadas ao negócio de decisões de plataforma, permitindo uma grande flexibilidade no desenvolvimento da arquitetura e na evolução desses sistemas (Brown, 2004).

A MDA é uma forma de organizar e gerenciar arquiteturas com o suporte de ferramentas automáticas e serviços para a definição de modelos e para facilitar transformações entre diferentes tipos de modelo.

A linguagem UML é consenso no desenvolvimento de software, sendo suportada pela maioria das ferramentas de modelagem, tanto proprietárias como *Open Source*.

A criação, o armazenamento e a transformação de modelos por meio dessas ferramentas colocam a modelagem no centro do processo de produção de software e forma a base da MDA (Watson, 2010).

Essa separação entre negócio e tecnologia é possível devido a uma hierarquização – de acordo com o nível de abstração – nos modelos na MDA: CIM (Modelo Independente de Computação); PIM (Modelo Independente de Plataforma); e PSM (Modelo Específico de Plataforma).

De maneira bem objetiva (Flore, 2003) afirma que sem a transformação automática entre modelos, o esforço para transformá-los manualmente e proibitivo; assim, as transformações automáticas são um conceito chave na MDA.

As transformações podem ocorrer entre modelos de níveis diferentes, mas também ocorrem para modelos em um mesmo nível; essas transformações são conhecidas como *refactoring* (refatoramento) e deixam um modelo com um mesmo comportamento mais uma estrutura mais adequada.

Segundo (Brown, 2004) as diferenças entre os diferentes tipos de modelos nos permitem pensar em software e desenvolvimento de sistemas como uma série de refinamentos entre representações diferentes do modelo.

Diante do exposto, percebe-se a importância que as transformações exercem quando se trata de modelagem de sistemas, e, consequentemente, do desenvolvimento de sistemas.

Essa ideia de transformações de modelos tem raízes na representação formal de sistemas - os métodos formais. Atualmente há diversas abordagens para execução de transformações, entretanto a maioria delas apresenta limitações à interoperabilidade com ambientes de desenvolvimento, apoio à definição e execução de diversos tipos de transformações e à possibilidade de expansão de suas capacidades (Lunelli, V. C. ; Bacelo, A. T. , 2009), além de geralmente seguir a tradição de formalismos.

Com as transformações é possível que se passe mais tempo compreendendo os problemas do usuário no domínio e no desenho de soluções que conceituem como será o comportamento do sistema para satisfazer as necessidades; ou seja, a elaboração do PIM é o foco do desenvolvimento, porque a passagem para PSM deve ser automatizada.

Este trabalho visa implementar um subconjunto de transformações expressas em formalismos (Ramos, 2005) dentro da evolução dos sistemas e explora a possibilidade de permitir as transformações confiáveis de forma mecanizada para os desenvolvedores.

Será investigada a possibilidade de – com uma nova ferramenta e linguagem: Stratego/XT – efetivar a automação de refatorações arquiteturais de modelos de sistemas baseados em componentes, modelados em UML-RT, um profile de UML com suporte a componentes ativos. Mostrando-se possível para parte das transformações, poder-se-á estender para as outras e atividades comuns de modelagem poderão ser simplificadas, facilitando e agilizando a evolução dos sistemas distribuídos no contexto de MDA.

Diante desse cenário, buscar ferramentas que ofereçam suporte à criação e à manutenção (evolução) de modelos de sistemas concorrentes e distribuídos é duplamente relevante: MDA ainda não é totalmente maturada e necessita de definição de transformações e o contexto de UML-RT é ainda menos explorado que a UML tradicional. Além disso, a tentativa de utilizar uma ferramenta nova, como Stratego/XT, para implementar algumas das transformações de maneira automatizada é desafiadora e recompensadora.

O trabalho está dividido em cinco principais partes: primeiro uma breve abordagem aos sistemas distribuídos é feita para que se reconheçam algumas das características e a relevância que o tema possui; a MDA é tratada de maneira detalhada já incluindo alguns pontos particulares deste trabalho; o *profile* UML-RT é apresentado com suas especificidades, essa seção inclui, ainda, importantes padrões definidos pela OMG como o XMI e o conceito de Metamodelagem; a ferramenta, Stratego/XT é apresentada juntamente com dados relacionados diretamente a este trabalho, como a metodologia criada; um exemplo prático é mostrado, juntamente com o delineamento de algumas limitações do desenvolvimento; e por fim, as conclusões são apresentadas, incluindo trabalhos relacionados e trabalhos futuros.

2. Sistemas Concorrentes e Distribuídos

Atualmente, sistemas de software estão presentes em qualquer negócio e no próprio dia-a-dia das pessoas. Além disso, a disseminação do uso da rede mundial de computadores ao mesmo tempo em que intensifica esse processo, torna-se indispensável

por promover a troca de informações entre esses computadores. Mas, por trás dessa sistemática de comunicação, há sempre um ou mais programas - sistemas - que coordenam essa comunicação.

Com o avanço tecnológico tanto na capacidade de processamento como nas redes de computadores, fica mais fácil e, muitas vezes, mais eficiente agrupar um grande número de CPUs conectadas por meio de uma rede trabalhando em um sistema distribuído.

Segundo (Coulouris, G.; Dollimore, J.; Kindberg, T., 1994), “um sistema distribuído é uma coleção de computadores autônomos conectados por uma rede e equipados com um sistema de software distribuído". Por essa afirmação percebemos que para a existência de um sistema distribuído (também chamados de SD) são necessários tanto computadores - ou processadores -, como uma rede e o próprio software que possua componentes em execução em cada um dos computadores pertencentes ao sistema. Segundo, (Andrews, 1991), entretanto, a programação concorrente é a atividade de construção de um programa contendo múltiplos processos que cooperam na realização de tarefas. Afirma ainda que um programa distribuído é um programa concorrente no qual os processos se comunicam através da troca de mensagens. Pode-se notar que nesta definição, o autor vincula o termo distribuído ao sistema de comunicação em nível de programação; ou seja, há uma abstração dos componentes de hardware em um eventual sistema distribuído.

No escopo desse trabalho, não há preocupação com os componentes de hardware, ou seja, não há distinção entre um sistema distribuído que tem suas partes executadas por um mesmo processador e outro que é fisicamente distribuído; isso porque para a modelagem – os modelos tratados neste trabalho, porque outros diagramas de UML podem incluir aspectos relacionados ao hardware, como os diagramas de processo – esses dois casos seriam representados da mesma maneira. Assim, o interesse deste projeto é em sistemas que possuam partes independentes e ativas.

Acrescentando à definição acima, (Tanenbaum, 1995) afirma que “um sistema distribuído é uma coleção de computadores independentes que aparentam ao usuário ser um computador único", nota-se uma importante característica dos sistemas distribuídos, a transparência. Muitas vezes as pessoas estão usando esse tipo de sistemas mesmo sem ter conhecimento deste fato. O software para sistemas distribuídos é completamente diferente do software para sistemas centralizados tendem, inclusive, a ser extremamente grandes e complexos (Selic, Bran; Rumbaugh, Jim, 1998).

Alguns exemplos de sistemas distribuídos são: computadores e PDAs em um estoque de uma companhia que estejam rodando o mesmo sistema de controle; um grande banco com muitas agências, cada qual com um computador e caixas automáticas; uma rede de estações de trabalho em uma universidade ou companhia; computadores de bancos que trocam diversas informações constantemente e, atualmente, pode-se destacar a computação em nuvem (*cloud computing)*, entre muitas outras aplicações distribuídas.

O crescimento do uso de sistemas distribuídos ocorre especialmente porque, para certas aplicações, essa modalidade consegue obter uma capacidade de processamento muito superior à oferecida por um sistema centralizado. Além disso, um sistema distribuído é escalável e pode reduzir os riscos caso haja falha em alguns dos processadores envolvidos, por não dependerem exclusivamente de um computador. Consequentemente, há maior confiabilidade nos dados e disponibilidade de informações. Os diversos usuários também podem, por meio dos SD, compartilhar recursos não só de software, mas também de hardware.

A concorrência em sistemas de software significa que há mais de um processo em execução a cada instante, assim está inerentemente presente nos sistemas distribuídos; afinal cada computador operando no sistema pode realizar algumas atividades independentemente. Há situações em que há o acesso concorrente aos recursos compartilhados, diante disso deve haver sincronização para controlar esses acessos. Assim, temos que um sistema de software distribuído é composto por instruções modularizadas em processos que são executadas concorrente ou paralelamente por um ou mais processadores.

Apesar dos benefícios expostos, segundo (Guimarães, 2008) o desenvolvimento de sistemas em ambientes distribuídos é muito complexo, afinal é preciso tratar das heterogeneidades das plataformas, sistemas operacionais e linguagens de programação; concorrência; comunicação entre os elementos distribuídos; segurança e qualidade de serviço. Além da necessidade de promover a transparência: esconder os detalhes inerentes à distribuição do processamento, tais como a localização geográfica dos elementos funcionais, replicação de dados e funcionalidades, além de ocorrência de falhas principalmente resultantes de problemas clássicos como *deadlock* e *livelock*. Exposto esse cenário fica clara a importância da comunicação entre os processos - entre as máquinas - para que o sistema funcione corretamente. De acordo com (Selic, Bran; Rumbaugh, Jim, 1998) é crucial para sistemas desse tipo que o software seja projetado com uma arquitetura segura, pois ela não apenas simplifica a construção inicial do sistema, como, de maneira mais importante, está apta a de adaptar as mudanças provocadas pela corrente de novos requisitos.

Atualmente, formalismos e técnicas de análise (verificação) de modelos conseguem oferecer suporte mecânico à verificação. Assim sendo, garantir que programas concorrentes satisfaçam certas propriedades desejáveis não tem sido um desafio na área de engenharia de software. Nossa abordagem é baseada na definição e evolução de modelos de forma construtiva, garantindo que os modelos evoluídos preservem o comportamento dos modelos iniciais e mais abstratos.

3. MDA

3.1. Modelagem

Modelos têm direcionado as disciplinas de engenharia por séculos e todas as engenharias dependem de modelos para promoção do entendimento de sistemas

complexos e do mundo real (Eriksson, Han-Eril. Penker, Magnus. Lyons, Brian. Fado, David., 2004), (Brown, 2004). A Engenharia Civil, por exemplo, não inicia um projeto sem antes ter um modelo daquilo que será construído, muitas vezes além da planta, existe a maquete, arquivos digitais 3D, entre outros. Para a Engenharia de Software não deve ser diferente e os modelos têm se mostrado fundamentais para permitir a pré-visualização do que será desenvolvido e representar a especificação e documentação do software.

A definição mais ampla alega que um modelo é uma visão simplificada da realidade (Selic, 2003) e, mais formalmente, segundo (Seidewitz, 2003), um modelo é um conjunto de demonstrações sobre um sistema em estudo. Além da definição (Selic, 2003) afirma que um modelo deve apresentar as seguintes características:

- Abstração (*Abstraction*): um modelo sempre é uma interpretação reduzida do sistema que representa;
- Compreensibilidade (*Understandability*): a notação de representação deve ser intuitivamente compreendida;
- Correção (*Accuracy*): um modelo deve ser uma representação real das características de interesse;
- Previsibilidade (*Predictiveness*): um modelo deve predizer corretamente as propriedades não-óbvias do sistema modelado;
- Inexpressividade de custo (Cost *Inexpressiveness*): um modelo deve ser significantemente menos custoso de modelar e analisar do que o sistema.

Outra característica, indispensável para esse projeto, é citada por (Eriksson, Han-Eril. Penker, Magnus. Lyons, Brian. Fado, David., 2004):

- Modificabilidade (*Modifiable*): Modelos devem ser fáceis de alterar e atualizar.

Os modelos especificam, ainda, como o projeto final deve ser e agir, comunicando a estrutura e o comportamento do sistema. Além disso, os modelos promovem um melhor entendimento do sistema; melhor visualização e controle da arquitetura; são bases para planos de implementação, de gerenciamento de riscos e de estimativas de custos e recursos. A razão para realizar a modelagem é justamente provir abstrações de um sistema físico, permitindo aos *stakeholders* se concentrarem em determinadas características, ignorando detalhes irrelevantes.

Segundo a OMG (*Object Management Group*) - organização que coordena padrões para aplicações orientadas a objetos - estruturar é uma forma de lidar com a complexidade dos sistemas e de promover o reuso de código. Afinal, a fase de análise e projeto do sistema é o momento mais fácil de estruturar uma aplicação como uma coleção de módulos e componentes.

No mundo da Engenharia de Software, a modelagem te uma rica tradição, datando dos primeiros dias de programação (Brown, 2004). Um modelo é escrito em alguma linguagem e descreverá um sistema, como mostra a Figura 1.

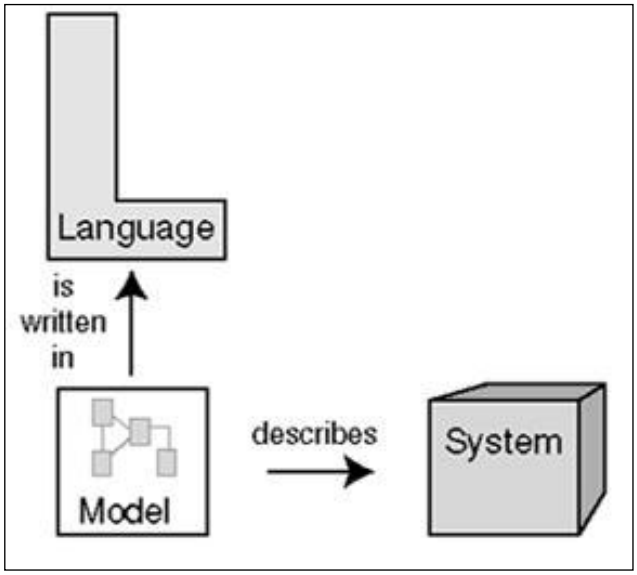


Figura 1 – Modelo, Linguagem e Sistema

Para permitir que um modelo possa ser lido, interpretado, alterado, persistido e, principalmente, para que mudanças automatizadas – as transformações – possam ser aplicadas neste modelo a linguagem de modelagem precisa ser bem definida: ter forma bem definida e significado (semântica) que possa ser interpretado por computadores (Warmer, Jos; Kleppe, Anneke; Bast, Wim., 2003).

Atualmente, a modelagem de software, em geral, utiliza algum tipo de notação gráfica – que representa os artefatos dos componentes utilizados e seus relacionamentos – e é suportada por alguma ferramenta CASE (*Computer-Aided Software Engineering*).

A modelagem permite que um mesmo sistema seja analisado sobre diversas óticas, essas diferentes representações de um sistema, cada uma sob uma diferente perspectiva, são denominadas de visões arquiteturais (Schots, Marcelo. Murta, Leonardo. Werner, Cláudia., 2009). Há várias abordagens para definir, descrever e identificar visões arquiteturais. Uma das possíveis formas para documentar tais visões é através da notação UML (*Unified Modeling Language*), voltada para sistemas orientados a objetos.

A UML foi desenvolvida, no final de 1994, por três dos principais nomes ligados às metodologias de desenvolvimento orientadas a objeto (Gradt Booch, James Rumbaugh e Ivar Jacobson., 2000). Segundo (Silva, 2001): "*A UML é uma linguagem para especificação, documentação, visualização e desenvolvimento de sistemas orientados a objetos. Sintetiza os principais métodos existentes, sendo considerada uma das linguagens mais expressivas para modelagem de sistemas orientados a objetos. Por meio de seus diagramas é possível representar sistemas de softwares sob diversas perspectivas de visualização*".

Em junho de 1995, o OMG (*Object Management Group*) iniciou o esforço para padronizar uma abordagem da modelagem de sistemas orientados a objetos e, em novembro de 1997, UML foi formalmente adotada como padrão pelo grupo (Tockey, 2001). Desde então a linguagem de modelagem tem sido mantida e atualizada – encontra-se na versão 2.0 – pelo OMG.

O surgimento da UML conseguiu padronizar a modelagem visual de software e aumentou dramaticamente o uso da modelagem, em 1995 as ferramentas de modelagem eram utilizadas por uma pequena fração de projetos de software (Watson, 2010); por volta de 2006 estimou que mais de 10 milhões de profissionais de TI usavam UML e por volta de 2008 mais do que 70% das organizações de desenvolvimento de software em todo o mundo estava utilizando a linguagem (Norton, 2006). Atualmente, a OMG afirma que a UML é a forma na qual o mundo modela não apenas a estrutura, o comportamento, e a arquitetura de um sistema, mas também os processos de negócio e a estrutura dos dados.

3.2. Características MDA

A expansão das infra-estruturas de computação empresariais para abranger novas plataformas e aplicações – para não mencionar a Internet – tem impulsionado a busca por maior portabilidade, interoperabilidade entre plataformas e independência de plataformas (Flore, 2003). Esse cenário, juntamente com a crescente demanda por sistemas mais elaborados, faz surgir novos princípios e paradigmas de desenvolvimento de software para tentar lidar com a complexidade inerente a estas aplicações. Neste contexto, encontra-se o MDE (Stahl, T.; Völter, M., 2006) o qual coloca o foco do desenvolvimento nos modelos. A iniciativa do MDE mais conhecida foi proposta pela OMG: MDA (MDA Guide, 2003).

A MDA foi iniciada em 1997, definida em 2000 e é constantemente melhorada. Seu objetivo é solucionar os problemas presentes em desenvolvimento de software tradicional (produtividade, portabilidade interoperabilidade, documentação, etc.) por meio da separação entre as especificações de um sistema e os detalhes de implementação (MDA Guide, 2003). Ela é “dirigida-a-modelos” porque o uso de modelos direciona o entendimento, o projeto, a construção, a implantação, a operação, a manutenção e a modificação dos sistemas (Brown, 2004). Assim, além das funções tradicionais, os modelos de software passam a ser usados como componentes principais ao longo do desenvolvimento.

A MDA proporciona uma abordagem que permite a especificação do sistema independentemente da plataforma[1] que irá suportá-lo. Após a seleção dessa plataforma, a metodologia prega a transformação – por meio de engenhos de transformações e meta-modelos – da especificação do sistema em uma plataforma específica. Assim, modelos com diferentes níveis de abstração são utilizados e para as transformações a automação é buscada. Um conjunto específico dessas transformações é a base deste trabalho.

Segundo (Warmer, Jos. Kleppe, Anneke., 2003) o que MDA traz para o ramo da computação é uma revolução do mesmo tipo da ocorrida no início de 1960 – a substituição da programação de linguagens *assembly* pelo uso de linguagem procedural. Eventualmente, modelos mais abstratos poderão ser compilados (i.e. transformados) em

[1]Plataforma é um conjunto de subsistemas e tecnologias que fornecem um conjunto de funcionalidades coerentes por meio de interfaces e padrões. Qualquer aplicação suportada por uma plataforma pode utilizar – sem preocupações com detalhes de funcionamento – as funcionalidades implementadas pela plataforma.

modelos de mais baixo-nível. Esses “compiladores” – ferramentas de transformações – ainda não serão tão eficientes por alguns anos. Os usuários precisarão fornecer dicas dobre como transformar partes de um modelo. A vantagem de se trabalhar em níveis mais altos de abstração vem se tornando progressivamente mais evidente na área de Engenharia de Software.

A MDA é um conceito que modifica a maneira que a organização projeta e desenvolve software através da separação da lógica de negócio da aplicação da infra-estrutura na qual ela é executada (Flore, 2003). A MDA auxilia os usuários de software a lidar com duas realidades primordiais no desenvolvimento de software: múltiplas tecnologias de implementação e a necessidade de manutenção ao longo de toda vida do sistema. Isso porque os principais objetivos da MDA são: portabilidade, interoperabilidade e reusabilidade por meio de uma separação arquitetural do que seja relevante para cada etapa do desenvolvimento.

Segundo (MDA Guide, 2003) quatro princípios descrevem a visão da OMG sobre a MDA:

- Modelos expressos em uma notação bem-definida são fundamentais para a compreensão de sistemas;
- A construção de sistemas pode ser organizada em volta de um conjunto de modelos por meio da aplicação de uma série de transformações entre modelos organizados em um framework arquitetural de camadas e transformações;
- A descrição com base formal num conjunto de metamodelos facilita a integração e transformação entre modelos;
- Aceitação e ampla adoção dessa abordagem baseada em modelo exigem padrões da indústria para oferecer abertura para os consumidores e fomentar a concorrência entre os fornecedores.

A MDA ajuda os usuários de software a lidar com duas realidades do atual ambiente de software: múltiplas tecnologias de desenvolvimento e a necessidade de manutenção ao longo do tempo de vida do sistema (Watson, 2010). A grande conquista da MDA é justamente permitir a integração de sistemas, alteração de plataformas e promover uma forma eficiente para as evoluções do sistema; aumentando, assim, a produtividade e a qualidade do software.

Para conquistar esses benefícios, a metodologia precisa proporcionar uma estrutura estável e ao mesmo tempo flexível, esses elementos serão apresentados nas próximas seções. É interessante observar, entretanto, que a MDA vem evoluindo com o passar do tempo e fatores fundamentais para a adoção prática da MDA, como operacionalização e definição de transformações de modelos ainda continuam em evidência entre os pesquisadores.

3.2.1. Padrões OMG e Arquitetura Quatro Camadas

A concepção de modelos no contexto MDA, como mencionado anteriormente, necessita de uma linguagem bem definida. Para suprir essa necessidade, as especificações dos modelos, em geral, possuem uma arquitetura de meta-dados, a qual

se compõe de, no mínimo, quatro camadas (Tenório, 2003). A Figura 2 representa esses níveis de representação.

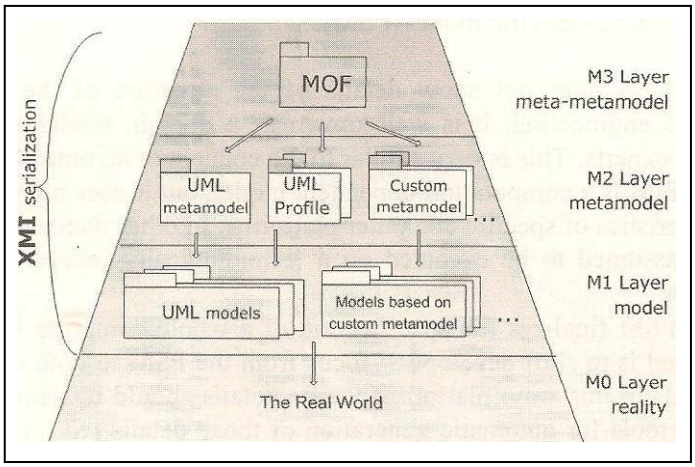


Figura 2 – Hierarquia da Metamodelagem

A MDA utiliza alguns padrões definidos pela OMG para representar artefatos de softwares:

- UML (*Unified Modeling Language*) – explicado na seção 3.1
- MOF (*Meta-Object Facility*) – define uma sintaxe abstrata para a especificação de meta-modelos (OMG, MOF 1.4 Specification, 2003). Este meta-metamodelo é composto por metaclasses, tipos de dados, exceções, constantes e restrições que serão utilizados na especificação dos metamodelos. Segundo (Gasevic, Gragan; Djuric, Dragan; Devedzic, Vladan, 2006) é uma linguagem abstrata e auto-definida e um framework para especificar, construir e manipular modelos tecnologicamente independentes. É a base para a definição de qualquer linguagem de modelagem como a UML.
- XMI (*XML (eXtensible Markup Language) Metadata Interchange*) – padrão que define mapeamentos de meta-metamodelos, metamodelos e modelos da MDA para documentos XML e Schemas XML. Como o XML é um padrão amplamente utilizado por diversas ferramentas de software, o XMI possibilita um intercâmbio de meta-metamodelos, metamodelos e modelos (Gasevic, Gragan; Djuric, Dragan; Devedzic, Vladan, 2006).

Retornando a Figura 2, pode-se observar que o nível mais elevado – M3 – possui o meta-metamodelo, neste caso o MOF. No nível M2 estão os metamodelos, padrões e metamodelos customizados definidos pelos usuários. Para o domínio OO, por exemplo há o Metamodelo UML e para o domínio *Data Warehouse* (DW) e *Business Intelligence* (BI), o Metamodelo CWM (*Common Warehouse Metamodel*) é utilizado. Um metamodelo, como o UML, pode, ainda, ser estendido para modelos adaptáveis a diferentes necessidades. Esse é o caso da linguagem UML-RT utilizada para o desenvolvimento das transformações neste trabalho. No nível M1 estão os modelos do mundo real representados por conceitos de um meta-modelo do nível M2. Já no nível M0, estão os elementos reais a serem modelados. Pode-se observar, ainda, o padrão XMI permeando

os níveis M1, M2 e M3; segundo (Tenório, 2003) o XMI facilita a troca de meta-dados entre diferentes ferramentas de modelagem e repositórios de meta-dados.

Para a implementação realizada neste trabalho, o XMI é foi base das transformações. Isso será explicado de maneira mais detalhada em seções futuras, mas pode-se adiantar que um modelo UML-RT tem seus elementos representados por um arquivo XMI, que, por sua vez, é descrito por um metamodelo e é mapeado para uma gramática definida na ferramenta de transformações, Stratego/XT, podendo assim ser interpretado e transformado.

3.2.2. Visões e níveis de abstração

Analisar sistemas sob diferentes visões (*viewpoint*) de um sistema é uma técnica para abstração[2] que utiliza um conjunto de conceitos arquiteturais e regras de estruturação, com o objetivo de focar em um ponto específico de um sistema (MDA Guide, 2003).

A MDA especifica três pontos de vista e, consequentemente, seus respectivos modelos em um sistema de software (MDA Guide, 2003):

- Modelo Independente de Computação (CIM - *Computation Independent Model*)

Esse ponto-de-vista foca no ambiente do sistema e nos seus requisitos, os detalhes sobre a estrutura e processos do sistema são ocultados e, possivelmente, ainda indeterminados. O CIM pode ser chamado de modelo de domínio (*domain model*) ou de modelo de negócio (*business model*). Um vocabulário familiar aos envolvidos no domínio (problema) é utilizado em sua especificação. Esses modelos preenchem a lacuna existente entre os que são *experts* no domínio e aqueles *experts* no projeto e na construção de artefatos. Os requisitos dos sistemas são modelados por meio da descrição da situação na qual sistema será utilizado. Tipicamente este modelo é independente de como o sistema será implementado.

- Modelo Independente de Plataforma (PIM - *Platform Idependent Model*)

O PIM foca a operação do sistema (modelo computacional), ao mesmo tempo em que esconde os detalhes de uma plataforma particular, ou seja, inclui a representação das funcionalidades de negócios e comportamento, mas não expressa aspectos técnicos (OMG, OMG, 2009). Esse nível de modelo exibe a parte da especificação que não se altera de uma plataforma para outra, assim, o mesmo PIM poderá ser utilizado para diferentes plataformas. Ele pode utilizar uma linguagem de modelagem de propósito geral ou uma linguagem específica para a área na qual o sistema será utilizado.

- Modelo Específico de Plataforma (PSM - *Platform Specific Model*)

O PSM combina p PIM com um foco adicional nos detalhes do uso de uma plataforma específica pelo sistema modelado, contendo informações da tecnologia utilizada na aplicação como a linguagem de programação, os componentes de *middleware*, a arquitetura de hardware e de software. O PSM produzido pela

[2] Abstração é usada, neste caso, como o processo de suprimir detalhes selecionados para estabelecer um modelo simplificado.

transformação é um modelo do mesmo sistema especificado pelo PIM mais as características da plataforma escolhida. O PSM pode oferecer mais ou menos detalhes dependendo do seu propósito: pode ser uma implementação (em UML ao invés de código) se fornecer todas as informações necessárias para construir um sistema e para colocá-lo em operação; ou pode se caracterizar como um PIM se for utilizado para futuros refinamentos objetivando outro PSM que possa ser diretamente implementado.

A Figura 3 expõe essa estrutura dos modelos na MDA. É interessante observar a evolução entre os modelos: documentos informais são analisados e transformados manualmente em modelos CIM; novas transformações manuais são realizadas, permitindo chegar-se no modelo PIM; a partir de então, a idéia é que transformações automáticas sejam implementadas em modelos do mesmo nível e entre níveis diferentes para converter o PIM em um PSM e este PSM em código.

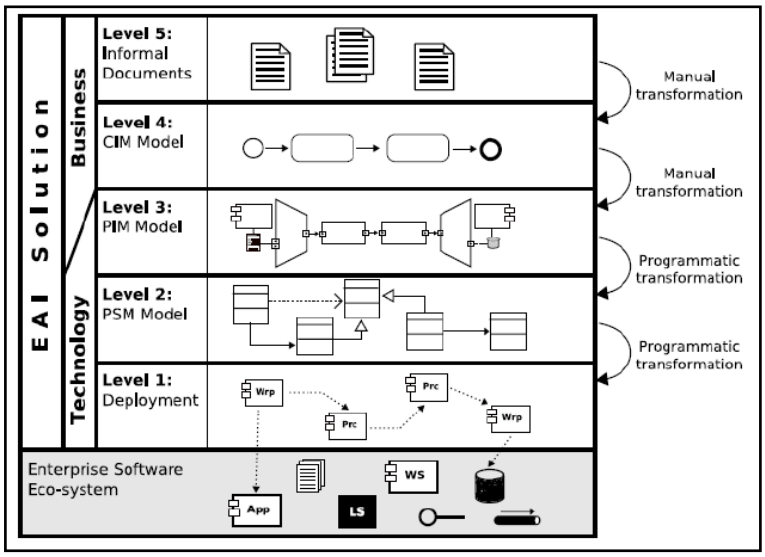


Figura 3 – Os modelos de acordo com os *viewpoint*s do MDA

Segundo (MDA Guide, 2003) a realidade ideal da MDA seria exatamente essa: permitir a definição de aplicações compreendidas por máquinas e modelos de dados que permitam a flexibilidade de:

- Implementação: até novas tecnologias e infra-estruturas poderão ser “plugadas” a modelos já existentes;
- Integração: pontes de integração também podem ser automatizadas;
- Manutenção: a especificação do sistema é totalmente acessível, tornando a manutenção muito mais simples;
- Teste e Simulação: os modelos que geram código também podem ser validados em confronto com os requisitos, testados em diversas infra-estruturas e usados para simular o comportamento dos sistemas desenvolvidos.

Segundo (Booch, G.; Rumbaugh, J.; Jacobson, I., 2000) todo modelo pode ser expresso em diferentes níveis de precisão. Utilizando-se dessa premissa no desenvolvimento em MDA, um sistema é definido em um modelo de alto nível e passa por transformações - que adicionam detalhes sobre o sistema ou que apenas convertem representações - até chegar a um modelo de uma tecnologia específica. Percebe-se que

as transformações entre modelos é o conceito chave para obter benefício do MDA, pois possibilita uma maior automação no desenvolvimento e implementação de software, reduzindo, consequentemente, esforço e custo (Bacelo, V; Lunelli, A., 2009.). A Figura 4 revela os tipos de transformações:

I. Modelo para Modelo: comumente aplicada para diminuir o nível de abstração do modelo; são refinamentos. As transformações automatizadas exploradas pela MDA não as PIM → PSM (uma vez que CIM → PIM exige muitos detalhes do usuário) e são essas transformações que trazem um grande benefício, pois o PIM, não se prendendo a semânticas específicas de cada plataforma pode gerar automaticamente diversos modelos PSM.
II. *Refactoring*: a transformação produz mudanças de reorganização no modelo de entrada (PIM ou PSM), preservando a semântica, mas alterando a estrutura. Em uma refatoração qualquer mudança não será percebida por entidades externas.
III. Modelo para código: gera código ou trechos de código executável a partir de um dado modelo.

As transformações que aumentam a abstração do sistema (Código → PSM; PSM → PIM; PIM → CIM) são, comumente chamadas de engenharia reversa.

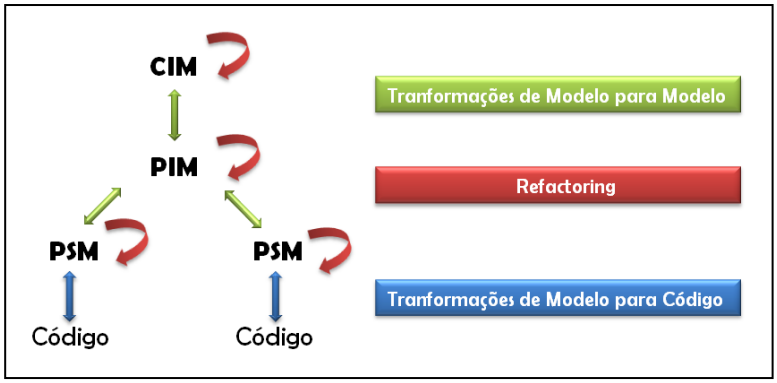


Figura 4 – Transformações nos *viewpoints*

Segundo (Flore, 2003), até o momento, entretanto, a criação das transformações entre PIM, PSM e código tem sido primariamente manual, exigindo muita mão-de-obra e bastante custosa por exigir uma equipe experiente de TI para obter sucesso. Para que MDA possa conquistar a produtividade prometida transformações automáticas ainda precisam de implementadas. Como a MDA tem bases em padrões gratuitos e disponíveis, existem muitas ferramentas inclusive do tipo open-source disponíveis. Entretanto, para modelos com suporte a concorrência – UML-RT – não há ferramenta no mercado que realize as transformações de maneira automática. Assim, neste trabalho, as transformações a serem exploradas serão as do tipo I – *Refactorings* no nível do PIM, colaborando com a automatização dessas reestruturações de um sistema modelado.

Algumas ferramentas dão suporte às transformações em um modelo em diversos passos: desde a análise inicial até o código executável (Brown, 2004), ou seja, desde uma visão mais abstrata até a mais concreta do sistema. Para isso, ferramentas de modelagem capturam o significado dos elementos dos diagramas e dos seus relacionamentos e utilizam essa compreensão para compor elementos de projeto, testes

de performance e até geração automática de partes do código da aplicação. Esse processo de transformações no modelo é o que se chama de refinamentos e segue a tendência que as próprias linguagens de desenvolvimentos sofrem: tornar-se cada vez mais abstrato e tentar automatizar o processo de refinamento (Bacelo, V; Lunelli, A., 2009.).

3.2.3. Metamodelagem

Outro ponto indispensável para a realização deste trabalho é o conceito de metamodelagem, que já foi tratado, entretanto, maiores detalhes serão oferecidos nesta seção. Um modelo passível de sofrer transformações automatizadas precisa ser representado em uma linguagem bem definida, esta definição é feita através do mecanismo de metamodelagem (Warmer, Jos; Kleppe, Anneke; Bast, Wim., 2003).

Relembrando, um modelo precisa ser definido de acordo com uma semântica fornecida por um metamodelo. Da mesma forma, um metamodelo tem que estar em conformidade com um meta-metamodelo. Nessas três camadas de arquitetura, o metametamodelo está em conformidade com sua própria semântica (Medeiros, 2008). Os meta-metamodelos existentes incluem o MOF e Ecore (Budinsky, 2004). O Ecore é introduzido com o Eclipse Modelling Framework e é originalmente baseado no MOF (Brand, 2009).

O objetivo de trabalhar com metamodelos é obter mecanismos que permitam verificar a consistência de cada uma das visões que são desenvolvidas é a definição precisa das construções e regras necessárias para a criação de modelos (Polido, 2007).

Este projeto optou por um metamodelo baseado no Ecore tanto pelo suporte oferecido pela ferramenta Eclipse (Eclipse, 2010) através do seu modelo .ecore que permite uma ótima gestão de elementos de modelos de forma textual ou gráfica; como por ter sido previamente definido por (Neto, 2008) e adaptado por (Oliveira, 2008).

O Anexo I apresenta duas visões do metamodelo e a seção 4.1.6, após uma explicação sobre a estrutura de UML-RT, apresenta o diagrama deste metamodelo. O EMF do Eclipe possui, ainda, uma extensão para criação de modelos arquivos XMI – neste caso, arquivos que representam modelos que estão de acordo com o metamodelo .ecore definido –, os quais auxiliaram tanto na construção das transformações como na sua validação, mais sobre esses arquivos na seção 4.1.6.

4. UML-RT e Transformações

4.1. UML-RT

Diante do exposto, percebemos que à medida que os sistemas distribuídos são cada vez mais necessários, eles trazem complexidade ao desenvolvimento; isso porque a especificação e projeto de sistemas distribuídos é uma tarefa complexa que envolve a especificação de dados, comportamento, comunicação e de aspectos arquiteturais do sistema (Ramos, R.; A. Sampaio e A. Mota. , 2006).

Por outro lado, há a busca por metodologias que suportem um desenvolvimento mais sistemático, menos dispendioso e com menor necessidade de retrabalho. Ou seja, apesar do aumento da complexidade dos sistemas, é desejável que o aumento da complexidade no seu desenvolvimento não siga a mesma proporção.

Como exibido, uma das metodologias que visa facilitar o desenvolvimento é a MDA. Para encaixar essas duas tendências, precisa-se de modelos que ofereçam suporte à modelagem da concorrência e da distribuição, permitindo, assim, o uso da MDA em sistemas distribuídos.

As abordagens tradicionais de UML mesmo com seus diagramas estáticos e dinâmicos – representando a estrutura e o comportamento respectivamente – possuem poucos mecanismos para detalhar características internas de componentes e propriedades como concorrência, interação e distribuição (Ferreira, 2006).

Para suprir essa necessidade no campo da modelagem desses sistemas a linguagem UML for Real-Time (UML-RT) foi criada (Selic, B.; Rumbaugh, J., 1998) e funciona como um *profile* de UML e ROOM para aplicações concorrentes e de tempo real. UML-RT incorpora conceitos de ROOM através de novos estereótipos e da adaptação de alguns diagramas convencionais. Há outras linguagens que têm o mesmo objetivo, com destaque para (Allen, 1997) que é uma ADL (*Architectural Description Language).* Na recente versão de UML 2.0 existe, inclusive, a idéia de componentes ativos. Neste trabalho, contudo, utilizamos UML-RT como linguagem de modelagem tanto por ser um modelo mais consolidado, como porque as transformações definidas por (Ramos, 2005), que são base a nossa proposta de automação, utilizam este *profile*.

UML-RT é um *profile* para UML (Selic, B.; Rumbaugh, J., 1998) utilizado para descrever sistemas concorrentes e distribuídos por meio da modelagem de sistemas com a definição clara de componentes que interagem entre si – por meio de protocolos – e operam concorrentemente. A comunicação é modelada através da troca de mensagens de entrada e saída, que podem ser síncronas ou assíncronas; para as transformações definidas elas são síncronas. (Ramos, 2005).

Embora o nome Real-Time sugira a presença de conceitos de tempo real, como restrições de tempo e requisitos não-funcionais, este *profile* é mais direcionado para modelar a arquitetura de sistemas distribuídos, com a facilidade para representar comportamentos reativos e concorrentes (Fischer, C.; Olderog, E.; Wehheim, H., 2001).

Em UML-RT quatro novos elementos de modelagem são introduzidos de maneira conservativa à UML e ROOM: cápsulas, portas, protocolos e conectores, conforme apresentados a seguir.

4.1.1. Cápsulas

Descrevem componentes arquiteturais cujos únicos pontos de interação são chamados de portas, também chamadas de classes ativas. As cápsulas correspondem ao conceito de Ator em ROOM; elas são complexas, potencialmente concorrentes e possivelmente distribuídas ativamente pelos componentes arquiteturais; a interação com

o ambiente por meio de um ou mais sinais baseados em objetos de fronteira chamados de portas (Ramos, R.; A. Sampaio e A. Mota. , 2006).

Cápsulas também possuem métodos e atributos, como as classe em UML; entretanto esses elementos são visíveis apenas no interior da cápsula, ou seja, são protegidos. A Figura 5 mostra como a cápsula se assemelha com uma classe, com o acréscimo do estereótipo e de um campo para as portas.

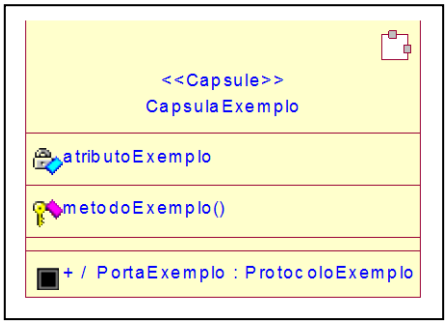


Figura 5 – Exemplo Cápsula

Segundo (Ramos, R.; A. Sampaio e A. Mota. , 2006) as cápsulas oferecem um elevado grau de *information hidding*, pois no mecanismo de comunicação por meio de mensagens todos os elementos (métodos e atributos) das cápsulas são escondidos, apenas as portas podem se conectar a outras cápsulas para estabelecer comunicação. Esse desacoplamento faz das cápsulas componentes altamente reusáveis.

4.1.2. Portas

As portas são o único meio de comunicação das cápsulas com o ambiente externo ou com suas sub-cápulas; é uma parte física da implementação de uma cápsula que intermedeia a interação da cápsula com o mundo externo. Uma porta é um objeto que implementa uma interface específica e realiza protocolo(s) (Lyons, 1998); ou seja, as portas são interligadas por conectores e comunicam sinais declarados previamente em protocolos.

O termo evento significa a recepção de uma mensagem pela porta, as portas podem ser classificadas – internamente a uma cápsula – como (Ferreira, 2006):

- Porta *End*: os eventos são percebidos diretamente pela máquina de estados da cápsula que pode ler e enviar mensagem através destas; são canais de troca de dados entre a cápsula e o ambiente externo ou sub-cápsulas.
- Porta *Relay*: são portas conectadas a outras portas, com eventos não percebidos pelas máquinas de estados da cápsula que a contém; os seus eventos são encaminhados a outro elemento (delegação).

4.1.3. Protocolos

São representados pelo estereótipo <<Protocol>> em UML-RT (Figura 6) e são classes abstratas e puramente comportamentais. Em um mesmo protocolo é possível definir mais de uma interface, chamada papel, que será implementada pela porta, geralmente, entretanto, se utiliza protocolos binários: base e conjugado, gerado pela inversão dos sinais e representado por um til (~) (Ferreira, 2006). Os protocolos definem o

fluxo válido de informações (os sinais) entre as portas conectadas das cápsulas (Lyons, 1998); em um sentido mais abstrato os protocolos funcionam como obrigações contratuais que existem entre as cápsulas. Os protocolos são altamente reusáveis por definem uma interface abstrata que será realizada pela porta.

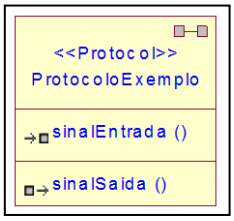


Figura 6 – Exemplo Protocolo

4.1.4. Conectores

Conectores são utilizados para conectar portas, sendo então uma visão abstrata do canal de comunicação baseado em sinais (mensagens) (Maranhão, 2005); são os conectores que capturam os relacionamentos com relevância arquitetural; cápsulas que interferem outras cápsulas por meio da comunicação (Lyons, 1998).

4.1.5. Diagramas

O modelo UML-RT é formando ainda por um conjunto de diagramas, na realidade esses diagramas são estendidos de UML: classe, estados e estrutura que representam, respectivamente, dados estáticos, comportamento dinâmico e relações entre as instâncias.

No diagrama de estados o funcionamento de uma simples cápsula é realizado diretamente pela máquina de estados associada àquela cápsula, enquanto as cápsulas mais complexas podem combinar a máquina de estado com uma rede interna de sub-cápsulas colaborativas interligadas por conectores; essas sub-cápsulas podem, inclusive, também serem compostas por sub-cápsulas (Lyons, 1998).

Os diagramas de estrutura representam uma extensão de diagramas de colaboração.

Percebe-se que essas duas tendências – Sistemas Distribuídos e MDA – realmente devem ser bem exploradas porque em sistemas distribuídos há extrema necessidade de modularização e, com modelos para representar essa disposição, a implementação e o entendimento do sistema são bastante aperfeiçoados.

Antes do desenvolvimento de qualquer sistema que promova automatização de transformações em modelos, é necessário que essas transformações já tenham sido validadas. Nas transformações a serem exploradas - *refactorings* - é preciso que uma lei de transformação possa ser aplicada em ambos os sentidos (obedecendo às condições) e mantenha o comportamento do sistema. Assim, as transformações definidas por (Ramos, 2005) serão as bases para o desenvolvimento dos experimentos desse trabalho: essas leis são divididas em leis básicas e leis derivadas e a especificação e prova formal das leis podem ser encontradas em (Ramos, 2005). Essas leis lidam tanto com aspectos estáticos quanto dinâmicos representados pelos três principais diagramas de UML-RT – classe, estrutura e estados – introduzidos na seção 4.1.5.

Os conceitos de UML-RT foram incorporados na ferramenta CASE *Rational Rose Real-TimeI* (RoseRT) (Gu, Z.; Shin, K.), essa ferramenta foi utilizada para elaboração de exemplos introdutórios à UML-RT e para a representação de modelos na seção de exemplos das transformações. Infelizmente, a conversão para arquivos XMI não existe na versão atual da ferramenta (IBM, 2010).

4.1.6. Metamodelo

As três figuras a seguir apresentam o metamodelo (o mesmo que o do Anexo I – XMI do Ecore e Visão Estruturada), com a visão de diagrama. Essa foi a base para a criação da gramática em Stratego. A primeira parte, Figura 7 oferece uma idéia das relações no diagrama de estados; a Figura 8 do diagrama de classes; e a Figura 9 do diagrama de estrutura. Alguns componentes estão presentes em mais de uma das partes porque fazem a ligação entre elas. É útil para se ter uma visão geral dos elementos do metamodelo; é interessante notar, o reflexo dos elementos no XMI: as agregações geram nomes de *tags* enquanto que as associações geram atributos para uma *tag*, por exemplo. Mais detalhes sobre o metamodelo em (Oliveira, 2008).

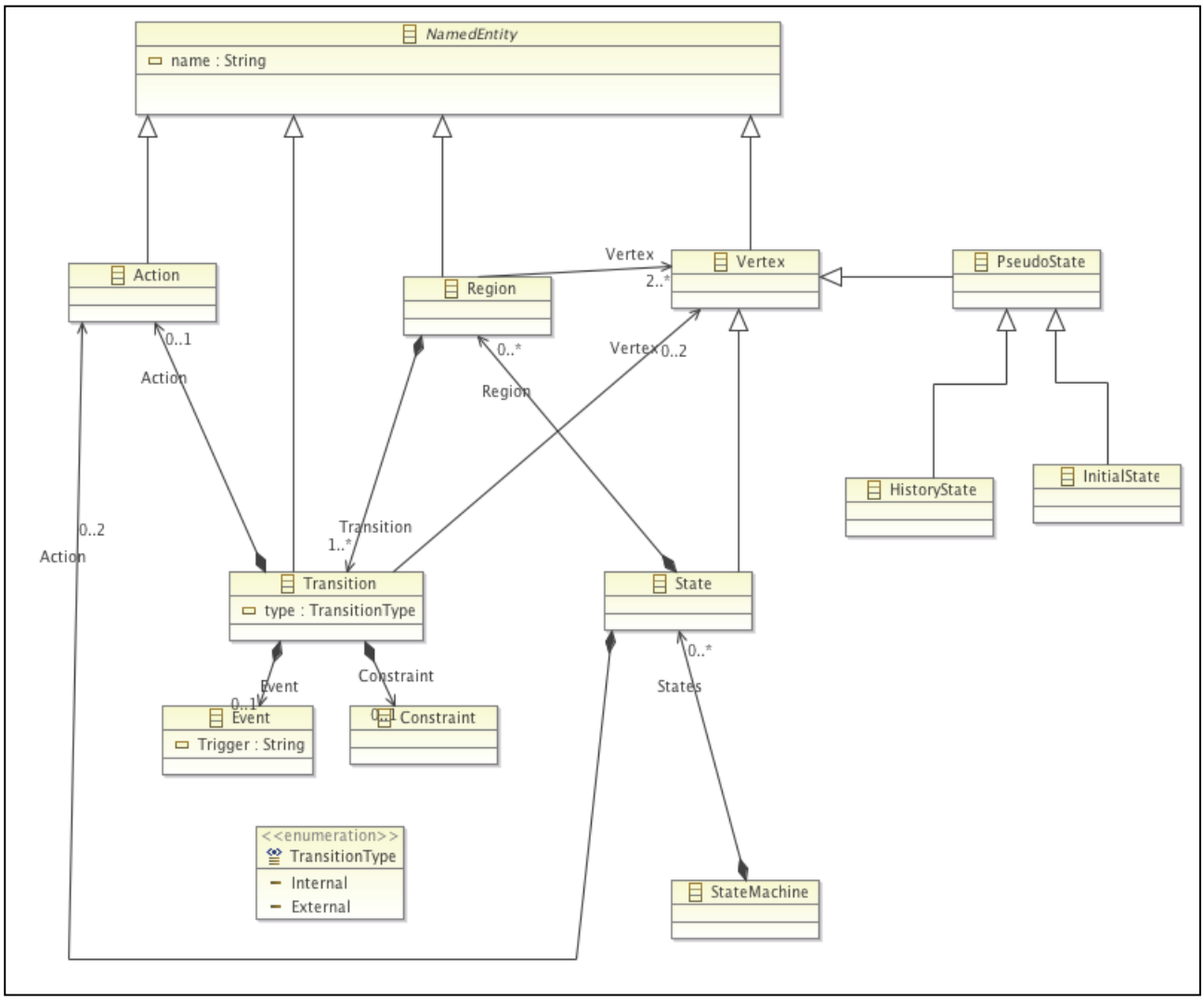


Figura 7 - Diagramas Metamodelo – Diagrama de Estados

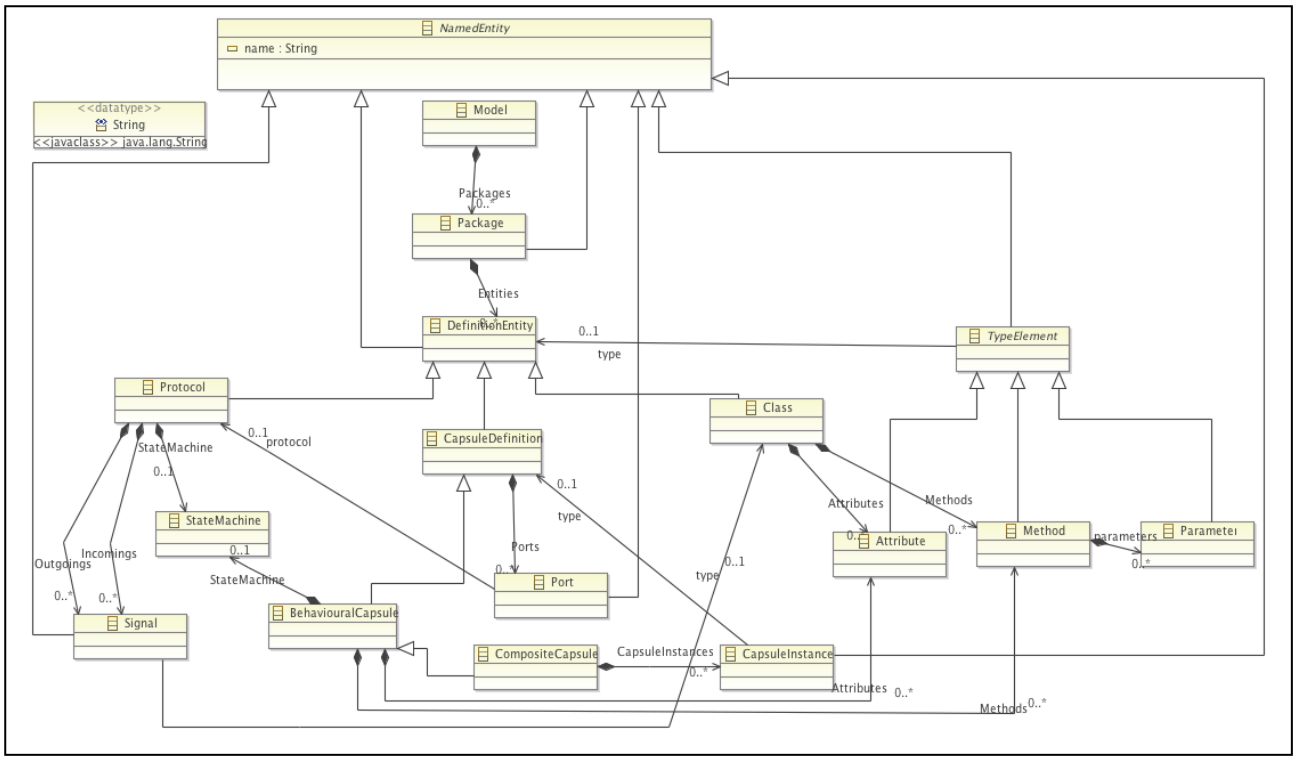


Figura 8 - Diagramas Metamodelo – Diagrama de Classes

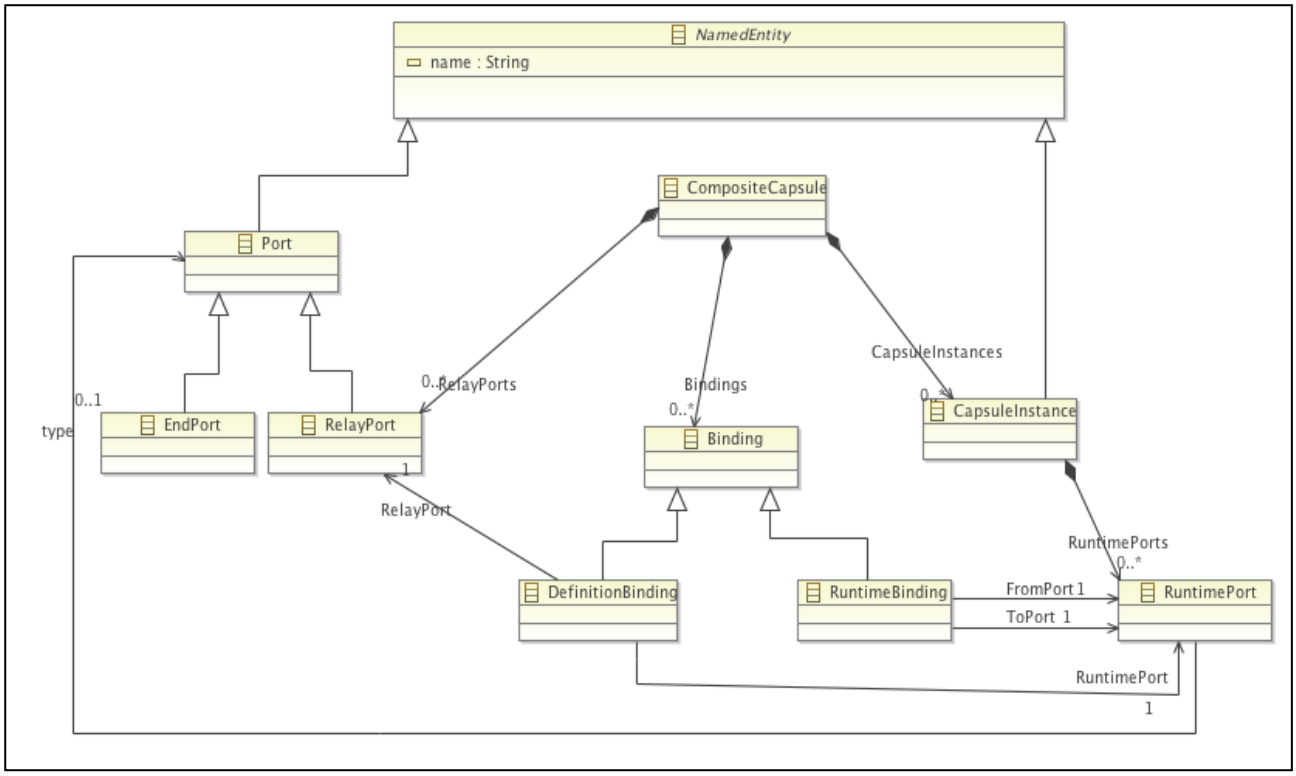


Figura 9 - Diagramas Metamodelo – Diagrama de Estrutura

4.1.7. XMI

O objetivo principal do XMI é permitir a troca de dados e meta-dados entre ferramentas de modelagem de UML e entre ferramentas e repositório de meta-dados em ambientes heterogêneos e distribuídos (Demuth & Obermaier, 2000).

XMI também é a linguagem básica para a realização de transformação entre modelos UML. Para as transformações aplicadas neste trabalho não é diferente. O arquivo XMI descreve o modelo de maneira compatível com o metamodelo apresentado na seção anterior. É com base nesse arquivo XMI que as transformações poderão ser aplicadas, resultando em outro arquivo XMI.

A Figura 10 mostra um exemplo de um arquivo XMI. As quatro primeiras linhas estão apenas especificando as versões do XML, do XMI e o metamodelo utilizado (neste caso é a indicação do caminho no qual esse metamodelo está localizado, este é o arquivo representado na seção anterior). Em seguida, há um elemento do tipo Package que possui dois elementos, uma cápsula e uma classe.

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:metamodel="platform:/resource/br.ufpe.cin.ic/oldMetamodel/Metamodel.ecore">
  <Packages name="Package_Intr_Capsule_Class_Assoc_Model">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Capsule_1_Association"/>
    <Entities xsi:type="metamodel:Class" name="Class_association_1"/>
  </Packages>
</metamodel:Model>
```

Figura 10 – Exemplo arquivo XMI

O diagrama de classes que esse arquivo especifica é mostrado na Figura 11.

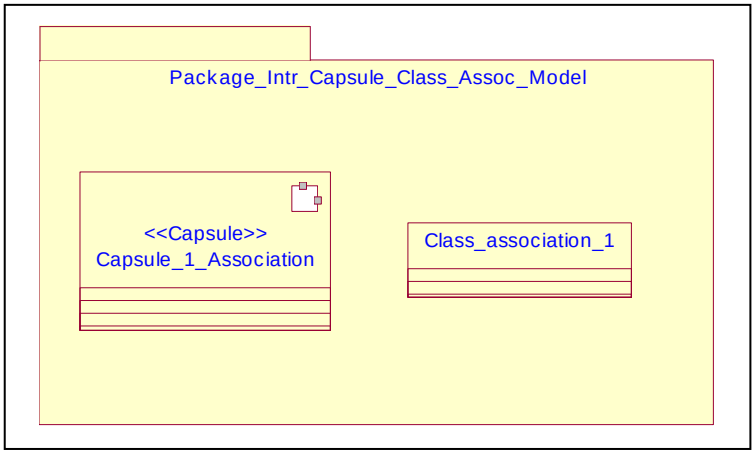


Figura 11 – Diagrama de Classe do arquivo XMI

4.2. Transformações

A transformação de modelos – de maneira automatizada – é o conceito chave de MDA; essas transformações são a geração automática de um modelo destino a partir de um modelo origem com base em algumas regras de transformação; ou seja, é a conversão entre modelos de um mesmo sistema (MDA Guide, 2003).

Como anteriormente mencionado, as transformação podem ocorrer entre mesmos níveis e entre diferentes níveis de abstração e também pode ocorrer entre mesmos domínios e diferentes. Segundo (Warmer, Jos; Kleppe, Anneke; Bast, Wim., 2003) essas transformações são definidas por um conjunto de regras que juntas descrevem como um modelo na linguagem origem pode ser transformado em um ou mais modelos na linguagem destino.

Existem diferentes método para se definir as regras de transformação, em geral, as elas descrevem como elementos de um modelo de origem devem ser traduzidos em um modelo de destino. Esse tipo de regra é formado por duas partes: lado esquerdo (*left-hand side* - LHS) que acessa o modelo de origem e o lado direito (*right-hand side* - RHS) que atua no modelo de destino.

4.2.1. Tipos de Transformações

Dependendo do tipo de implementação dessas regras há uma categorização para as abordagens de transformações definidas por (MDA Guide, 2003):

Marking (Marcação): a Figura 12 mostra como esse tipo de transformação é realizada: uma plataforma é escolhida; um mapeamento para essa plataforma é criado ou já está disponível, o qual inclui um conjunto de marcas. Assim, essas marcas no modelo (na figura um modelo PIM) são utilizadas para guiar as transformações e o modelo marcado é transformado – utilizando o mapeamento – para produzir o modelo de saída (na figura um modelo PSM).

Como exemplo, ao converter um PIM em EJB (Enterprise JavaBeans) - um componente do tipo servidor da plataforma Java que executa no container do servidor de aplicação – pode-se utilizar um mapeamento por meio de marcações no qual a marcação "*Sessions*" em uma classe UML resultará em uma "*Session Bean*" do EJB.

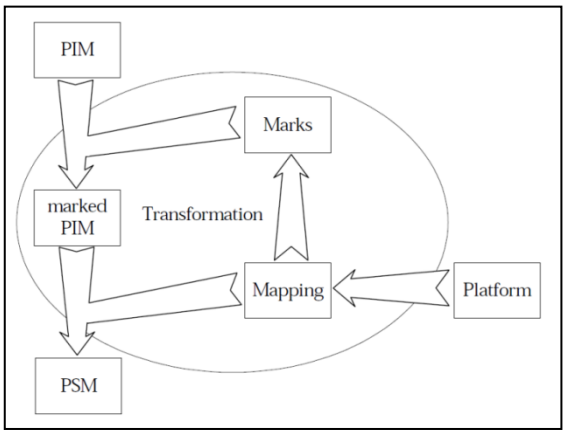


Figura 12 – *Marking*

Transformação por Metamodelos: um modelo é preparado utilizando uma linguagem especificada por um metamodelo; escolhe-se um metamodelo que especifica a linguagem destino, e transformações – especificadas previamente e que são mapeamentos entre os metamodelos – serão aplicadas. A Figura 13 mostra graficamente as etapas desse tipo de transformação para o caso específico de transformações de um modelo PIM para um modelo PSM.

Por exemplo, um metamodelo para um modelo PIM é o *EDOC ECA Business Process Model*[3] e o metamodelo do modelo PSM é um engenho de workflow especificado em MOF. A transformação será definida utilizando QVT e um par de modelos que se equivalem baseados nesses metamodelos.

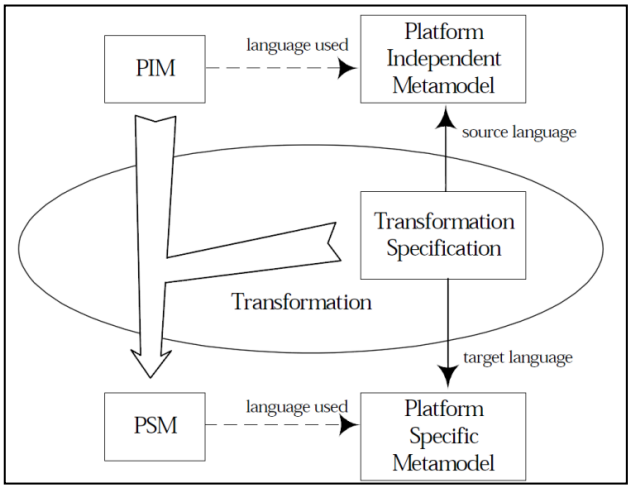


Figura 13 – Transformação por Metamodelos

Transformação de Modelos: a Figura 14 mostra esse tipo de transformação (utilizando modelos PIM e PSM, como modelos de entrada e de saída, respectivamente). Um modelo é preparado com tipos especificados em um modelo; ou seja, seus elementos são subtipos dos tipos desse modelo. As transformações são especificadas e, termos de mapeamentos entre os tipos; assim, são aplicadas, e o modelo resultante possui elementos do tipo da plataforma destino selecionada.

No exemplo mostrado na figura o *Plataform Independent Types* declara capacidades e recursos genéricos; enquanto o *Plataform Specific Types* é uma mistura de classes e classes compostas que proporcionam as mesmas capacidades e recursos, mas para uma plataforma específica. A abordagem de transformação por modelos difere do mapeamento por metamodelo especialmente porque aquela utiliza tipos especificados em um modelo para realizar o mapeamento, enquanto essa utiliza conceitos especificados em um metamodelo.

[3] *Enterprise Distributed Object Computing* (EDOC) um padrão OMG com um componente chave *Enterprise Collaboration Architecture* (ECA)

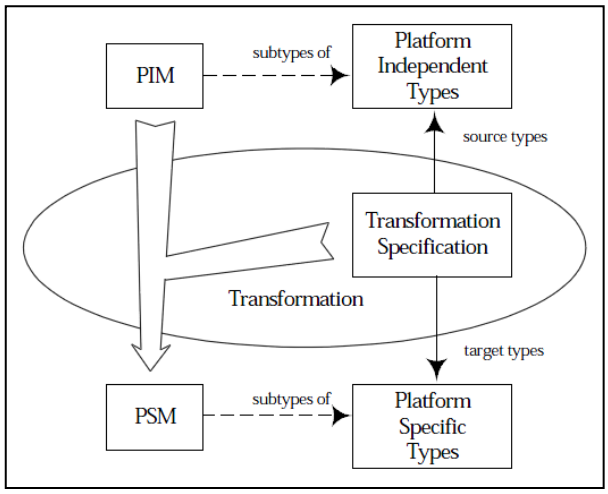


Figura 14- Transformação de Modelos

Aplicação de padrões: é uma extensão do mapeamento entre metamodelos e da transformação de modelos com a inclusão de padrões nos tipos ou nos conceitos das linguagens. A Figura 15 mostra como os padrões estão junto das representações dos tipos dos elementos em um determinado modelo.

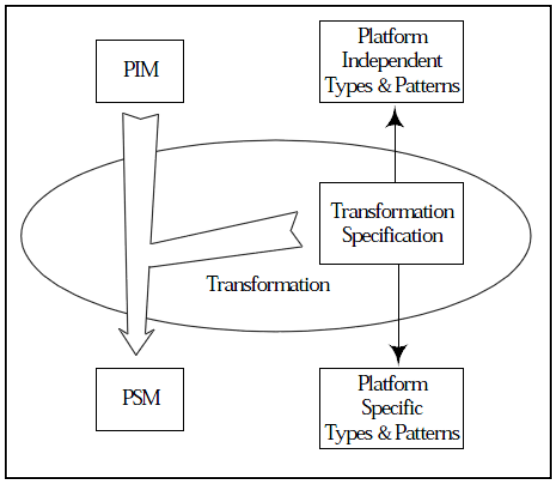


Figura 15 – Aplicação de padrões na transformação de modelos

Esse acréscimo de padrões modifica apenas o escopo de aplicação das regras, pois ao invés de aplicar ao modelo como um todo, regras podem ser definidas como aplicáveis a padrões específicos.

Fusão de Modelos (*Model Merging*): há diversas abordagens de MDA que são baseadas na idéia de fundir modelos, a representação dessa possibilidade está na Figura 16, neste caso, um modelo PIM está se juntando a algum outro modelo e gerando PSM, certamente, transformações precisariam ser feitas para tornar compatíveis os dois modelos de entrada.

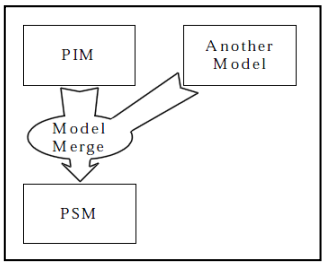


Figura 16 – Fusão de Modelos

Para todas as transformações, contudo, é necessário que as regras de transformações sejam precisas. Para este trabalho, as transformações baseadas no metamodelo e em padrões (busca-se os padrões no modelo de origem) foram utilizadas; afinal o arquivo XMI (representação do modelo UML-RT) é interpretado pela ferramenta e as transformações são aplicadas de acordo com o casamento de padrão nessa estrutura interpretada.

O desenvolvimento de estratégias é uma demanda existente especialmente porque o desenvolvimento dirigido a modelos está se tornando um padrão industrial. Nesse contexto, as transformações, antes de implementadas, precisam ser comprovadas como válidas no seu domínio.

Neste contexto, este trabalho implementa algumas das transformações que especificadas e provadas em (Ramos, 2005). O conjunto dessas leis de transformações objetiva sistematizar a evolução semântica bem-definida de modelos UML-RT, com a preservação de aspectos estáticos e dinâmicos; propriedade estruturais e comportamentais, respectivamente (Ramos, R.; A. Sampaio e A. Mota. , 2006).

As leis suportam a transformação de modelos iniciais da etapa de análise, para modelos mais concretos referentes a etapa de projeto. Além disso, essas leis são algébricas para UML-RT e capturam tanto transformações simples no modelo, como *refactorings* precisos (Godoi, R.; Ramos, R.; Sampaio, A., 2006).

A proposta da MDA é evoluir com um modelo, atingindo níveis mais baixos de abstração possibilitando chegar até no código-fonte. O conjunto de leis demonstradas por (Ramos, 2005), entretanto, possibilitam, além da evolução, a normalização de um modelo arbitrário de UML-RT: partindo-se de um modelo complexo, é possível gerar um modelo com uma única cápsula ativa, mais detalhes sobre essa normalização na seção 7.

Essa possibilidade decorre do fato de todas as transformações serem bidirecionais. A Figura 17 – Representação básica das é uma visão geral de como as transformações são apresentadas. Os modelos “Modelo 1” e “Modelo 2” são semanticamente equivalentes. As transformações podem ser aplicadas nos dois sentidos desde que o modelo de entrada seja compatível[4] com o “Metamodelo 1” na transformação da esquerda para a direita; ou com “Metamodelo 2” na transformação da direita para a esquerda e que as condições em cada sentido também sejam respeitadas.

[4] A expressão compatível, neste caso, é utilizada para designar que apenas parte do modelo precisa corresponder ao “Metamodelo 1” ou ao “Metamodelo 2”, nas transformações reversas.

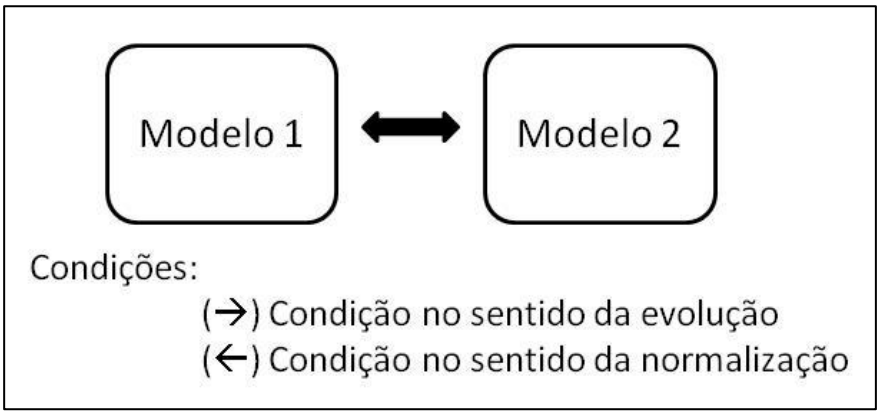


Figura 17 – Representação básica das Transformações

A Tabela 1 – Leis de Transformação Implementadas mostra as transformações que foram implementadas neste projeto. Maiores detalhes dessas leis de transformações estão no Anexo III – Leis de Transformações Implementadas, e o restante pode ser encontrado em (Ramos, 2005).

1	Declarar Cápsula
2	Introduzir Associação Cápsula-Cápsula
3	Introduzir Método
4	Introduzir Associação Cápsula-Classe
5	Introduzir Associação Cápsula-Protocolo

Tabela 1 – Leis de Transformação Implementadas

Segundo (Pereira, M.A.; Prado, A.F.; Biajiz, M.; Fontanette, V.; Lucrédio, D., 2006) o transformador pode viabilizar:

A – Novos modelos no mesmo domínio, a fim de se obter um grau maior de especificidade em relação ao modelo original, essa transformação é chamada de endógena ou *rephrasings*.

B – Novos modelos em domínios diferentes, a fim de se obter reuso de modelos para criação de novos modelos em diferentes domínios, essa transformação é chamada de exógena ou *translation.*

As transformações implementadas são do tipo endógenas e objetivam tornar o modelo de entrada em um modelo de saída com uma estrutura mais adequada. A Figura 18 mostra como a Figura 13 se enquadra neste contexto trabalhado, afinal as transformações são do tipo metamodelo. É interessante observar que o metamodelo origem e o metamodelo destino são iguais, justamente por manter o modelo no mesmo domínio. Ainda nesta imagem podemos observar o "*Trasnformation Model*" que é são as próprias transformações definidas por regras em Stratego com auxílio da gramática também definida em Stratego (essas regras são baseadas no conhecimento das

estruturas dos elementos do metamodelo), as estratégias de aplicação das transformações e as ferramentas oferecidas por XT; mais sobre a ferramenta utilizada e os detalhes da implementação na seção 6.1. Para a concretização das transformações deve haver o mapeamento entre elementos da linguagem fonte para elementos da linguagem destino; esses mapeamentos são as regras de transformações, que definem quando e quais alterações serão realizadas o modelo.

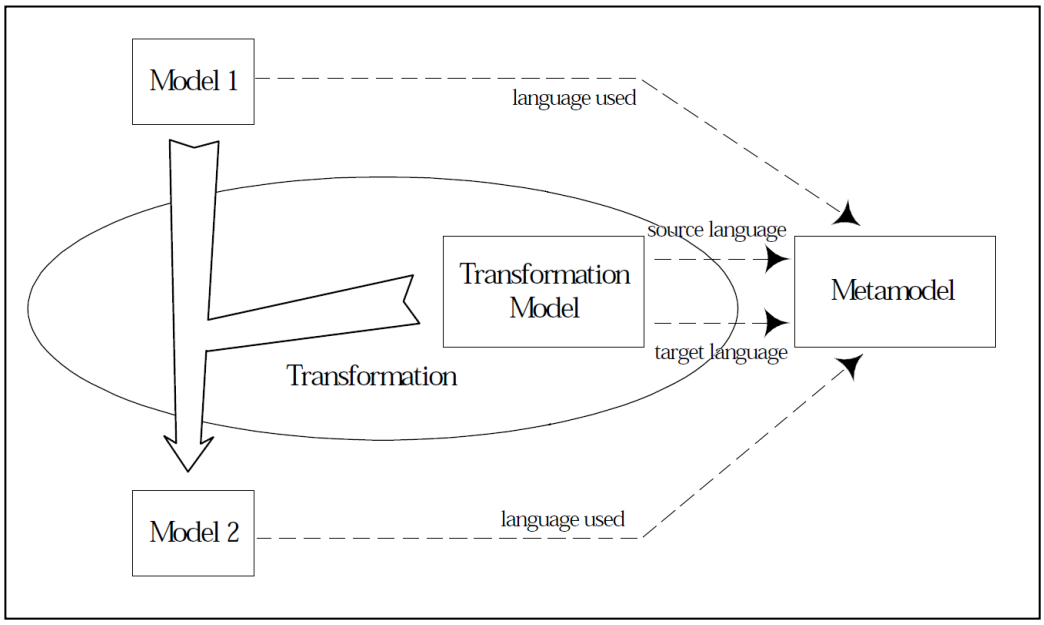


Figura 18 – Visão geral das transformações

Mais especificamente as trasnfromações neste trabalho são aplicadas como mostra a Figura 19:

Figura 19 – Visão específica das transformações

5. Stratego/XT

Segundo (Agrawal, A.; Karsai, G., Shi, F., 2002) a escrita de transformações é uma atividade complexa, por isso a necessidade do apoio de ferramentas.

Stratego/XT é definido por (Stratego, 2010) como um framework dividido em duas partes: a linguagem (Stratego) e a ferramenta (XT) para a transformação de programas. A linguagem fornece regras para expressar transformações básicas; estratégias programáveis para controlar a aplicação das regras; sintaxe concreta para expressar padrões de regras na sintaxe da linguagem destino e reescrita dinâmica de regras para expressar transformações sensíveis ao contexto, suportando, assim, o desenvolvimento de componentes de transformação num alto nível de abstração. A ferramenta, por sua vez, oferece um conjunto de ferramentas de transformações extensíveis e reusáveis como poderosos *parsers*[5] e geradores de *pretty-printer*[6].

Apesar dessa separação entre Stratego: linguagem e XT: ferramenta; quando o termo Stratego for utilizado, será uma referência ao framework como um todo; a menos que haja a especificação de que se fala da linguagem.

5.1. O framework

Stratego/XT é uma DSL (*Domain-Specifc Languagem*[7]) para implementar transformações em linguagens. Tecnologias como essa são conhecidas por promover a Meta-Programação porque nelas são implementados programas que atuam em programas. Assim, Startego/XT suporta o desenvolvimento de infra-estrutura de transformações, de outras DSL's, de compiladores, de geradores de programas e uma gama de atividades de meta-programação.

Alguns conceitos são fundamentais para o entendimento do framework e serão mais bem detalhados nas seções seguintes: termos, regras, estratégia e meta-variáveis; todas essas definições são baseadas no manual de Stratego/XT disponível em (Stratego, 2010).

A aplicação das transformações em Stratego é baseada na idéia de casamento de padrão: para aplicação de qualquer transformação o programa testa se determinada parte de um arquivo de entrada possui uma estrutura que casa – se encaixa – com um determinado padrão e, então, aplica a transformação.

O resultado da tentativa de aplicação de uma transformação é a própria transformação ou a falha; com o uso de estratégias corretas, é possível continuar com a execução das tentativas de transformações mesmo que não haja casamento de padrão; ou seja, mesmo que haja falha. É interessante observar que há a possibilidade de aplicação de mais de uma regra de transformação.

Stratego possui o conceito de módulos o que aumenta a possibilidade de reuso dos sistemas, tanto de transformações como de definições de sintaxes.

---

[5] *Parser* é um analisador sintático que permite a interpretação de um programa de acordo com uma gramática.

[6] *Pretty-print* é a aplicação de diversos formatos de estilos a textos, código-fonte, markup e outros conteúdos similares.

[7] Uma DSL (Linguagem de Domínio Específico) é uma linguagem de programação ou uma especificação executável de uma linguagem que oferece – por meio de notações apropriadas e abstrações – um poder expressivo focado, e comumente restrito a, um problema particular.

O framework inclui também algumas sintaxes já pré-definidas, como AspectJ, Java, Jimple, PHP4, PHP5; possibilitando, assim, a implementação de transformações imediata. Entretanto, permite, também a definição de sintaxes por meio de SDF (*Syntax Definition Formalism*) que é uma meta-sintaxe usada para definir gramáticas livres do contexto; ou seja, é uma maneira formal de descrever linguagens. Em Stratego é possível definir a gramática livre de contexto, construtores (facilitam a leitura), a parte léxica, as variáveis, as prioridades, entre outros elementos. A SDF suporta a definição de linguagens de programação de maneira modular, declarativa e com alto-nível. A modularidade do formalismo permite que duas linguagens sejam facilmente combinadas ou que uma seja incluída dentro de outra.

Essa é o procedimento utilizado neste projeto: uma gramática foi escrita em Stratego, essa gramática é capaz de interpretar os modelos UML-RT expressos em XMI e as regras objetivam aplicar as transformações.

5.1.1. A representação em ATerm

Alguns sistemas de transformação atuam diretamente no texto; esta abordagem, entretanto, não é indicada para transformações complexas, sendo é necessária uma representação estruturada. São os *parsers* que realizam essa conversão do texto – linguagem na qual o programa é escrito – para os termos estruturados; e os *unparsers* realizam a tradução contrária. Essas duas ferramentas foram indispensáveis neste trabalho e convertem o XMI para uma estrutura mais adequada e dessa estrutura de volta ao XMI.

Em Stratego, essa representação estruturada é o que se chama de Aterm (*Annotaded Term*); o formato fornece um conjunto de construções para a representação em árvore. ATerms são utilizados para a representação externa – é a maneira na qual os arquivos são manipulados pelas transformações – e também para a representação interna dos programas em Stratego/XT, isso porque essa representação é utilizada na própria linguagem Stratego e a execução de Stratego é baseada na biblioteca Aterm.

É interessante que Startego permite que se use *parsers* externos, oferece, contudo, meios para que esse *parser* seja implementado a partir da definição de uma gramática. A Figura 20 oferece uma visão geral de como os programas podem ser construídos em Stratego/XT: a sintaxe SDF, com auxílio de ferramentas de XT, gera o *parser*, a gramática em árvore e o *unparser* (por meio do *pretty-printer generator*); a tabela de *parser* permite que o programa seja convertido para a representação ATerm-árvore; a árvore da gramática permite que as transformações implementadas sejam aplicadas; e a tabela de pretty-print permite que o programa seja convertido na linguagem desejada. Neste trabalho, a única sintaxe utilizada é a do metamodelo, assim, o *unparser* leva à mesma linguagem de origem.

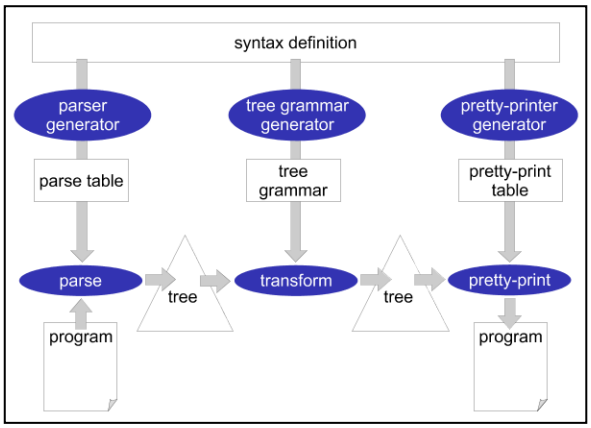


Figura 20 - Visão Geral do Processo em Stratego/XT

Os ATerms são construídos a partir dos seguintes elementos básicos:

- Inteiros (*Interger*): uma constante inteira, que é, na realidade, uma lista de dígitos decimais;
- String: uma string constantes é uma lista de caracteres entre aspas duplas; caracteres especiais precisam ser “escapados” utilizando um *backlash* (barra ao contrário);
- Aplicação de Construtor: um construtor é um identificador (uma string alfanumérica), a aplicação de um construtor c(t1, ..., tn), por exemplo, cria um termo por meio da aplicação do construtor c aos termos t1,..., tn;
- Lista (*List*): é um termo da forma [t1,...,tn], enquanto construtores têm o mesmo número de subtermos, as listas podem ter um número variáveis, mas numa lista os termos são timicamente do mesmo tipo;
- Tupla (*Tuple*): é a aplicação de um construtor sem o construtor;
- Anotação (*Annotation*): os elementos definidos acima são utilizados para criação da parte estrutural do termo; opcionalmente um termo pode ser anotado com uma lista de termos que adicionam informações semânticas sobre o termo; *annotated term* tem o formato: `t{t1,...,tn}`.

Para uma idéia mais clara, a Tabela 2 mostra como um fragmento de código seria na representação em ATerm.

Código	ATerm
`4 + f(5 * x)`	`Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))]))`

Tabela 2 – Exemplo de ATerm

5.1.2. As Regras

São as próprias transformações, mas as que são simples; que podem ser escritas com a sintaxe. Elas são ditas funções parciais de termos para termos; o termo parcial é utilizado porque podem retornar falha (*fail*).

A Figura 21 revela a estrutura básica de uma regra em Startego/XT. O LHS é o padrão que se procura casar e o RHS é o padrão que será aplicado.

```
NomeDaRegra:
  LHS -> RHS
  where
    <estrategiaCondicional>
```

Figura 21 – Estrutura de regra em Stratego/XT

A cláusula *where* é opcional na construção de uma regra, optou-se por exibi-la (que é, na realidade, uma estratégia) logo aqui, entretanto, porque para a todas as transformações deste trabalho uma ou mais condições de aplicação são utilizadas.

Uma regra pode ser aplicada a um termo através do seu nome; se o termo casar com o LHS (e com as condições, se houver); a transformação será aplicada e este termo será substituído pelo RHS. É interessante observar que as variáveis ligadas (*bind*) durante o casamento de padrão serão mantidas na substituição. Como mencionado anteriormente, a aplicação dessa regra falhará se o termo não casar com o LHS ou se a condição *where* falhar.

Uma regra, originalmente, é aplicada à raiz de um termo, e não a todos os subtermos. Para que esse subtermo seja considerado, certas estratégias precisam ser utilizadas.

Outra característica das regras é a possibilidade de haver múltiplas regras com o mesmo nome; essa realidade implica na tentativa de aplicação de todas as regras caso esse nome seja convocado até que algum tenha sucesso ou até que todas falhem. A ordem de aplicação dessas regras, contudo, não é definitiva, assim, só há razão para utilizar essa técnica quando as regras forem mutuamente exclusivas (i.e. não possuem sobreposição do LHS; não houver como um termo casar com mais de uma regra).

As regras permitem o uso não só dos ATerms para definição dos padrões, mas da própria sintaxe.

As regras são, na realidade, uma convenção sintática de definição de estratégias; segue as características das estratégias no contexto de Stratego/XT.

5.1.3. As Estratégias

As expressões de estratégia (*strategy expression)* são combinações de uma ou mais transformações (regras) em uma nova transformação, mais complexa.

Operadores de estratégia (*strategy operators*) são funções de transformações para transformações.

Os nomes das regras são expressões básicas de estratégia e a partir de uma coleção de regras transformações mais complexas podem ser criadas com o uso de operadores.

Stratgeo/XT oferece um conjunto de estratégias redefinidas. A estratégia "*innermost*" cria de um conjunto de regras uma nova transformação que aplica essas regras de maneira exaustiva. Essa estratégia sempre tem sucesso, mas pode não terminar; outras podem retornar falha; assim, as estratégias estendem a características dos termos e também são funções parciais de termos para termos.

As estratégias nem sempre definem um termo para a sua execução; é comum o uso do termo corrente (*current term*) de maneira implícita para a aplicação da estratégia. Ou seja, não há variável que é ligada ao termo corrente e em seguida passado como argumento para a estratégia; assim, um operador de estratégia pode ser dito como uma função de transformação para transformação.

As estratégias foram utilizadas nesse trabalho de duas principais maneiras:

- Estratégias que definem a regra a ser aplicada
- Estratégias criadas para as cláusulas condicionais

No primeiro caso, o uso das *traversals* pré-definidas por Stratego foi fundamental. As *traversals strategies* definem como será a passagem – a leitura e busca de termos – ao longo da árvores (da representação em ATerm). Por exemplo, uma estratégia "*bottom-up*" visita os subtermos de um nó antes de visitar o próprio nó, enquanto a estratégia "*top-down*" visita um nó e, em seguida, os seus filhos.

No segundo caso, para cada condição uma ou mais estratégias foram criadas e a sua execução, em geral, invoca novas regras.

5.1.4. As meta-variaveis

A possibilidade de definição de metas-variáveis foi fundamental para implementação das transformações, uma vez que elas devem ser genéricas e aplicáveis em diferentes situações independente do nome de uma cápsula, por exemplo.

Embora seja mais fácil escrever padrões com a menor quantidade de meta-variáveis; essa característica é desejável, e com Stratego/XT é obitida por meio de declarações de variáveis (*variable declarations*) na definição da SDF.

5.2. A solução adotada

A vantagem de se utilizar uma DSL é que a implementação pode ser conquistada a partir de uma representação concisa; com Stratego/XT a possibilidade de reconhecimento de padrão de maneira automática e as estratégias pré-definidas ofereceram poder extra para a elaboração das transformações.

Além disso, todo o apoio para a construção de parsers e pritty-priters foi fundamental já que a realidade foi a criação de uma nova gramática. Neste ponto, Stratego também ofereceu vantagens ao dispor ferramentas que permitiram a construção da SDF de uma maneira objetiva.

Os principais arquivos criados se dividem em dois principais grupos:

5.2.1. A gramática

Se deu com o mapeamento do Metamodelo.ecore para uma gramática que permitisse a criação das regras de maneira genérica; arquivos .sdf foram criados e representam a parte da sintaxe livre do contexto; a parte léxica e a definição das variáveis. OS arquivos encontram-se no Anexo II – Gramática do Metamodelo.

Para este grupo, as ferramentas de XT foram a base para a interpretação desses arquivos.

5.2.2. As transformações

São as regras e estratégias escritas em Stratego. Um exemplo claro de como esses arquivos são é mostrado na seção 6.1.

5.2.3. A metodologia criada

Para a aplicação das transformações uma série de passos precisa ser realizada. Para tornar esse método um pouco mais didático foi dividido em três partes uma responsável pela interpretação da gramática, outra pela importação da gramática no contexto das transformações e outra pela aplicação real das estratégias. Essas etapas devem ser executadas nesta ordem (com algumas exceções que poderiam ser executadas independentemente da ordem).

A Figura 22 mostra a primeira parte: com a sintaxe definida por meio de arquivos sdf uma só definição de gramática é criada agregando todas as partes da gramática em um arquivo def; em seguida, o *parser*, baseado nessa definição, pode ser criado; e, então, um arquivo de entrada (do tipo XMI) poderá ser lido, interpretado e transformado em Aterm; a última etapa é apenas a edição do arquivo Aterm para que sua visualização seja mais compreensível. A partir daí transformações poderão ser realizadas nesse arquivo, já que ele passou a ser “entendido” pelo framework.

Todas essas etapas são realizadas diretamente no terminal, um arquivo que executasse essas etapas poderia ter sido criado para facilitar a execução. Entretanto, optou-se por não fazê-lo para que os passos ficassem bem definidos.

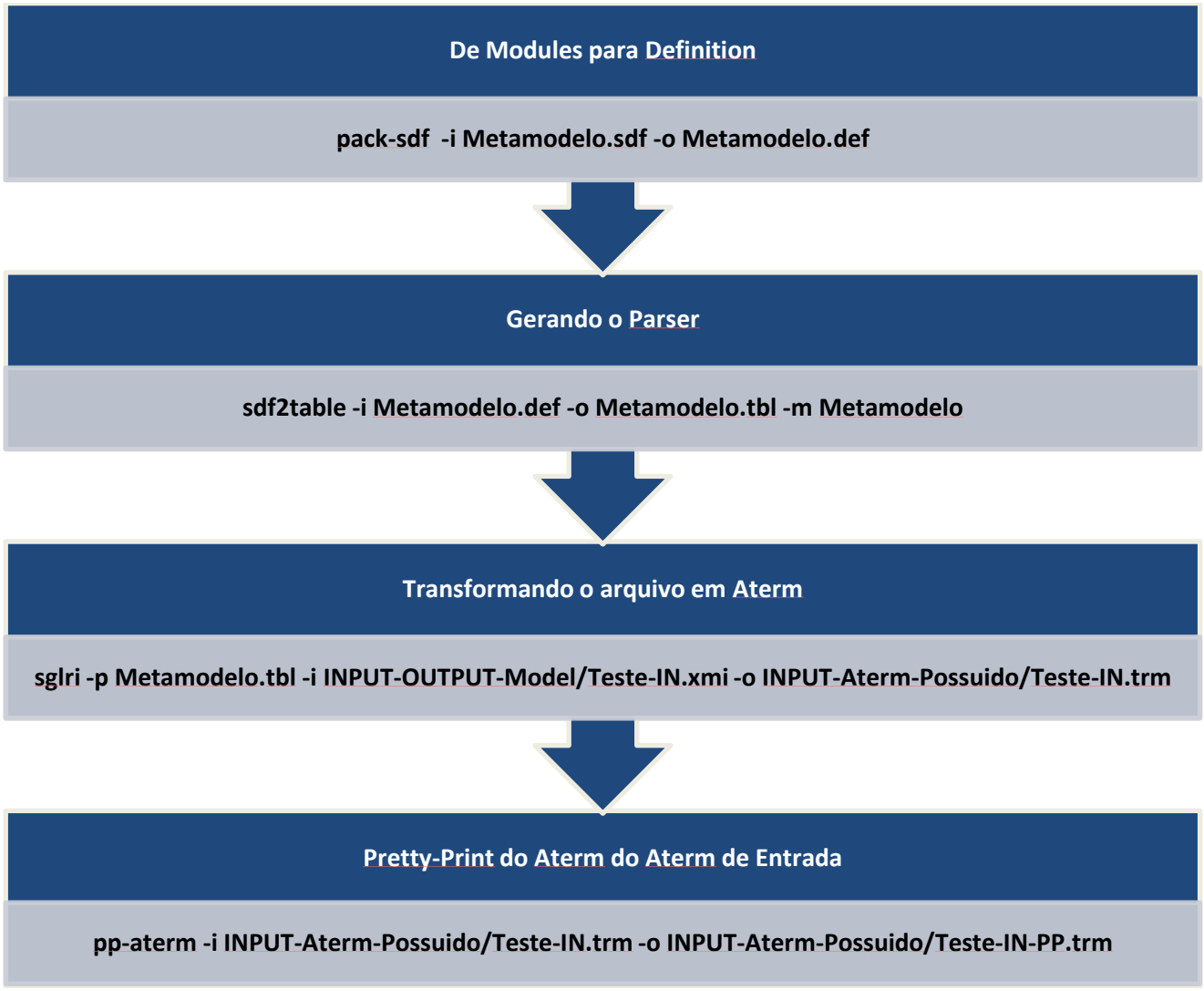


Figura 22 – Metodologia – Parte 1

A Figura 23 mostra a segunda etapa desse processo: primeiro uma *Regular Tree Grammar* é gerada – a partir da definição da gramática criada na etapa anterior –; essa árvore é criada para que se possa gerar um arquivo de assinaturas que seja compreendido pelos arquivos que codificam as transformações (regras e estratégias), esses arquivos possuem, inclusive, a mesma extensão str; o último passo, poderia ter sido executada antes da primeira, porque são independentes, mas o que ele faz é criar o *unparser* para que o ATerm, depois de transformado, possa retornar ao XMI.

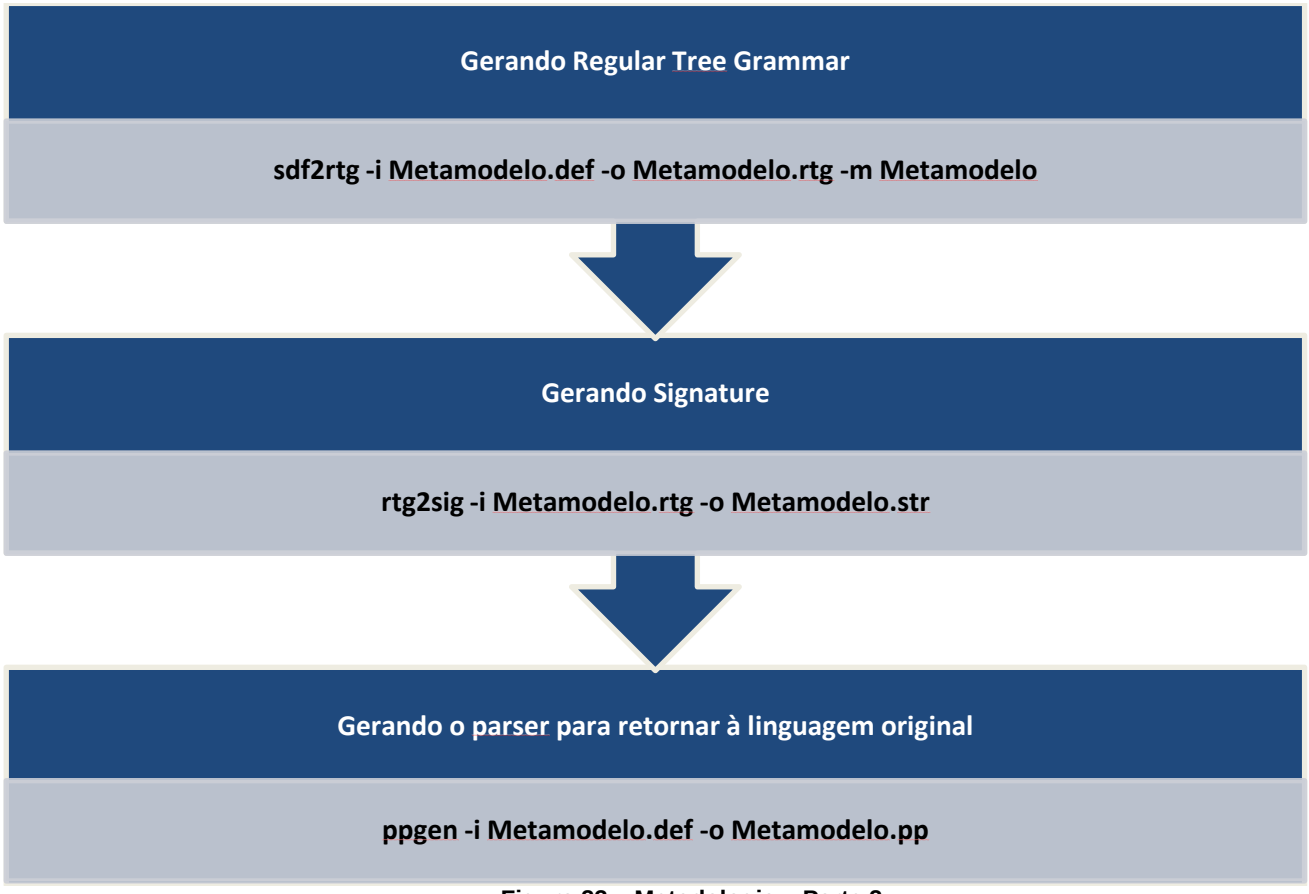


Figura 23 – Metodologia – Parte 2

A Figura 24 expressa a última etapa do processo: primeiro os arquivos que implementam as transformações são compilados; como resultado dessa compilação, a estratégia inicial implementada pode ser aplicada a algum arquivo de entrada (deve ser um ATerm); o passo seguinte é opcional, serve apenas para uma visualização mais estruturada do ATerm gerado como resultado; o último passo transforma o ATerm de saída em um arquivo do tipo XMI com o apoio do *unparser* criado na etapa anterior.

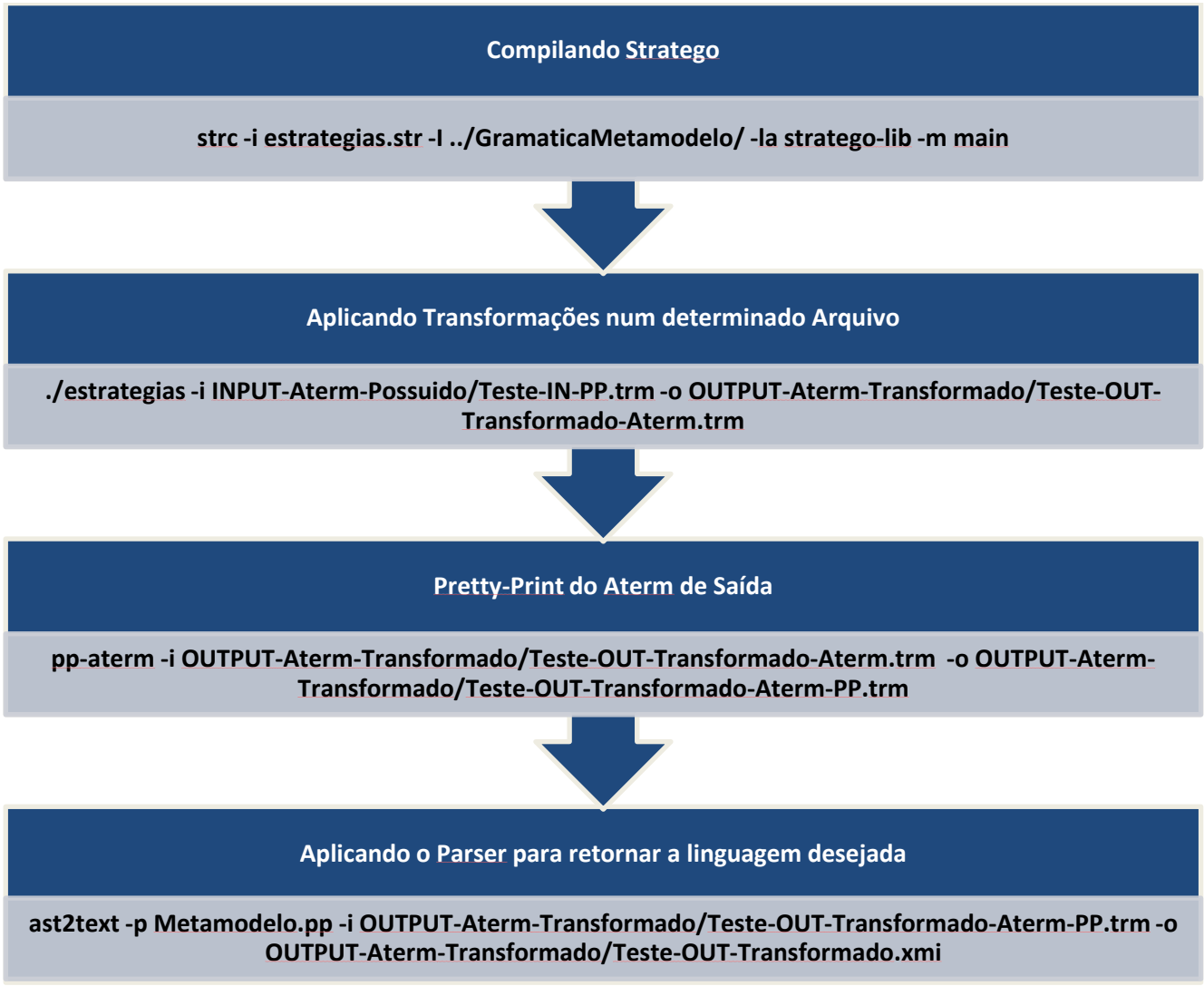


Figura 24 – Metodologia – Parte 3

Essa metodologia conta ainda com mais uma etapa que serviu, na realidade, para testar a validade das transformações. Essa etapa de "validação" utiliza passos já mostrados nas etapas anteriores, mas de maneira mais objetiva e com uma comparação dos arquivos. Essa validação é possível porque o trabalho de (Oliveira, 2008) disponibiliza arquivos de entrada (XMI) e arquivos de saída (XMI) depois da aplicação de uma determinada transformação. A Figura 25Figura 23 mostra os passos dessa etapa, o passo 1 transforma o XMI em ATerm; o passo 2 aplica a transformação e o passo 3 retorna ao formato do XMI. Para validação pode-se comparar um arquivo OUT transformado (depois do passo 3) com um arquivo OUT (um XMI já pronto) apenas *parseado* pelo passo 1. A mesma validação pode ser feita com os arquivos XMI que sofrem as transformações inversas.

	PASSO 1		PASSO 2		PASSO 3	
INPUT-OUTPUT--Models (IN-ORIGINAL.xmi)	--(sglri)-->	IN-Aterm-Possuido (Aterm.trm, PP.trm)	--./estrategias -->	OUT-Aterm-Transformado (OUT-Transformado.trm, PP.trm)	--ast2text-->	OUT-Transformado (OUT-Transformado.xmi)
INPUT-OUTPUT--Models (OUT-ORIGINAL.xmi)	--(sglri)-->	OUT-Aterm-Possuido (Aterm.trm, PP.trm)	--./estrategias -->	IN-Aterm-Transformado (IN-Transformado.trm, PP.trm)	--ast2text-->	IN-Transformado (IN-Transformado.xmi)

Figura 25 – Metodologia Validação

Ou seja, pode-se confrontar para constatar a validade das transformações os arquivos de IN-Aterm-Possuido VS. Out-Aterm-Transformado e os arquivos de OUT-Aterm-Possuido VS. IN-Aterm-Transformado.

6. Transformação com Stratego

6.1. Exemplo

A lei a ser descrita – Declarar Cápsula – é uma das leis básicas e estabelece quando é possível inserir uma nova cápsula no modelo do sistema. Apesar de não ser muito complexa, possui extrema importância porque uma cápsula só pode fazer parte da arquitetura do sistema modelado, caso ela tenha sido declarada previamente (Ramos, 2005).

Na realidade, a lei mostra quando é possível inserir e quando é possível remover, uma vez que todas as leis podem ser realizadas em ambos os sentidos. Como as condições de aplicação são diferentes, apenas a implementação no sentido da inclusão da uma cápsula será explicada.

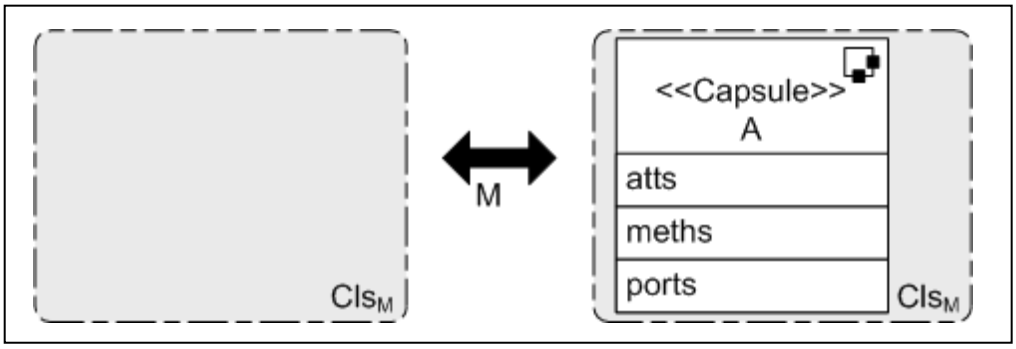


Figura 26 – Declarar Cápsula

A Figura 26 mostra como o modelo está antes da transformação e como fica depois dela, é interessante destacar que as imagens mostram apenas a parte do modelo que é relevante para a aplicação da transformação; ou seja, outros elementos podem estar presentes no modelo, mas o apenas o que será afetado é exibido. É por essa razão, que essa transformação não mostra nem o diagrama de estados, nem o diagrama de estrutura; pois a cápsula a ser criada não estará, ainda, associada a nenhum outro elemento do sistema, não fazendo parte de nenhum desses diagramas.

Como condição de aplicação da regra, é necessário que não haja nenhum elemento com o mesmo nome no mesmo pacote que a cápsula será inserida. Isso ocorre

porque em UML, e, consequentemente, em UML-RT não é permitido que dois elementos possuam o mesmo nome em um mesmo pacote.

Para cada transformação uma regra principal é criada com o nome dessa lei de transformação. A Figura 27 mostra o arquivo que contém essa regra; na realidade, essa lei de transformação está escrita em duas regras de Stratego com o mesmo nome. Essa é uma das características de Stratego: a possibilidade de criar regras com o mesmo nome; a ferramenta tenta aplicar as regras até alguma obter sucesso ou todas falharem.

```
module regras
imports libstratego-lib Metamodelo
strategies

  uniqueNameStrategy = not(try(UniqueName))

rules

  DeclareCapsuleModel:
  [Package(NamePackages(str1), [ListDefinitionEntity1*])]
  ->
  [Package(
     NamePackages(str1), [ListDefinitionEntity1*]
   , [DefinitionEntityBehaviouralCapsule(EntitiesBehaviouralCapsule(TypeBehaviouralCapsule(), NameEntity("\"Capsule_Introduced\"")))]
   )]
   where <uniqueNameStrategy> ("\"Capsule_Introduced\"")

  DeclareCapsuleModel:
  [Package(NamePackages(str1))]
  ->
  [Package(
     NamePackages(str1)
   , [DefinitionEntityBehaviouralCapsule(EntitiesBehaviouralCapsule(TypeBehaviouralCapsule(), NameEntity("\"Capsule_Introduced\"")))]
   )]
   where <uniqueNameStrategy> ("\"Capsule_Introduced\"")

   UniqueName:
    str -> str
```

Figura 27 – Regras - Declarar Cápsula

Para a implementação dessa transformação, primeiro foi definido o padrão que se deseja encontrar. Assim, baseado na geração pelo ATerm, formaliza-se o LHS da regra, neste caso queremos encontrar uma cápsula que tenha um nome qualquer. Para tornar esse padrão genérico, é necessário o uso de metavariáveis (Figura 28); nesse caso as metas variáveis que definem uma String (str1) e uma lista de elementos do tipo *DefinitionEntity* (ListDefinitionEntity1*) são fundamentais para que a regra de transformações sejam aplicadas independente do nome do pacote em que se deseja incluir a cápsula e independente dos outros elementos que a cápsula contém, respectivamente.

```
variables
 "srt" [0-9]*      -> String           {prefer}
 "ListDefinitionEntity"[0-9]* "*" -> DefinitionEntity* {prefer}
```

Figura 28 – Definiçaõ das variáveis - Declarar Cápsula

Há duas regras com o mesmo nome porque a segunda se aplica no caso em que a cápsula ainda está vazia e a primeira no caso em que há outros elementos na cápsula.

A cláusula condicional, *where*, utiliza uma nova estratégia criada: *uniqueNameStrategy*, que checa se um determinado nome – no caso “Capsula_Introduced” – já existe no termo atual; ou seja no pacote atual. Para isso, ela testa outra regra (*UniqueName*) que busca esse nome no termo, se existir retorna o

próprio termo, se não, falha; a estratégia *uniqueNameStrategy* inverte esse resultado (com o uso do *not*, predefinido por Stratego) e habilita ou não a realização da transformação.

É importante notar como informações do usuário seriam importante nesse contexto, para dizer qual o nome da cápsula que deseja criar; para dizer em qual dos possíveis muitos pacotes deseja incluir e para informar quantas cápsula deseja incluir. Esses elementos estão fixos nessa transformações.

Para efetivar a aplicação das regras, as estratégias são utilizadas. A Figura 29 mostra como é o processo de execução da regra. A primeira estratégia é indispensável para execução de qualquer transformação – *io-wrap* – porque é a responsável pela leitura de um arquivo de entrada e a escrita em um arquivo de saída. Em seguida, utiliza-se uma estratégia criada – "transformacaoA" – que determina quais as estratégias de passagem pelos nós do termo – *traversals* – e qual a regra de transformação a ser executada, neste caso a Declarar Cápsula (*DeclareCapsule*). A estratégia de navegação do termo utilizada é a *bottomup* eu lê os nós de cima para baixo. Além disso try é utilizado para que a busca pelo padrão continue até visitar todos os nós ou conseguir aplicar em algum deles.

```
module estrategias
imports libstratego-lib regras
strategies
  main = io-wrap(transformacaoA)
  transformacaoA = bottomup(try(DeclareCapsuleModel))
```

Figura 29 – Estratégias - Declarar Cápsula

6.2. Limitações e Dificuldades

Uma das principais limitações para a realização do trabalho é porque o framework, Stratego/XT, não disponibiliza o uso de maneira direta de parâmetros externos. As transformações são automáticas porque podem identificar padrões e gerar novos modelos automaticamente. Entretanto, no mundo real, algumas das transformações são definidas pelo usuário (Warmer, Jos. Kleppe, Anneke., 2003); ou seja, a interação com o usuário é indispensável, para dizer, por exemplo, quantos métodos devem ser gerados e em quais cápsulas. Se assim não fosse, regras que acrescentam elementos, por exemplo, teriam que ter alguma condição de parada, ou ficariam eternamente nessa geração.

Diante disto, o ideal é que um sistema mostre ao usuário quais transformações podem ser realizadas em um determinado contexto e ele selecione – incluindo parâmetros, se necessário – as desejadas, para a produção de um novo modelo.

A percepção que se teve é que Stratego funciona mais adequadamente para transformações que apenas traduzem algo em outra estrutura preexiste, sem que novas informações sejam adicionadas. É como se tudo que a transformação precisasse saber já estivesse presente no termo. Como não há o conceito de variável que pode armazenar o valor do termo em um determinado momento, houve muita dificuldade em se calcular a posição relativa de uma cápsula dentro de um pacote, por exemplo, e essa é uma

informação indispensável para a construção de novos relacionamentos entre os componentes de um modelo.

Outra dificuldade encontrada com a ferramenta é a falta de documentação; muitas vezes incompleta e, para algumas partes, a documentação inexiste. Por ser uma ferramenta relativamente nova (iniciou em 2000) há também poucos exemplos e poucos usuários ativos.

Stratego/XT é uma ferramenta muito poderosa no sentido que permite elevado nível de abstração, sendo possível fazer muitas operações com pouca codificação. Entretanto, ao mesmo tempo que essa característica pode ser vista como um benefício, também proporciona conseqüências negativas. Afinal a curva de aprendizado é muito maior e, diversas vezes, atividades que seriam facilmente realizadas por uma linguagem imperativa não conseguem ser implementadas de maneira direta.

Uma dificuldade para avançar mais rapidamente com o trabalho foi a necessidade de constantes mudanças na gramática, cada vez que algo era alterado todas as etapas de transformações precisam ser modificadas ou completamente refeitas e todo o processo explicado na metodologia precisava ser refeito. A lentidão para compilação e falta de informações objetivas nos erros também retardou o processo. A ferramenta ainda tem a desvantagem de não ser compatível com todas as plataformas e de ser bastante trabalhosa para instalação e ambientação.

Inicialmente houve a tentativa de se utilizar o XML-front que já vem no pacote básico da ferramenta, depois de diversas tentativas os resultados alcançados estavam muito aquém do desejado porque havia alteração na estrutura do XMI: o que era *tag* passava a ser conteúdo em outra *tag*.

Apesar das dificuldades a ferramenta apresenta um potencial muito grande e algumas soluções já foram pensadas para tentar solucionar os problemas mais críticos como a inserção de parâmetros. Mais informações na seção sobre trabalhos futuros, 7.1.

7. Conclusão

A metodologia de desenvolvimento MDA promete grandes benefícios especialmente no que diz respeito à portabilidade, por separar o conhecimento da aplicação do mapeamento para uma tecnologia específica; ao aumento da produtividade, por automatizar o mapeamento; à melhoria da qualidade, pelo reuso de padrões e práticas reconhecidas para o mapeamento; e à manutenção dos sistemas mais eficaz devido à melhor separação dos conceitos e rastreabilidade entre código e modelos.

Percebe-se que a MDA, na realidade, é um passo natural na história da programação. Desde a utilização de linguagens de máquina, perpassando pela idéia de pseudo-código, por linguagens procedurais e por linguagens orientadas a objetos; o objetivo é tornar o desenvolvimento algo mais próximo da realidade apreendida. Contemporaneamente, a busca pela possibilidade de desenvolver sistemas com linguagens de alto nível, com a conseqüente conversão dessas linguagens para que

possam ser interpretadas por um computador (responsabilidade dos compiladores e tradutores) continua, promovendo ainda mais a MDA.

Os modelos existem como uma abstração do sistema que é mais facilmente compreendida, no contexto atual de elevada complexidade se mostram ainda mais fundamentais. A MDA coloca o foco do desenvolvimento nos modelos que são tipicamente: CIM (modelo independente de computação), PIM (modelo independente de plataforma) e PSM (modelo específico de plataforma). Assim, a equipe pode se concentrar na elaboração dos modelos CIM e PIM, que é tipicamente o negócio do sistema; e as ferramentas realizam as transformações para a plataforma desejada e, na maioria dos casos, gera o próprio código a partir do PSM.

É importante relembrar que existem as transformações entre modelos em um mesmo nível; é o caso deste estudo que implementa *refectorings* para modelos PIM. Essas transformações também devem ser automatizadas, mesmo que haja alguns pontos que exijam interação com o usuário.

Para que as vantagens da MDA possam ser alcançadas, então, é necessário que ferramentas dêem o suporte para as transformações dos modelos de uma maneira automática e com credibilidade para as alterações que opera no modelo.

Muito já se avançou no sentido de produção de tecnologias que ofereçam esse suporte a MDA e de ferramentas que oferecem a possibilidade da construção dessas transformações. Algumas das mais difundidas são:

- QVT (*Query/View/Trasnformations*): é baseada em OCL e permite a definição de regras de transformações unidirecionais por meio da abordagem híbrida: declarativa e imperativa (Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 2008)
- ATL (Atlas Transformation Language)
- M2T (*MOF to Text Transformation*): define uma abordagem baseada em Templates para especificação das regras de transformação entre modelos baseados no MOF ou em texto;
- Kermeta: é uma linguagem imperativa e de metamodelagem que permite a descrição da estrutura e do comportamento de modelos. Foi desenvolvida para ser compatível com a linguagem de metamodelagem da OMG, EMOF (parte da especificação de MOF 2.0) e com Ecore, do Eclipse
- VIATRA (*Visual Automated Model*): com o auxílio de grafos cria relacionamentos entre os elementos dos modelos de origem e de destino; as regras são aplicadas de maneira não determinística e tem o objetivo de trasnformar os modelos de análise em modelos de projeto (Varró, D.; Varró, G., Pataricza, A., 2004)
- *MT Model Transformation Language*: uma derivação da QVT que permite a miscigenação de código no meio das regras de transformação [GReaT: é declarativa e baseada na transformação de grafos; oferece um elevado nível de abstração e maior confiabilidade ao processo de transformação; utiliza

C++ para a manipulação de atributos (Szemethy, T.; Karsai, G.; Balasubramanian, D., 2005).

Apesar do progresso ainda há muito o que ser feito, especialmente para o tipo de modelagem que suporta componentes ativos - tão presentes nos dias atuais - há um longo caminho a ser percorrido.

As transformações implementadas nesse trabalho atuaram exatamente neste tipo de modelagem. Tratando, especificamente, de modelos UML-RT; as transformações visam tornar a estrutura de um modelo do tipo PIM mais adequada tanto em termos de entendimento como da própria distribuição do sistema. Assim, são transformações que independem de tecnologia e quem mantêm o comportamento – estático e dinâmico – do sistema modelado.

A ferramenta escolhida para a implementação das transformações, Stratego/XT, foi selecionada tanto porque apresenta características desejadas num engenho de transformação; como porque, sendo uma ferramenta nova no mercado e com uma abordagem distinta do comum, o seu uso se mostrou como um desafio.

Constatou-se a importância dos padrões definidos pelo OMG: o MOF, UML e XMI, indispensáveis para a execução das transformações. Com a ferramenta percebe-se que Stratego/XT permite o reconhecimento de padrões de maneira prática; permite a implementação do inverso (entretanto as condições são a maior dificuldade); e que o ambiente é trabalhoso, mas muito poderoso.

Verifica-se que o projeto tem muita relevância porque permitiu se conhecer a possibilidade de promover transformações confiáveis em um sistema de software representado em UML-RT (e assim, potencialmente a representação de u sistema concorrente) com uma ferramenta inovadora que é Stratego. As regras automatizadas são apenas um subconjunto das leis definidas em (Ramos, 2005); entretanto essa abrangência pode ser estendida – com alguns ajustes, obviamente – pois constatou-se a possibilidade de facilitar a evolução de modelos de sistemas tão necessária no contexto de MDA.

Além disso, o trabalho conseguiu integrar diversas tecnologias/linguagens: MOF, UML, UML-RT, XMI, Stratego/XT, Eclipse, EMF para conseguir montar a estrutura de aplicação das transformações.

A construção da gramática; da metodologia e das transformações “pilotos” são as grandes conquistas desse trabalho e a concretização do seu objetivo. Entretanto, existem outros passos a serem dados para torná-lo aplicável num contexto real.

7.1. Trabalhos Futuros

Algumas adaptações precisam ser realizas para capturar detalhes mais específicos do modelo, como, por exemplo, a cerca da máquina de estados. A gramática ainda precisa sofrer alguns ajustes, especialmente no que diz respeito aos arquivos que definem as variáveis e o uso da sintaxe concreta, para que assim, novas transformações sejam realizadas.

Além disso, seria interessante existir a possibilidade de inclusão de parâmetros nas transformações, especialmente de maneira arbitrária. Uma solução já foi pensada: a partir de leituras de arquivos essas novas informações poderiam ser transmitidas, além de parâmetros a informação de qual transformação se deseja aplicar em determinado momento também seria enriquecedor. Stratego/XT lançou recentemente uma nova ferramenta –Spoofax – para uma integração maior com o Eclipse, certamente o uso dessa nova ferramenta tornaria mais ágil a implementação especialmente dos elementos da gramática.

Mais testes precisam ser aplicados, de preferência com o uso de ferramentas que gerem o XMI automaticamente. A ferramenta Rose-RT (mais difundida para modelagem dos sistemas concorrentes), entretanto, não exporta esse tipo de arquivo. A busca por outras ferramentas deve ser feita.

Para um mundo ideal, ainda, a aplicabilidade de uma ferramenta desse tipo deveria permitir que o desenvolvedor pudesse, manipulando modelos, criar transformações personalizadas e a ferramenta traduzisse isso numa transformação de modelos no nível de XMI.

A implementação de todas as leis tornaria possível a realização da normalização de modelos UML-RT. Esse processo visa provar que um sistema arbitrário, representado por diversos componentes isolados e ativos, pode ser, sem perder o seu comportamento, transformado num sistema com um único componente ativo. Numa perspectiva mais ampla, isso quer dizer que um sistema concorrente complexo pode ser modelado em um sistema simples semântica e comportamentalmente equivalente. Conseguir implementar todas as transformações seria ideal para a execução dessa normalização.

8. Bibliografia

Agrawal, A.; Karsai, G., Shi, F. (2002). Graph Transformations on Domain-Specific Models. *Institute for Software Integrated Systems, Vanderlbilt University* .

Allen, R. J. (30 de Agosto de 1997). *A Formal Approach to Software Architecture.* Acesso em 2010, disponível em http://www.cs.cmu.edu/~able/paper_abstracts/rallen_thesis.htm

Andrews, G. (Março de 1991). Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys* .

Bacelo, V; Lunelli, A. (2009.). Transformação de modelos em processos de desenvolvimento de software. (PUCRS, Ed.) *X Salão de Iniciação Científica* .

Booch, G.; Rumbaugh, J.; Jacobson, I. (2000). *UML, Guia do Usuário.* (F. F. Silva, Trad.) Rio de Janeiro: Campus.

Brand, M. v. (25 de Novembro de 2009). Meta-modeling and model trasnformations. Technische University Eindhowen.

Brown, A. (12 de Janeiro de 2004). *IBM.* (IBM, Ed.) Acesso em 08 de 2010, disponível em An introdution do Model Driven Architecture Part I: MDA and today's systems.: http://www.ibm.com/developerworks/rational/library/3100.html

Budinsky, F. (2004). *Eclipse Modeling Framework: A developer's Guide.* England: Addison Wesley.

Coulouris, G.; Dollimore, J.; Kindberg, T. (1994). *Distributed Systems: Concepts and Design.* Addison-Wesley.

Demuth, B., & Obermaier, S. (2000). Experiments with XMI Based Transformations of Software Models.

Eclipse. (2010). Acesso em Setembro de 2010, disponível em http://www.eclipse.org/

Eriksson, Han-Eril. Penker, Magnus. Lyons, Brian. Fado, David. (2004). *UML2 Toolkit.* Indianapolis, Indiana, EUA: Wiley.

Ferreira, P. M. (29 de Agosto de 2006). Geração Automática de Diagramas UML-RT a partir de Especificações CSP. *Dissertação de Mestrado* . Recife: Centro de Informática UFPE.

Fischer, C.; Olderog, E.; Wehheim, H. (1-6 de April de 2001). A CSP vies on UML-RT structure diagrams. (I. H. Hussmann, Ed.) *Fundamental Approaches to Software Engineering* .

Flore, F. (2003). MDA: The proof is in automating transformations between models. *Compuware Corporation* .

Freudenthal, M. (2010). Simplicitas: Software Architecture Document. *CYBERNETICA Institute of Information Security* .

Gasevic, Gragan; Djuric, Dragan; Devedzic, Vladan. (2006). *Model Driven Architecture and Onology Development.* Alemanha: Srpinger.

Godoi, R.; Ramos, R.; Sampaio, A. (2006). Uma Extensão do RUP para Modelagem Rigorosa de Sistemas Concorrentes. *XX Simpósio Brasileiro de Engenharia de Software* .

Gradt Booch, James Rumbaugh e Ivar Jacobson. (2000). *The UML Modeling Language User Guide.* Segunda Edição. Editora Campus Ltda.

Gu, Z.; Shin, K. (s.d.). Synthesis of Real-Time Implementation from UML-RT Models. (U. o. Michigan, Ed.)

Guimarães, E. G. (2008). Introdução aos Sistemas Distribuídos e Componentes de Software. *CTI Renato Archer* .

IBM. (2010). *IBM*. Acesso em 30 de Novembro de 2010, disponível em http://www-01.ibm.com/support/docview.wss?rs=64&context=SSSHKL&q1=XML&uid=swg21118055&loc=en_US&cs=utf-8&lang=en

Lapenda, Rodrigo; Madruga, Paulo; Loniewski, Grzegorz. (2000). Utilizando transformação de modelos para o desenvolvimento de software de qualidade .

Lengtel, L.; Lenvendovszky, T.; Mezei, G.; Forstner, B.; Hassan, C. (2005). Metamodel-Based Model Transformation with Aspect-Orientes Constraints. *Goldmann György* .

Lunelli, V. C. ; Bacelo, A. T. . (2009). Transformação de modelos em processos de desenvolvimento de softwar. *X Salão de Iniciação Científica PUCRS* .

Lyons, A. (1998). UML for Real-Time Overview.

Maranhão, R. G. (Setembro de 2005). Uma disciplina de Análise e Projeto para Aplicações Concorrentes, baseada no RUP. (C. d. UFPE, Ed.) *Dissertação de Mestrado* .

Medeiros, A. L. (Junho de 2008). MARISA-MDD: Uma abordagem para Tranformações entre Modelos Orientados a Aspectos: deos Requisitos ao Projeto Detalhado. Natal, RN.

Neto, E. L. (2008). *Automatizando Refatoramentos Arquiteturais em UML-RT utilizando Transformação de Modelos.* Centro de Informática UFPE.

Norton, D. (2006). View DSLs and UML as 'Fraternal Twins', Not Competitors. *Gartner Research* .

Oliveira, J. D. (2008). *Automação de Leis de Refatoração Arquitetural.* Centro de Informática - UFPE.

OMG. (12 de Junho de 2003). Acesso em 8 de 2010, disponível em OMG - MDA: http://www.omg.org/mda/

OMG. (Abril de 2008). Fonte: OMG.

OMG. (02 de Abril de 2003). *OMG.* Acesso em Agosto de 2010, disponível em Object Management Group: http://www.omg.org/cgi-bin/doc?formal/02-04-03

OMG. (2009). *OMG.* Acesso em 20 de Agosto de 2010, disponível em Object Management Group, Inc: http://www.omg.org/gettingstarted/what_is_uml.htm

Pellegrini, F.; Silva, F.; Silva, B.; Maciel, S. (2010). Transformações de Modelos para um Processo MDA. Bahia: Universidade Federal da Bahia.

Pereira, M.A.; Prado, A.F.; Biajiz, M.; Fontanette, V.; Lucrédio, D. (2006). Transformando modelos da MDA com apoio de Componentes de Software. (U. d. Paulo, Ed.)

Polido, M. F. (2007). *Um método de refinamento para desenvolvimento de software embarcado: Uma abordagem baseada em UML-RT e especificações formais.* São Paulo: Escola Politécnica da Universidade de São Paulo.

Ramos, R. (2005). Desenvolvimento Rigoroso com UML-RT. *Dissertação de Mestrado CIn-UFPE* .

Ramos, R.; A. Sampaio e A. Mota. . (2006). Transformation Laws for UML-RT. (S. - L. Science, Ed.) *IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems , 4037*.

Schots, Marcelo. Murta, Leonardo. Werner, Cláudia. (2009). PREVia: Uma abordagem para a representação visual da evolução de modelos arquiteturais de software. *XXIII SBES - WTES* .

Seidewitz, E. (2003). *What model mean* (Vols. vol. 20, no. 5). IEE Software.

Selic, B. (2003). *The pragmatics of model-driven development* (Vols. vol. 20, no. 5). IEE Software.

Selic, B.; Rumbaugh, J. (1998). *Using UML for Modeling Complex RealTime Systems.* Rational Software Corporation.

Selic, Bran; Rumbaugh, Jim. (11 de Março de 1998). Using UML for Modeling Complex Real-Time Systems.

Silva, D. M. (2001). *UML - Um guia de consulta rápida.* Editora Novatec.

Stahl, T.; Völter, M. (2006). *Mdel-Driven Software Development - Technology, Engineering, Management.* England: John Willey and Sons Ltda.

Stratego, G. (2010). *Stratego/XT*. Acesso em Agosto-Dezembro de 2010, disponível em Stratego/XT: http://strategoxt.org

Szemethy, T.; Karsai, G.; Balasubramanian, D. (2005). Model Transformation in the Model-Based Development of Real-time System. *Institure for Software Integrated Systems* . Nashville.

Tanenbaum, A. S. (1995). *Distributed Operating Systems.* Prentice-Hall.

Tenório, L. E. (27 de Fevereiro de 2003). Transformlets: Um framework para construção de trasnformadores de modelos MDA. *Dissertação de Mestrado* . Recife, Pernambuco.

Tockey, S. (2001). *OMG .* Acesso em 13 de 11 de 2010, disponível em Apresentações: http://www.omg.org/news/meetings/workshops/presentations/eai_2001/tutorial_monday/tockey_tutorial/1-Intro.pdf

Varró, D.; Varró, G., Pataricza, A. (2004). Generic and Meta-TRansformation for Model Trasnformation Engineering. *Procedings of the 7th Internatioal Conference on Unified Modeling LAnguage* . Lisboa.

Warmer, Jos. Kleppe, Anneke. (2003). *The Object Constraint Language* (Second Edition ed.). Boston, MA, EUA: Addison Wesley.

Warmer, Jos; Kleppe, Anneke; Bast, Wim. (2003). *MDA Explained: The Model.* Boston, MA, EUA: Addison Wesley.

Watson, A. (2010). *Visual Modelling: past, present and future.* Object Management Group.

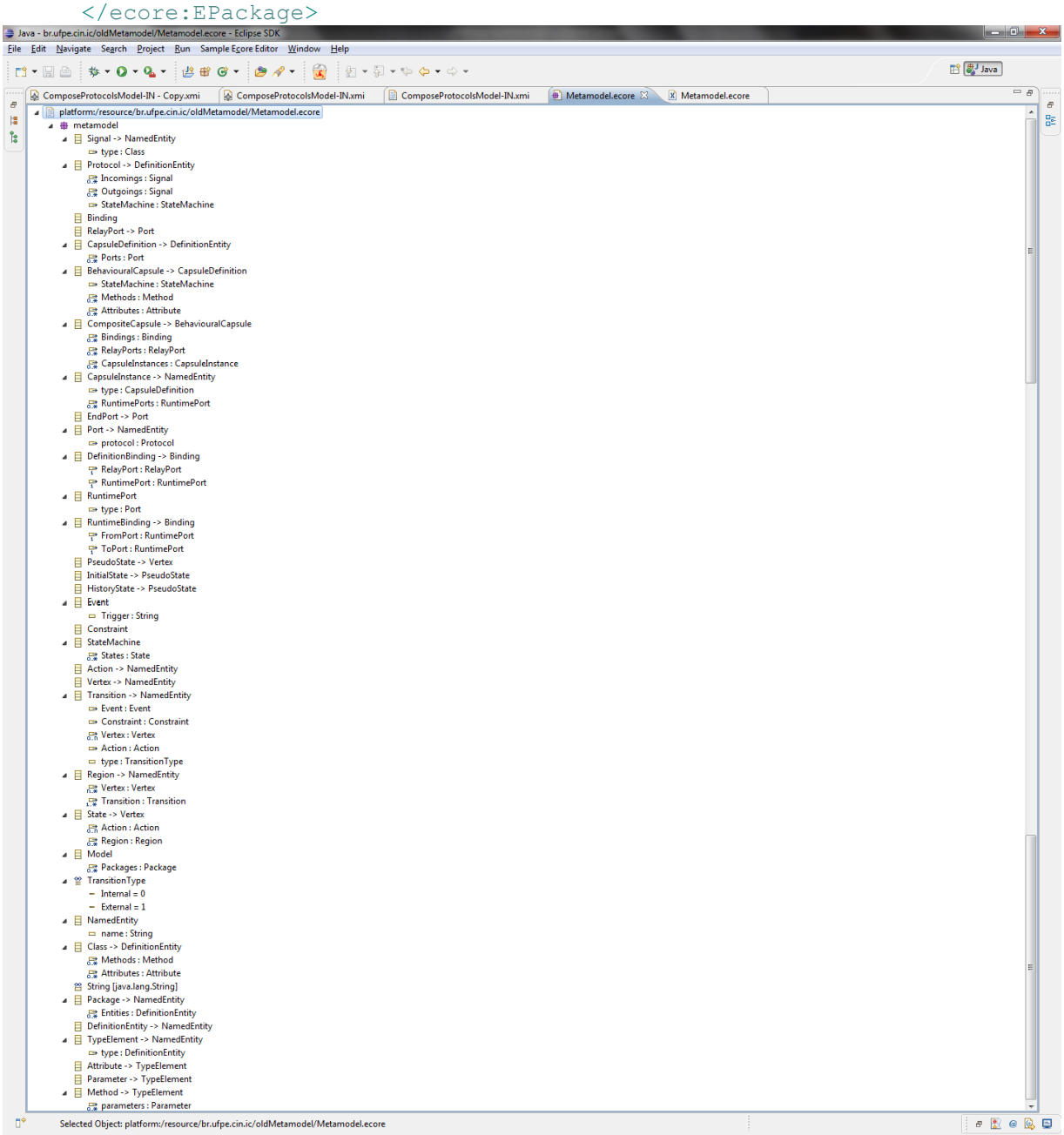
Anexo I – XMI do Ecore e Visão Estruturada

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage XMI:version="2.0"
    xmlns:XMI="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="metamodel"
    nsURI="platform:/resource/br.ufpe.cin.ic/oldMetamodel/Metamodel.ecore"
nsPrefix="metamodel">
  <eClassifiers xsi:type="ecore:EClass" name="Signal"
eSuperTypes="#//NamedEntity">
    <eStructuralFeatures xsi:type="ecore:EReference" name="type"
eType="#//Class"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Protocol"
eSuperTypes="#//DefinitionEntity">
    <eStructuralFeatures xsi:type="ecore:EReference" name="Incomings"
upperBound="-1"
      eType="#//Signal" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="Outgoings"
upperBound="-1"
      eType="#//Signal" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="StateMachine"
eType="#//StateMachine"
      containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Binding"/>
  <eClassifiers xsi:type="ecore:EClass" name="RelayPort"
eSuperTypes="#//Port"/>
  <eClassifiers xsi:type="ecore:EClass" name="CapsuleDefinition"
eSuperTypes="#//DefinitionEntity">
    <eStructuralFeatures xsi:type="ecore:EReference" name="Ports"
upperBound="-1"
      eType="#//Port" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="BehaviouralCapsule"
eSuperTypes="#//CapsuleDefinition">
    <eStructuralFeatures xsi:type="ecore:EReference" name="StateMachine"
eType="#//StateMachine"
      containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="Methods"
upperBound="-1"
      eType="#//Method" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="Attributes"
upperBound="-1"
      eType="#//Attribute" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="CompositeCapsule"
eSuperTypes="#//BehaviouralCapsule">
    <eStructuralFeatures xsi:type="ecore:EReference" name="Bindings"
upperBound="-1"
      eType="#//Binding" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="RelayPorts"
upperBound="-1"
      eType="#//RelayPort" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
name="CapsuleInstances" upperBound="-1"
      eType="#//CapsuleInstance" containment="true"/>
```

```
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="CapsuleInstance"
eSuperTypes="#//NamedEntity">
          <eStructuralFeatures xsi:type="ecore:EReference" name="type"
eType="#//CapsuleDefinition"
              resolveProxies="false"/>
          <eStructuralFeatures xsi:type="ecore:EReference" name="RuntimePorts"
upperBound="-1"
              eType="#//RuntimePort" containment="true"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="EndPort"
eSuperTypes="#//Port"/>
        <eClassifiers xsi:type="ecore:EClass" name="Port"
eSuperTypes="#//NamedEntity">
          <eStructuralFeatures xsi:type="ecore:EReference" name="protocol"
eType="#//Protocol"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="DefinitionBinding"
eSuperTypes="#//Binding">
          <eStructuralFeatures xsi:type="ecore:EReference" name="RelayPort"
lowerBound="1"
              eType="#//RelayPort"/>
          <eStructuralFeatures xsi:type="ecore:EReference" name="RuntimePort"
lowerBound="1"
              eType="#//RuntimePort"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="RuntimePort">
          <eStructuralFeatures xsi:type="ecore:EReference" name="type"
eType="#//Port" resolveProxies="false"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="RuntimeBinding"
eSuperTypes="#//Binding">
          <eStructuralFeatures xsi:type="ecore:EReference" name="FromPort"
lowerBound="1"
              eType="#//RuntimePort"/>
          <eStructuralFeatures xsi:type="ecore:EReference" name="ToPort"
lowerBound="1"
              eType="#//RuntimePort"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="PseudoState"
eSuperTypes="#//Vertex"/>
        <eClassifiers xsi:type="ecore:EClass" name="InitialState"
eSuperTypes="#//PseudoState"/>
        <eClassifiers xsi:type="ecore:EClass" name="HistoryState"
eSuperTypes="#//PseudoState"/>
        <eClassifiers xsi:type="ecore:EClass" name="Event">
          <eStructuralFeatures xsi:type="ecore:EAttribute" name="Trigger"
eType="#//String"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="Constraint"/>
        <eClassifiers xsi:type="ecore:EClass" name="StateMachine">
          <eStructuralFeatures xsi:type="ecore:EReference" name="States"
upperBound="-1"
              eType="#//State" containment="true"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="Action"
eSuperTypes="#//NamedEntity"/>
        <eClassifiers xsi:type="ecore:EClass" name="Vertex"
eSuperTypes="#//NamedEntity"/>
        <eClassifiers xsi:type="ecore:EClass" name="Transition"
eSuperTypes="#//NamedEntity">
```

```
          <eStructuralFeatures xsi:type="ecore:EReference" name="Event"
eType="#//Event"
              containment="true"/>
          <eStructuralFeatures xsi:type="ecore:EReference" name="Constraint"
eType="#//Constraint"
              containment="true"/>
          <eStructuralFeatures xsi:type="ecore:EReference" name="Vertex"
upperBound="2"
              eType="#//Vertex"/>
          <eStructuralFeatures xsi:type="ecore:EReference" name="Action"
eType="#//Action"
              containment="true"/>
          <eStructuralFeatures xsi:type="ecore:EAttribute" name="type"
eType="#//TransitionType"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="Region"
eSuperTypes="#//NamedEntity">
          <eStructuralFeatures xsi:type="ecore:EReference" name="Vertex"
lowerBound="2"
              upperBound="-1" eType="#//Vertex"/>
          <eStructuralFeatures xsi:type="ecore:EReference" name="Transition"
lowerBound="1"
              upperBound="-1" eType="#//Transition" containment="true"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="State"
eSuperTypes="#//Vertex">
          <eStructuralFeatures xsi:type="ecore:EReference" name="Action"
upperBound="2"
              eType="#//Action" containment="true"/>
          <eStructuralFeatures xsi:type="ecore:EReference" name="Region"
upperBound="-1"
              eType="#//Region" containment="true"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="Model">
          <eStructuralFeatures xsi:type="ecore:EReference" name="Packages"
upperBound="-1"
              eType="#//Package" containment="true"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EEnum" name="TransitionType">
          <eLiterals name="Internal"/>
          <eLiterals name="External" value="1"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="NamedEntity"
abstract="true">
          <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
eType="#//String"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="Class"
eSuperTypes="#//DefinitionEntity">
          <eStructuralFeatures xsi:type="ecore:EReference" name="Methods"
upperBound="-1"
              eType="#//Method" containment="true"/>
          <eStructuralFeatures xsi:type="ecore:EReference" name="Attributes"
upperBound="-1"
              eType="#//Attribute" containment="true"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EDataType" name="String"
instanceClassName="java.lang.String"/>
        <eClassifiers xsi:type="ecore:EClass" name="Package"
eSuperTypes="#//NamedEntity">
          <eStructuralFeatures xsi:type="ecore:EReference" name="Entities"
upperBound="-1"
```

```
                eType="#//DefinitionEntity" containment="true"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="DefinitionEntity"
eSuperTypes="#//NamedEntity"/>
        <eClassifiers xsi:type="ecore:EClass" name="TypeElement" abstract="true"
eSuperTypes="#//NamedEntity">
          <eStructuralFeatures xsi:type="ecore:EReference" name="type"
eType="#//DefinitionEntity"/>
        </eClassifiers>
        <eClassifiers xsi:type="ecore:EClass" name="Attribute"
eSuperTypes="#//TypeElement"/>
        <eClassifiers xsi:type="ecore:EClass" name="Parameter"
eSuperTypes="#//TypeElement"/>
        <eClassifiers xsi:type="ecore:EClass" name="Method"
eSuperTypes="#//TypeElement">
          <eStructuralFeatures xsi:type="ecore:EReference" name="parameters"
upperBound="-1"
                eType="#//Parameter" containment="true"/>
        </eClassifiers>
      </ecore:EPackage>
```



Anexo II – Gramática do Metamodelo

A gramática está como imagem porque facilita a visualização, por conta do layout; está dividida em duas partes. É importante lembrar que essa é apenas uma parte da gramática porque ainda faltam os elementos léxicos e as variáveis. Mais informações com a autora do texto que pode disponibilizar o conteúdo real.

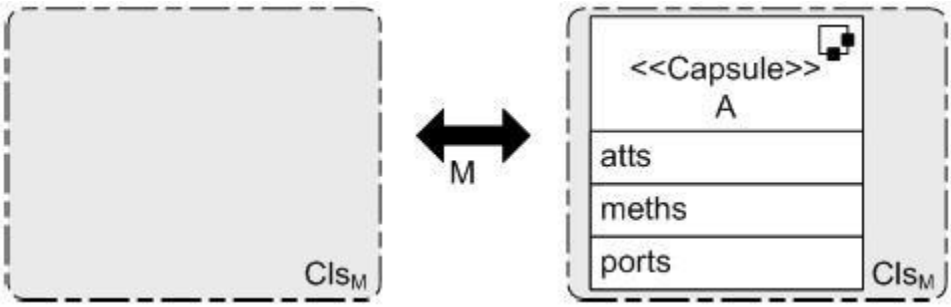
```
module Elements
imports
    Lexical

exports
  sorts XMLDocument Model TypeAttribute TypeCapsuleInstance TypeMethod Package NamePackages TypeProtocol EntitiesProtocol NameEntity
        EntitiesClass TypeClass DefinitionEntity EntitiesCapsuleDefinition TypeCapsuleDefinition EntitiesBehaviouralCapsule TypeBehaviouralCapsule
        EntitiesCompositeCapsule TypeCompositeCapsule Signal LigacaoClass NameSignal Attribute NameAttribute LigacaoTipoAtributo
        StringLigacaoTipoAtributo Method NameMethod LigacaoTipoRetorno StringLigacaoTipoRetorno Parameter NameParameters LigacaoTipoParametro
        StringLigacaoTipoParametro Port NamePort LigacaoProtocolo TypeRelayPort TypeEndPort CapsuleInstance NameCapsuleInstances RelayPort
        TypeDefinitionBinding LigacaoRelayPort LigacaoRuntimePort TypeRuntimeBinding LigacaoFromPort LigacaoToPort RuntimePort TypePort
        StateMachine State NameVertex Region NameRegion LigacaoVertex Action NameAction Transition NameTransition TransitionType Constraint Event
         Trigger StringLigacaoProtocolo Binding MetamodelType StringLigacaoClass
  context-free syntax

    "<?xml version=\"1.0\" encoding=\"ASCII\"?>" Model ->  XMLDocument {cons("XMLDocument")}

    "<metamodel:Model xmi:version=\"2.0\" xmlns:xmi=\"http://www.omg.org/XMI\" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
         xmlns:metamodel=\"platform:/resource/br.ufpe.cin.ic/oldMetamodel/Metamodel.ecore\">" Package* "</metamodel:Model>"-> Model {cons("Model")}
    "<metamodel:Model xmi:version=\"2.0\" xmlns:xmi=\"http://www.omg.org/XMI\" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
         xmlns:metamodel=\"platform:/resource/br.ufpe.cin.ic/oldMetamodel/Metamodel.ecore\"/>" -> Model {cons("ModelSElement")}

    "<Packages" NamePackages ">"DefinitionEntity* "</Packages>" -> Package {cons("Package")}
    "<Packages" NamePackages "/>" -> Package {cons("PackageSElement")}
    "name=" String -> NamePackages {cons("NamePackages")}

    "<Entities" TypeProtocol NameEntity ">" Signal* Signal* StateMachine? "</Entities>" -> EntitiesProtocol {cons("EntitiesProtocol")}
    "<Entities" TypeProtocol NameEntity "/>" -> EntitiesProtocol {cons("EntitiesProtocolSElement")}
    "xsi:type=\"metamodel:Protocol\""-> TypeProtocol {cons("TypeProtocol")}
    EntitiesProtocol -> DefinitionEntity {cons("DefinitionEntityProtocol")}
    "name=" String -> NameEntity {cons("NameEntity")}

    "<Entities" TypeClass NameEntity ">" Attribute* Method* "</Entities>" -> EntitiesClass {cons("EntitiesClass")}
    "<Entities" TypeClass NameEntity "/>" -> EntitiesClass {cons("EntitiesClassSElement")}
    "xsi:type=\"metamodel:Class\""-> TypeClass {cons("TypeClass")}
    EntitiesClass -> DefinitionEntity {cons("DefinitionEntityClass")}

    "<Entities" TypeCapsuleDefinition NameEntity ">" Port* "</Entities>" -> EntitiesCapsuleDefinition {cons("EntitiesCapsuleDefinition")}
    "<Entities" TypeCapsuleDefinition NameEntity "/>" -> EntitiesCapsuleDefinition {cons("EntitiesCapsuleDefinitionSElement")}
    "xsi:type=\"metamodel:CapsuleDefinition\""-> TypeCapsuleDefinition {cons("TypeCapsuleDefinition")}
    EntitiesCapsuleDefinition -> DefinitionEntity {cons("DefinitionEntityCapsuleDefinition")}

    "<Entities" TypeBehaviouralCapsule NameEntity ">" Port* StateMachine? Method* Attribute* "</Entities>"
    -> EntitiesBehaviouralCapsule {cons("EntitiesBehaviouralCapsule")}
    "<Entities" TypeBehaviouralCapsule NameEntity "/>" -> EntitiesBehaviouralCapsule {cons("EntitiesBehaviouralCapsulesElement")}
    "xsi:type=\"metamodel:BehaviouralCapsule\""-> TypeBehaviouralCapsule {cons("TypeBehaviouralCapsule")}
    EntitiesBehaviouralCapsule -> DefinitionEntity {cons("DefinitionEntityBehaviouralCapsule")}

    "<Entities" TypeCompositeCapsule NameEntity ">" Attribute* Method* Port* CapsuleInstance* RelayPort* Binding* "</Entities>" ->
          EntitiesCompositeCapsule {cons("EntitiesCompositeCapsule")}
    "<Entities" TypeCompositeCapsule NameEntity "/>" -> EntitiesCompositeCapsule {cons("EntitiesCompositeCapsuleSElement")}
    "xsi:type=\"metamodel:CompositeCapsule\""-> TypeCompositeCapsule {cons("TypeCompositeCapsule")}
    EntitiesCompositeCapsule -> DefinitionEntity {cons("DefinitionEntityCompositeCapsule")}

    "<Incomings" NameSignal LigacaoClass? "/>"-> Signal {cons("SignalIncoming")}
    "type=" StringLigacaoClass -> LigacaoClass {cons("LigacaoClass")}
    String -> StringLigacaoClass {cons("StringLigacaoClass")}
    "name=" String -> NameSignal {cons("NamingSignal")}

    "<Outgoings" NameSignal LigacaoClass? "/>"-> Signal {cons("SignalOutgoing")}

    "<Attributes" TypeAttribute? NameAttribute LigacaoTipoAtributo? "/>" -> Attribute {cons("Attribute")}
    "name=" String -> NameAttribute {cons("NamingAttribute")}
    "xsi:type=\"metamodel:Attribute\"" -> TypeAttribute {cons("XSIAttributeTyping")}
    "type=" StringLigacaoTipoAtributo -> LigacaoTipoAtributo {cons("LigacaoTipoAtributo")}
    String -> StringLigacaoTipoAtributo {cons("StringLigacaoTipoAtributo")}

    "<Methods" TypeMethod? NameMethod LigacaoTipoRetorno? "/>" -> Method {cons("MethodSElement")}
    "<Methods" TypeMethod? NameMethod LigacaoTipoRetorno? ">" Parameter* "</Methods>" -> Method {cons("Method")}
    "name=" String -> NameMethod {cons("NamingMethod")}
    "xsi:type=\"metamodel:Method\"" -> TypeMethod {cons("XSIMethodTyping")}
    "type=" StringLigacaoTipoRetorno -> LigacaoTipoRetorno {cons("LigacaoTipoRetorno")}
    String -> StringLigacaoTipoRetorno {cons("StringLigacaoTipoRetorno")}
```

```
"<parameters" NameParameters LigacaoTipoParametro? "/>" -> Parameter {cons("Parameter")}
"name=" String -> NameParameters {cons("NamingParameter")}
"type=" StringLigacaoTipoParametro -> LigacaoTipoParametro {cons("LigacaoTipoParametro")}
String -> StringLigacaoTipoParametro {cons("StringLigacaoTipoParametro")}

"<Ports" NamePort LigacaoProtocolo? "/>"-> Port {cons("Port")}
"name=" String -> NamePort {cons("NamingPort")}
"protocol=" StringLigacaoProtocolo -> LigacaoProtocolo {cons("LigacaoProtocolo")}
String -> StringLigacaoProtocolo {cons("StringLigacaoProtocolo")}

"<Ports" TypeRelayPort NamePort LigacaoProtocolo? "/>"-> Port {cons("PortRelay")}
"xsi:type=\"metamodel:RelayPort\""-> TypeRelayPort {cons("TypeRelayPort")}

"<Ports" TypeEndPort NamePort LigacaoProtocolo? "/>"-> Port {cons("PortEnd")}
"xsi:type=\"metamodel:EndPort\""-> TypeEndPort {cons("TypeEndPort")}

"<CapsuleInstances" TypeCapsuleInstance? NameCapsuleInstances? TypeCapsuleDefinition? ">"RuntimePort* "</CapsuleInstances>" -> CapsuleInstance {cons("CapsuleInstance")}
"<CapsuleInstances" TypeCapsuleInstance? NameCapsuleInstances? TypeCapsuleDefinition? "/>" -> CapsuleInstance {cons("CapsuleInstanceSElement")}
"name=" String -> NameCapsuleInstances {cons("NamingCapsuleInstance")}
"xsi:type=\"metamodel:CapsuleInstance\"" -> TypeCapsuleInstance {cons("XSICapsuleInstanceTyping")}
"type=" String -> TypeCapsuleDefinition {cons("BindingCapsuleDefinition")}

"<RelayPort" NamePort LigacaoProtocolo? "/>" -> RelayPort {cons("RelayPort")}

"<Bindings/>" -> Binding {cons("Binding")}

"<Bindings" TypeDefinitionBinding LigacaoRelayPort LigacaoRuntimePort "/>"-> Binding {cons("BindingTypeDefinition")}
"xsi:type=\"metamodelo:DefinitionBinding\""-> TypeDefinitionBinding {cons("TypeDefinitionBiding")}
"RelayPort=" String ->LigacaoRelayPort {cons("BindingRelayPort")}
"RuntimePort=" String -> LigacaoRuntimePort {cons("BindingRuntimePort")}

"<Bindings" TypeRuntimeBinding LigacaoFromPort LigacaoToPort "/>"-> Binding {cons("BindingRuntime")}
"xsi:type=\"metamodelo:RuntimeBinding\""-> TypeRuntimeBinding {cons("TypeRuntimePort")}
"FromPort=" String -> LigacaoFromPort {cons("BindingFromPort")}
"ToPort=" String -> LigacaoToPort {cons("BindingToPort")}

"<RuntimePorts" TypePort? "/>" -> RuntimePort {cons("RuntimePort")}
"type=" String -> TypePort {cons("BindingPort")}

"<StateMachine>" State* "</StateMachine>" -> StateMachine {cons("StateMachine")}
"<StateMachine/>" -> StateMachine {cons("StateMachineSElement")}

"<States" NameVertex ">" Region* Action? Action? "</States>"-> State {cons("State")}
"<States" NameVertex "/>" -> State {cons("StateSElement")}
"name=" String -> NameVertex {cons("NamingVertex")}

"<Region" NameRegion LigacaoVertex LigacaoVertex LigacaoVertex*">" Transition+ "</Region>" -> Region {cons("Region")}
"name=" String -> NameRegion {cons("NamingRegion")}
"Vertex=" String -> LigacaoVertex {cons("BindingVertex")}

"<Action" NameAction "/>"-> Action {cons("Action")}
"name=" String -> NameAction {cons("NamingAction")}

"<Transition" NameTransition TransitionType? LigacaoVertex? LigacaoVertex?">" Action? Constraint? Event? "</Transition>" -> Transition {cons("Transition")}
"<Transition" NameTransition TransitionType? LigacaoVertex? LigacaoVertex?"/>" -> Transition {cons("TransitionSElement")}
"name=" String -> NameTransition {cons("NamingTransition")}

"type=\"Internal\"" ->  TransitionType {cons("TransitionTypeInternal")}
"type=\"External\"" ->  TransitionType {cons("TransitionTypeExternal")}

"<Constraint/>" -> Constraint {cons("Constraint")}

"<Event" Trigger "/>"-> Event {cons("Event")}

"Trigger=" String -> Trigger {cons("Trigger")}

String -> MetamodelType {cons("MetamodelType")}
```

Anexo III – Leis de Transformações Implementadas

1. Declarar Cápsula

Esta lei estabelece quando é possível introduzir uma nova cápsula ao modelo. Observe que apesar de o contexto no lado esquerdo da lei estar vazio, o M subscrito fixa o contexto para a aplicação da lei.



Condições:

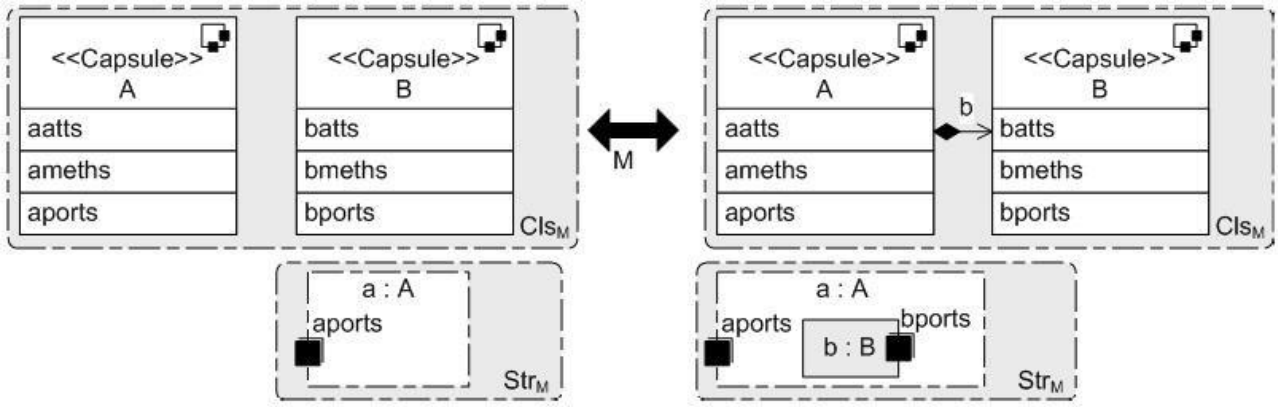
(→) Cls_M não possui a declaração de nenhum elemento, no mesmo pacote, chamado **A**.

(←) Nenhuma cápsula em **M** tem uma relação com a cápsula **A** em qualquer diagrama.

Usamos o símbolo (→) antes de uma condição para indicar que ela é requerida somente para aplicações da lei da esquerda para direita. Similarmente, utilizamos (←) para indicar que a condição é necessária somente para aplicações da lei da direita para a esquerda. CIsm representa a visão de diagrama de classes enquanto que Strm representa a visão de diagrama de estruturas.

2. Introduzir Associação Cápsula-Cápsula

Esta lei estabelece quando podemos adicionar ou remover uma associação entre cápsulas. Assumimos que, quando uma cápsula A é criada, é criado também seu diagrama de estrutura; este diagrama contém as instâncias de todas as cápsulas com as quais A possui uma associação.



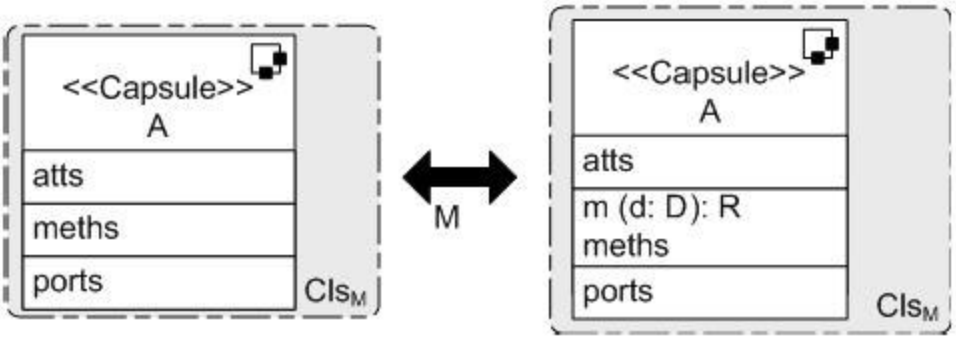
Condições:

(→) Não existe outra instância de cápsula chamada **b** no diagrama de estrutura de **A**.

(←) A instância **b** da cápsula **B** não está conectada a nenhuma outra na estrutura de **A**, inclusive à instância **a** que a contém.

3. Introduzir Método

Esta lei estabelece quando é permitido adicionar ou remover métodos em uma cápsula.



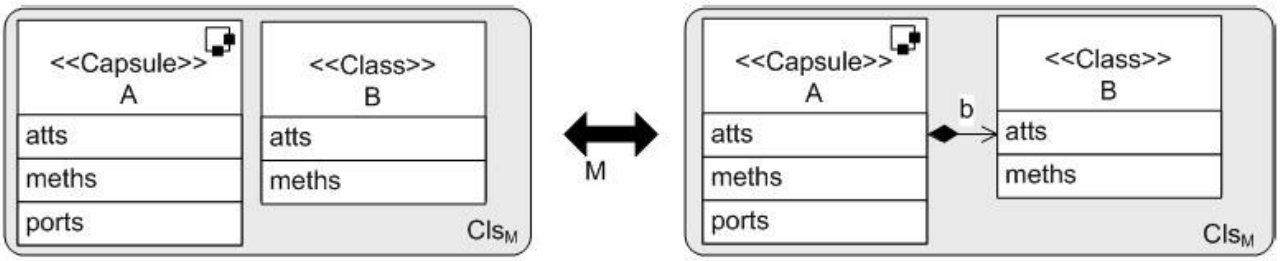
Condições:

(→) Não existe um método chamado m em A.

(←) O método m não é utilizado por outro método em meths, nem tampouco no diagrama de estados de A, ou em um predicado de A. Assumimos aqui que meths é o conjunto de métodos da cápsula A.

4. Introduzir Associação Cápsula-Classe

Esta lei introduz um atributo em uma cápsula, como conseqüência da criação de uma associação da cápsula com a classe. Assumimos que a introdução de uma associação b entre uma cápsula A e uma classe B introduz implicitamente um atributo b em A, por esta razão a existência de uma condição que indica que não pode haver outro atributo chamado b em A.



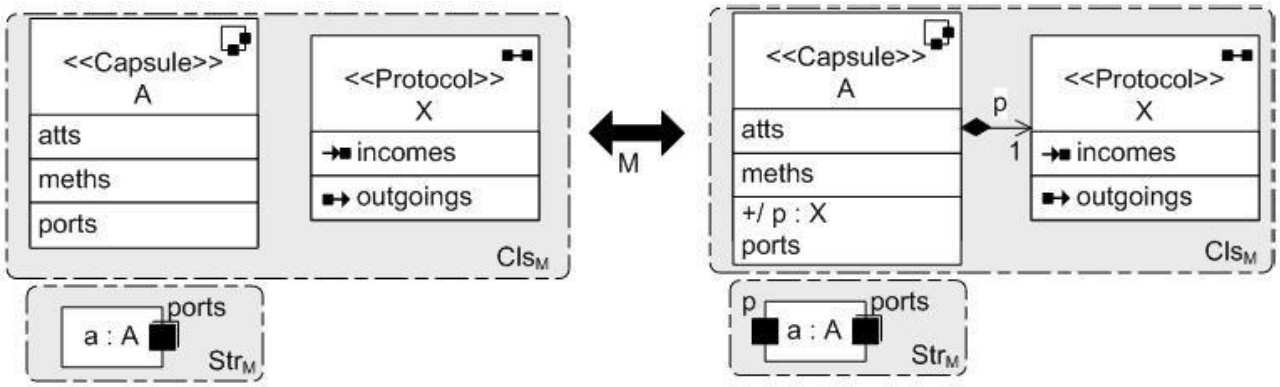
Condições:

(→) Não há um atributo chamado **b** em **A**.

(←) O atributo **b** não é utilizado por nenhum método, diagrama de estados ou predicado de **A**.

5. Introduzir Associação Cápsula-Protocolo

Esta lei estabelece quando podemos adicionar ou remover uma associação entre um protocolo e uma cápsula. Como conseqüência da introdução desta associação, é criada uma instância deste protocolo (porta) na cápsula.



Condições**:**

(→) Não existe outra porta chamada **p** na cápsula **A**.

(←) A porta **p** não é utilizada no diagrama de estados de **A**; não existe conexão ligada a **p** em Str_M.

Orientador

_____

Prof. Ph.D. Augusto Cesar Alves Sampaio

Aluna

_____

Camila Sá da Fonseca