UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TRANSFORMAÇÃO DE MODELOS SYSML PARA UML USANDO A LINGUAGEM ATL

MARCEL DA SILVA MELO

Uberlândia - Minas Gerais
2014

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MARCEL DA SILVA MELO

TRANSFORMAÇÃO DE MODELOS SYSML PARA UML USANDO A LINGUAGEM ATL

Dissertação de Mestrado apresentada à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software.

Orientador:
Prof. Dr. Michel dos Santos Soares

Uberlândia, Minas Gerais
2014

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Transformação de modelos SysML para UML usando a linguagem ATL**" por **Marcel da Silva Melo** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 19 de Agosto de 2014

Orientador: _____
Prof. Dr. Michel dos Santos Soares
Universidade Federal de Uberlândia

Banca Examinadora:

_____
Prof. Dr. Flávio de Oliveira Silva
Universidade Federal de Uberlândia

_____
Prof. Dr. Edson Alves de Oliveira Junior
Universidade Estadual de Maringá

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Agosto de 2014

Autor:	**Marcel da Silva Melo**
Título:	**Transformação de modelos SysML para UML usando a linguagem ATL**
Faculdade:	**Faculdade de Computação**
Grau:	**Mestrado**

_____

Autor

Dedicatória

Dedico este trabalhos a todos meus familiares, principalmente aos meus pais Celso Prudêncio de Melo e Maria Luzia da Silva Melo, a minha avó Orlanda Rodrigues da Silva e ao meu irmão Thiago da Silva Melo.

Agradecimentos

Primeiramente, agradeço à Deus por toda força e proteção que tem me dado durante toda a vida.

Agradeço à toda minha família que representa um porto-seguro onde posso contar sempre. Especialmente aos meus pais Celso Prudêncio de Melo e Maria Luzia da Silva Melo que sempre me proporcionam o apoio necessário em todas etapas da minha vida. Obrigado pela dedicação, apoio, amor e carinho. Um agradecimento especial também à minha avó Orlanda Rodrigues da Silva e ao meu irmão Thiago da Silva Melo por estarem sempre presentes em minha vida com muito amor e carinho.

Agradeço a minha namorada Bruna Andrade da Silva por estar sempre ao meu lado nos momentos bons e ruins desta caminhada. Obrigado pelo amor, companheirismo, amizade e compreensão.

Agradeço ao meu orientador Michel dos Santos Soares por ter acreditado e apoiado a realização desta pesquisa sempre de forma paciente e solícita. Obrigado pela confiança, profissionalismo e incentivo que sempre depositou neste trabalho. Certamente, sem sua dedicação e atenção este trabalho não teria sido possível.

Agradeço a todos meus amigos que sempre estiveram do meu lado nesta caminhada e que de forma direta ou indireta participaram para o sucesso deste trabalho.

Agradeço também aos meus amigos do mestrado e as grandes amizades que fiz no Instituto Federal do Triângulo Mineiro - Campus Patrocínio.

Por fim, agradeço a todos que contribuíram direta ou indiretamente para a realização e conclusão deste trabalho. Obrigado!

"Tente. E mesmo quando estiver cansado, tente. Um dia por teimosia, seus sonhos acontecerão."

(Ita Portugal)

Resumo

Devido ao grande aumento da complexidade no desenvolvimento de software nos últimos anos, academia e organizações criaram uma solução racional na engenharia de software, chamada de Engenharia Dirigida por Modelos, que busca suportar o gerenciamento de tal complexidade. A Engenharia Dirigida por Modelos é uma abordagem que move o foco do desenvolvimento de software de código para modelos. A UML é atualmente a linguagem mais utilizada para modelagem de software. Apesar do seu uso intenso em vários domínios de aplicação, a UML apresenta deficiências na modelagem em alguns domínios, como por exemplo *Software-Intensive Systems*, onde a modelagem de elementos que não são software é de grande importância. Uma das grandes vantagens da UML é a sua ampla capacidade de extensão e adaptação aos diferentes domínios de aplicação usando *profiles*, como é o caso da SysML. A SysML é um *profile* UML e representa uma linguagem de propósito geral usada no domínio de Engenharia de Sistemas.

Uma operação importante na Engenharia Dirigida por Modelos é a transformação de modelos, que consiste em um processo automatizado de conversão de um modelo origem para modelo destino. A construção de novas transformações, e o reuso das existentes, representam pontos-chave para a popularização da Engenharia dirigida por modelos. Este trabalho tem como objetivo apresentar os relacionamentos e as transformações automatizadas entre diagramas da SysML e diagramas da UML. Os relacionamentos são apresentados por meio de mapeamento entre metamodelos que apresentam as relações entre os elementos dos diagramas estudados. Os diagramas escolhidos para estudo foram o Diagrama de Blocos e Diagramas de Blocos Internos da SysML que são transformados em Diagrama de Classes e Diagrama de Atividades da UML, respectivamente. Uma abordagem orientada a modelos é usada para implementar essas relações como transformações de modelos automáticas. Para implementação destas transformações é usada a linguagem de transformação ATL. Dois estudos de casos reais, um para cada transformação implementada, são usados para validar as transformações.

Palavras chave: engenharia dirigida a modelos, sysml, uml, transformação de modelos, diagrama de blocos, diagrama de blocos internos, diagrama de atividades, diagrama de classes, atl, linguagem de transformação, engenharia de sistemas, engenharia de software

Abstract

Due to the large increase of complexity in software development in recent years, academia and organizations have a rational solution in software engineering, called Model-Driven Engineering, that seeks to support managing this complexity. Model-Driven Engineering is an approach that moves the focus of the development of software from code to models. The UML is currently the most widely used language for software modeling. Despite its extensive use in various application domains, UML is flawed in some domains, such as Software-Intensive Systems, where modeling elements that are not software are of great importance. A major advantage of UML is its wide extensibility and adaptation to different application domains using profiles, such as SysML. SysML is a UML profile and represents a general purpose language used in Systems Engineering domain.

One important operation in Model-Driven Engineering is model transformation, which consists of an automated process of converting a source model to target model. The construction of new transformations, and the reuse of existing ones, represent key points for popularization of Model-Driven Engineering. This work has objectives to present relationships and automated transformations between SysML diagrams and UML diagrams. Relationships are presented by means of metamodels that show relationships between elements of the diagrams studied. Diagrams chosen in the study were the Block Diagram and Internal Block Diagram of SysML that are transformed to Class Diagram and Activity Diagram of UML, respectively. A model-driven approach is used to implement these relationships as automatic model transformations. To implement these transformations the ATL transformation language is used. Two real case studies, one for each implemented transformation, are used to validate model transformations.

Keywords: model driven engineering, sysml, uml, model transformation, block diagram, internal block diagram, activity diagram, class diagram, atl, transformation language, systems engineering, software engineering

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Modelagem é uma atividade essencial para diversas atividades humanas. De fato, várias ações são precedidas pela construção (implícita ou explícita) de um modelo [Bézivin 2005]. Por exemplo, quando se idealiza a construção de um edifício, um planejamento detalhado pelo construtor é essencial, tendo por finalidade pensar sobre as várias formas de se construir, estimar o tempo e o custo, além dos materiais necessários para a realização do projeto.

Para o desenvolvimento de um sistema de software de qualidade é necessária a mesma preocupação, pois também reflete o problema abordado como uma questão de arquitetura e de ferramentas para satisfazer as restrições do cliente quanto a tempo, desempenho e custo.

Um modelo representa a realidade para um dado propósito. O modelo é uma abstração da realidade, no sentido de que não é possível representar todos os aspectos da realidade. Isso nos permite lidar com o mundo de forma simplificada, evitando a complexidade, o perigo e a irreversibilidade da realidade [Rothenberg 1989]. Modelos são abstrações de sistemas físicos que permitem raciocinar e entender o sistema, ignorando detalhes irrelevantes, enquanto focalizamos os mais relevantes [Booch et al. 2006] [Brown e McDermid 2007]. Essa simplificação (ou abstração) é a essência da modelagem [Booch 2004].

Na Engenharia de Software, modelos ajudam a visualizar o sistema como ele é, ou definir como ele será. Um benefício fundamental da modelagem é que ela proporciona um canal de comunicação entre os membros da equipe, diminuindo o risco de abstrair o conhecimento do sistema por meio de suas próprias opiniões pessoais. Assim, qualquer projeto pode ser beneficiado pelo uso da modelagem, pois os modelos auxiliam a equipe a ter uma visão mais detalhada e abrangente sobre o funcionamento do software, agilizando o processo de construção e auxiliando na precisão da abordagem efetuada.

A indústria de software vem sofrendo cada vez mais com a complexidade no desenvolvimento, manutenção e evolução de sistemas de software. Esta complexidade não se deve somente porque a quantidade de dados e código cresceu, mas também porque os sistemas de software passaram a ser distribuídos e implementados em diferentes plataformas. Com

o objetivo de fornecer uma solução racional para suportar o gerenciamento desta complexidade, academia e organizações propuseram abordagens baseadas em modelos para facilitar o desenvolvimento de software de qualidade.

A Engenharia Dirigida por Modelos (MDE, *Model Driven Engineering*) é uma abordagem que move o foco do desenvolvimento de software de código para modelos. A MDE provê benefícios tais como redução de custos e tempo no desenvolvimento de software e melhora a qualidade final dos produtos [Kent 2002]. A relevância dos modelos em MDE não consiste apenas em uma documentação do software desenvolvido. Eles podem ser compreendidos por computadores [Kent 2002] e são fundamentais para o desenvolvimento do software, pois podem ser manipulados, refinados e transformados para uma nova versão, e a partir deles o código-fonte é gerado [Kleppe et al. 2003]. A promessa da MDE é que o esforço necessário para o desenvolvimento e manutenção de software pode ser reduzido apenas por trabalhar no nível de modelos ao invés do nível de código. [Deursen et al. 2007]

Uma operação importante na MDE é a transformação entre modelos [Kleppe et al. 2003], que consiste em um processo automatizado de conversão de um modelo origem, utilizando uma linguagem de modelagem origem, para modelo destino, utilizando uma linguagem de modelagem destino [Kleppe et al. 2003]. O processo de transformação é a base da MDE, pois após sucessivas transformações o objetivo é alcançado. Os objetivos da transformação de modelos podem ser o refinamento de um modelo existente, acrescentando ou subtraindo informações [Calegari e Szasz 2013]; melhoramento do nível de abstração de um modelo; aumento da reutilização e da manutenção dos sistemas e testes; geração de código-fonte; ou mudança da linguagem de modelagem para melhoramento da análise, compreensão e/ou simulações dos modelos [France e Rumpe 2007] [Gerlich et al. 2007] [Feher e Lengyel 2012].

Para a construção de modelos, engenheiros de software dispõem de linguagens específicas de modelagem, como a UML e a SysML. A *Unified Model Language* (UML) [OMG 2011b] é uma linguagem de modelagem visual, orientada a objetos e de proposta geral para visualizar, especificar, construir e documentar projetos de software. Apesar da UML ser usada em vários domínios, como em sistemas de tempo real [Douglass 1997] e sistemas de controle de tráfego [K.Ranjini et al. 2011], e ser uma linguagem de proposta geral, ela não possui semântica suficiente para trabalhar com alguns domínios de aplicação por si só. Para isso a UML possui mecanismos para criar *Profiles*, que permitem adaptar seu metamodelo para se adequar às necessidades de um domínio de aplicação específico.

A *System Modeling Language* (SysML) [OMG 2012] é um *profile* UML e representa uma linguagem de propósito geral para a aplicação em Engenharia de Sistemas, com a intenção de unificar as diversas linguagens de modelagem utilizadas por engenheiros de sistemas. A engenharia de sistemas é mais ampla que a engenharia de software, assim uma linguagem com sua origem e aplicação primária baseada em software não adequa

todos aspectos de engenharia de sistemas [Willard 2007]. Por não ser utilizada apenas para a engenharia de software, a SysML aumenta a aplicação da UML para sistemas que não são puramente baseados em software, e pode ser aplicada em particular para projetar sistemas diversos. Segundo [Soares e Vrancken 2007], isso pode facilitar a comunicação entre times heterogêneos, por exemplo times compostos por engenheiros mecânicos, eletricistas e de software. A SysML é considerada uma evolução da UML 2.0 podendo ser aplicada a sistemas que podem incluir *hardware*, software, informações, processos, pessoas e instalações [OMG 2012].

Para implementação das transformações entre modelos em MDE, existem linguagens de domínio específico para escrita destas transformações [Czarnecki e Helsen 2003], como por exemplo, ATL [ATLAS 2006], QVT [OMG 2011a], MOLA, Fujaba, OptimaJ, SmartQVT e ETL. No trabalho de Biehl [Biehl 2010], várias outras linguagens de transformações são citadas, o que mostra uma dimensão do número de linguagens para transformações de modelos existentes e disponíveis para o usuário atualmente.

1.1 Motivação

De acordo com o trabalho de France e Rumpe [France e Rumpe 2007], pesquisas em transformação de modelos são de grande importância na MDE, destacando-se a criação de novas transformações como um tema chave para a popularização da MDE. Assim, encontram-se trabalhos propondo novas transformações para os mais diferentes objetivos e domínios de aplicação. Na literatura, nem sempre encontram-se trabalhos com transformações implementadas em alguma linguagem de transformação. Pode ocorrer de apenas a transformação ser apresentada e sua execução ser totalmente manual. Com o propósito de automatizar as transformações propostas neste trabalho, foi utilizada a linguagem de transformação ATL.

A UML é um padrão *de facto* para linguagem de modelagem utilizada na Engenharia de Software [Langer et al. 2014]. Devido sua popularidade, existem várias transformações que possuem como entrada modelos UML. Criar uma transformação de uma linguagem menos utilizada e que possui um número de transformações reduzido, como a SysML, para uma linguagem popular com grande número de transformações já propostas em vários domínios de aplicação, como a UML, permite aplicar de forma indireta qualquer transformação existente da UML nessa outra linguagem, aumentando o reuso e a aplicabilidade das transformações existentes.

Uma transformação de uma linguagem voltada para a engenharia de sistemas, como a SysML, para uma linguagem voltada para a engenharia de software, como a UML, pode melhorar a comunicação entre times heterogêneos formados por vários engenheiros de sistemas, como engenheiros eletricistas, mecânicos e mecatrônicos, engenheiros de software e desenvolvedores.

Não é difícil encontrar na literatura trabalhos que propõem novas transformações em vários domínios de aplicação. Porém, apenas um trabalho que visa transformar diagramas SysML em diagramas UML foi encontrado na literatura durante a execução deste trabalho. Tal trabalho, [Lasalle et al. 2011], busca transformar diagramas SysML para diagramas UML com o objetivo de gerar casos de testes de forma automática a partir de diagramas SysML. Devido aos poucos estudos que relacionam as linguagens SysML e UML e devido aos poucos trabalhos que combinam SysML e ATL usando abordagens baseadas em modelos é que este trabalho é proposto.

1.2 Objetivos

O objetivo principal deste trabalho é a criação de transformações *Model-to-Model* entre duas linguagens complementares, SysML [OMG 2012] e UML [OMG 2011b], mais especificamente entre Diagramas de Blocos e de Blocos Internos da SysML para Diagramas de Classes e de Atividades da UML. Inicialmente será criada uma transformação entre um diagrama estrutural da SysML, o Diagrama de Blocos, e um Diagrama Estrutural da UML, o Diagrama de Classes. Posteriormente, será criada outra transformação entre um diagrama estrutural da SysML, o Diagrama de Blocos Internos, e um diagrama comportamental UML, o Diagrama de Atividades. Para implementação das transformações propostas será utilizada a linguagem de transformação ATL.

A *Atlas Transformation Language* (ATL) [ATLAS 2006] foi escolhida por representar dois estilos de programação: declarativa e imperativa, ter sido projetada e implementada de acordo com padrão Object Constraint Language (OCL) [OMG 2014b], metamodelo *Meta Object Facility* (MOF) [OMG 2014a], ser uma linguagem de transformação com muitos recursos e gratuita, possuir documentação extensa, vários tutoriais, vídeos e exemplos úteis. A ATL possui módulos bens estruturados e escrita simples para transformações complexas, além de uma comunidade ativa que fornece valiosos *feedbacks*. [Jouault et al. 2008] [ATLAS 2006].

De forma geral, este projeto contempla os seguintes objetivos específicos:

- Definir o mapeamento entre os metamodelos do Diagrama de Blocos da SysML e do Diagrama de Classes da UML;
- Definir o mapeamento entre os metamodelos do Diagrama de Blocos Internos da SysML e do Diagrama de Atividades da UML;
- Implementar as transformações automáticas utilizando a linguagem de transformação ATL;
- Aplicar as transformações propostas em estudos de casos reais com o objetivo de avaliar a transformação;

1.3 Metodologia

A primeira etapa deste trabalho foi a realização de uma revisão da literatura com o objetivo de avaliar e interpretar pesquisas relevantes disponíveis para o tema proposto.

Após a realização do levantamento bibliográfico, que serve de suporte para outras etapas, foi necessário também um estudo aprofundado sobre a linguagem de transformação ATL [ATLAS 2006] e as linguagens de modelagem UML [OMG 2011b] e SysML [OMG 2012], principalmente os Diagramas de Classes e de Atividades da UML e os Diagramas de Blocos e de Blocos Internos da SysML, com o objetivo de aumentar o conhecimento das linguagens, e principalmente dos diagramas utilizados para realização do trabalho. Com o estudo concluído, foi possível a criação das transformações propostas utilizando a linguagem escolhida.

Após a criação e implementação de cada transformação proposta, foi realizado um trabalho de análise da aplicabilidade da transformação produzida. Para isso, foi realizado um estudo que buscou identificar os casos onde é possível realizar a transformação sem perda de informação e de forma automática. Nos casos onde não foram possíveis realizar toda a transformação de forma automática, foram identificados os casos onde é possível a realização da transformação de forma semi-automática que ainda mantém todas as informações relevantes do diagrama original.

A técnica de Estudo de Caso foi utilizada como instrumento de pesquisa. O estudo de caso é um método de pesquisa adequado para se investigar a ocorrência de um fenômeno em um contexto real e, portanto, possui um alto grau de realismo [Runeson e Höst 2009]. Estudos de casos não produzem o mesmo resultado que um experimento controlado, mas apresentam uma visão mais profunda do caso estudado. Juntamente com o estudo de caso, é realizada uma validação da transformação proposta. Segundo [Runeson e Höst 2009], o estudo de caso é um instrumento de pesquisa adequado para muitos tipos de pesquisas em engenharia de software.

De acordo com [Shaw 2003], esse trabalho é caracterizado como um procedimento por apresentar uma nova forma de realizar uma determinada tarefa e é caracterizado também como uma solução específica por ser uma solução para um problema de aplicação que possui o uso de princípios da engenharia de software na solução. Conforme as técnicas de validação também apresentadas por [Shaw 2003], como o trabalho é caracterizado por um procedimento, a validação para o trabalho é o uso de um exemplo baseado em um problema real acompanhado de uma explicação de porque o exemplo mantém a essência do problema que está sendo resolvido. Usando exemplos baseados em problemas reais, pode-se mostrar como e para quais tipos de problemas reais o procedimento funciona.

1.4 Trabalhos relacionados

Segundo Féher e Lengyel [Feher e Lengyel 2012], a importância da transformação de modelos tem se tornado cada vez mais aparente, tornando a transformação de modelos atualmente o centro das pesquisas da MDE. São encontrados na literatura alguns trabalhos que apresentam essa importância. No trabalho [Zhao e Zhang 2007], os autores apresentam uma revisão da atual pesquisa em MDE, focando principalmente em transformação de modelos. O artigo apresenta as abordagens existentes e ferramentas que apoiam a transformação, além de classificar as abordagens existentes para realização das transformações e identificação das questões para pesquisas futuras. Uma das questões que os autores destacam no trabalho é a automatização da transformação de modelos.

No trabalho [France e Rumpe 2007], os autores apresentam uma visão geral da pesquisa em MDE e discutem alguns dos principais desafios da área. Para isso, eles dividem os desafios em três grupos: desafios de linguagens de modelagem, desafios de separação de interesses e desafios de manipulação e gerenciamento de modelos. Um dos principais desafios destacados pelos autores na categoria de manipulação e gerenciamento de modelos é a definição, análise e usabilidade das transformações de modelos.

Pode-se notar que a área de transformações de modelos é um tema de grande importância e que possui várias pesquisas em andamento. Criar novas linguagens para transformação, analisar abordagens para a realização das transformações e criar novas transformações até então não existentes são alguns dos temas mais frequentes encontrados em pesquisas referentes a transformação de modelos. Em [Sendall e Kozaczynski 2003], os autores apresentam a importância da transformação de modelos e apontam essa operação como o coração e a alma da MDE. Como no trabalho de [Sendall e Kozaczynski 2003], o trabalho de [Czarnecki e Helsen 2003] busca qualificar tanto as abordagens para transformação de modelos quanto as próprias transformações. Já os trabalhos [Biehl 2010], [Czarnecki e Helsen 2006] e [Mens e Van Gorp 2006] buscam levantar fatores importantes tanto para classificar as transformações de modelos quanto as linguagens para realização destas transformações.

A criação de novas transformações é considerada um tema chave para a popularização da MDE. Assim, é fácil encontrar trabalhos que propõem novas transformações para os mais diferentes objetivos. Em alguns trabalhos são apresentadas apenas a transformação, não a implementando de forma automática. Em outros, são utilizadas linguagens de transformação, como ATL, para realização das transformações. A seguir alguns trabalhos que propuseram novas transformações de modelos são apresentados.

Em [Meng et al. 2010], os autores propõem uma transformação entre modelos DFD (*Data Flow Diagrams*) e diagramas de atividades da UML. É realizada uma análise dos elementos das duas linguagens, e baseado nas relações correspondentes de DFD e o diagrama de atividades é fornecida a transformação. Já no trabalho [Staines 2008], o autor

descreve como diagramas de atividades da UML 2 podem ser transformados em Redes de Petri coloridas. Tanto no trabalho de [Meng et al. 2010] quanto no trabalho de [Staines 2008] não foi utilizada nenhuma linguagem de transformação, ou seja, não foi realizada a transformação de forma automática. Os trabalhos apresentam apenas como é realizado o processo de transformação.

No trabalho [Salem et al. 2008], os autores têm como objetivo criar uma transformação de modelos entre *Graphe à Rèsultats et Activités Interliés (GRAI) extended Actigrams* e diagramas de atividades da UML utilizando três abordagens automatizadas diferentes e realizar o comparativo entre estas abordagens. Os autores também buscam apresentar a usabilidade da transformação de modelos no contexto de *Enterprise Modelling*. Para a realização das transformações, os autores definem um *profile* UML para evitar a perda de informações dos modelos GRAI quando transformados nos diagramas de atividades correspondentes. São utilizadas três linguagens de transformação com abordagens diferentes: ATL, MTF e Semaphore. Ao final, é feito um comparativo entre as abordagens utilizadas, exibindo pontos fortes e pontos fracos de cada abordagem.

No trabalho [Ngo e Soriano 2012], os autores apresentam uma transformação de *Real-Time Unified Modeling Language* (RT-UML) para SysML com o objetivo de criar controles de pilotos automáticos de navios como sistemas mecatrônicos integrados. Também na área mecatrônica, no trabalho [Qamar et al. 2009] os autores realizam a transformação entre Diagramas de Blocos da SysML para modelos Matlab/Simulink no intuito de simular comportamentos contínuos expressos em SysML no domínio específico do Matlab/Simulink. Os autores buscam também verificar como a transformação de SysML para MatLab/Simulink pode ser aplicada em sistemas mecatrônicos industriais.

Em [Bouquet et al. 2012], os autores propõem uma abordagem para transformar diagramas de Blocos, Blocos Internos e diagramas Paramétricos da SysML para código VHDL-AMS, com o objetivo de transferir modelos SysML para ambientes de simulação, para realização de verificações formais. Para isso, os autores utilizaram a linguagem de transformação ATL para realização das transformações *Model-to-Model* e a linguagem de transformação Xpand para a geração do código fonte.

No trabalho [Grobshtein e Dori 2009], os autores desenvolveram e aplicaram um algoritmo automatizado para transformar modelos *Object-Process Methodology* (OPM) em modelos SysML. O objetivo da transformação é permitir que usuários OPM compartilhem e apresentem seus modelos para outros *stakeholders*, que são familiarizados com a notação SysML, de uma maneira fácil e rápida. Outro beneficio é melhorar a análise e o entendimento do sistema. No trabalho, os autores além de apresentar a transformação, implementam uma ferramenta que realiza a transformação proposta. Esta ferramenta foi construída utilizando OPCAT, uma ferramenta de suporte de modelagem de OPM.

No trabalho [Paredis et al. 2010], os autores oferecem uma visão geral da transformação entre duas linguagens complementares, SysML e Modelica, no intuito de in-

tegrar o poder descritivo dos modelos da SysML com o poder analítico e computacional dos modelos da Modelica. Para realização da transformação, foi criado um *profile* SysML chamado SysML4Modelica. A transformação acontece entre os construtores da SysML4Modelica e os construtores da linguagem Modelica. No trabalho não é usada linguagem para automatizar a transformação entre os modelos. Porém, no site oficial do trabalho (http://www.omgwiki.org/OMGSysML), os autores apresentam o estado atual do trabalho, em que utilizam a linguagem de transformação ATL para automatização da transformação.

No trabalho [Anastasakis et al. 2007], os autores criam uma transformação de modelos UML para código da linguagem formal Alloy. O objetivo da transformação é explorar a capacidade de análise da linguagem Alloy em modelos UML. No trabalho os autores apresentam a transformação de Diagrama de Classes da UML para código Alloy sem usar nenhuma linguagem de transformação específica. Já no trabalho [Shah et al. 2010], os autores automatizam a transformação de UML para Alloy e criam a transformação contrária, de código Alloy para modelos UML.

Em [Lasalle et al. 2011], os autores buscaram criar uma transformação de diagramas SysML para diagramas UML com o objetivo de gerar casos de testes de forma automática a partir de diagramas SysML. Para isso, diagramas SysML foram transformados em diagramas UML e a partir da UML foram criados os casos de testes correspondentes. No trabalho foi criado o SysML4MBT, um *profile* da SysML para modelos baseados em testes, que será transformado em diagramas da UML4MBT, um *profile* da UML para modelos baseados em testes. Os casos de testes são gerados a partir dos modelos UML4MBT. A transformação foi implementada utilizando a linguagem JAVA.

1.5 Organização da dissertação

Este trabalho contribui para a pesquisa e prática em Engenharia de Software propondo novas transformações entre modelos SysML e UML e aplicando-as em estudos de casos de sistemas propostos na literatura. O restante do trabalho está organizado em 6 capítulos como descrito a seguir.

No Capítulo 2 é apresentada uma revisão teórica dos principais conceitos abordados neste trabalho. São apresentados os conceitos de modelos, engenharia dirigida a modelos, *Model Driven Architecture*, linguagens de transformação ATL e QVT.

No Capítulo 3 são apresentadas as linguagens de modelagem UML e SysML. Neste capítulo os diagramas participantes da transformação são abordados detalhadamente.

No Capítulo 4 é apresentada a transformação estrutural implementada usando a ATL. Neste capítulo é apresentado um mapeamento entre metamodelos que descreve a relação entre o Diagrama de Blocos SysML e o Diagrama de Classes UML. Posteriormente, uma transformação automática é implementada com base no mapeamento apresentado. Um

estudo de caso de uma aplicação prática no domínio de gestão de tráfego rodoviário é realizado para validação da transformação.

No Capítulo 5 é apresentada uma transformação dinâmica implementada usando a ATL. Um mapeamento entre metamodelos que descreve a relação entre o Diagrama de Blocos Internos SysML e o Diagrama de Atividades UML é apresentado. Logo após, uma transformação automática, utilizando a linguagem de transformação ATL, é implementada com base no mapeamento definido. Um estudo de caso extraído do site oficial da SysML é aplicado para validação da transformação.

No Capítulo 6 é realizada uma validação qualitativa onde o trabalho proposto é comparado a vários outros trabalhos semelhantes encontrados na literatura.

Por fim, o Capítulo 7 apresenta a conclusão referente ao trabalho, as contribuições propostas e os desafios que viabilizam trabalhos e pesquisas futuras.

Capítulo 2

Referencial Teórico

Neste capítulo são abordados os principais conceitos e tecnologias estudadas e utilizadas no desenvolvimento de toda a pesquisa. A Seção 2.1 apresenta uma visão geral da Engenharia Dirigida por Modelos e seus principais conceitos. A Seção 2.2 apresenta a *Model Driven Architecture*, um *framework* OMG que implementa a Engenharia Dirigida por modelos. Na Seção 2.3 é definido o conceito de linguagens de transformação, com um enfoque na linguagem de transformação ATL.

2.1 Engenharia Dirigida por Modelos

Modelos são abstrações de sistemas que permitem raciocinar e entender o sistema, ignorando detalhes irrelevantes no modelo, enquanto focalizamos os mais relevantes [Booch et al. 2006] [Brown e McDermid 2007]. Essa simplificação (ou abstração) é a essência da modelagem [Booch 2004]. Um modelo não se destina a captar todos os aspectos de um sistema, mas principalmente abstrair apenas algumas dessas características [Bézivin e Gerard 2002] com o intuito de simplificar a realidade [Muller e Gaertner 2004]. Um sistema é geralmente representado por um conjunto de modelos diferentes, cada um capturando aspectos específicos [Bézivin e Gerard 2002].

A Engenharia Dirigida por Modelos (*Model-Driven Engineering*, MDE) é uma proposta complementar aos processos de desenvolvimento tradicionais que se concentra em modelos, mapeamentos e transformações para apoiar o ciclo de vida do desenvolvimento de software [Lima et al. 2007] [Lopes 2008].

A Engenharia Dirigida por Modelos tem como enfoque utilizar modelos no centro dos processos de desenvolvimento de software. Na MDE, os modelos assumem o papel principal em todo ciclo de vida de um software. A relevância dos modelos vai além da documentação do software desenvolvido, eles podem ser compreendidos por computadores [Kent 2002] e são fundamentais para o desenvolvimento do software, pois podem ser manipulados, refinados e transformados para uma nova versão, até que a partir deles é gerado o código-fonte [Kleppe et al. 2003]. Entre os benefícios da utilização da MDE,

pode-se destacar a redução de custos e o tempo no desenvolvimento de software e a melhoria da sua qualidade final.

A utilização de modelos para o desenvolvimento de software não é algo novo e proposto pela MDE. Segundo Lopes [Lopes 2008], modelos estão sendo usados há algumas décadas para a concepção e projeto de software, sendo utilizados basicamente nas fases iniciais do desenvolvimento. O problema é que quando a codificação é iniciada, os modelos são esquecidos ou descartados, nem mesmo servindo como documentação, pois não há sincronia com o código final. A MDE traz uma nova proposta para a utilização dos modelos, que passam a ser utilizados como principais artefatos produzidos e manipulados em quaisquer fases do processo de desenvolvimento de software. Assim, cada modelo que é usado em uma fase pode ser projetado em outro modelo na fase subjacente, adquirindo ou omitindo novas informações. [Lopes 2008]

A MDE não pretende substituir os processos tradicionais de desenvolvimento de software [Lopes 2007] e sim contribuir para seu aprimoramento, porém, de uma maneira racional para mover informações de uma fase do desenvolvimento para outra, trazendo respostas rápidas e eficientes para atender as necessidades inesperadas de novos requisitos tanto funcionais quanto não-funcionais, apresentando rápida adaptação de novas tecnologias e integração de software desenvolvidos em diversas plataformas.

2.1.1 Modelos e Metamodelos

Em [Kleppe et al. 2003], um modelo é definido como uma descrição de um (ou de uma parte de) sistema expresso em uma linguagem bem definida, isto é, respeitando um formato preciso (uma sintaxe) e um significado (uma semântica). Esta descrição deve ser conveniente para uma interpretação automatizada por computadores. Assim, para criar um modelo devemos seguir uma sintaxe precisa com uma semântica bem definida afim de regulamentar a criação de elementos e suas relações. Este formalismo é resolvido utilizando uma linguagem de modelagem.

Uma linguagem de modelagem é uma especificação que contém os elementos de base para construir modelos, concebida dentro de um domínio limitado e com objetivos específicos. Várias linguagens de modelagem, cada uma delas adaptada a um domínio específico, podem existir. Uma linguagem de modelagem pode ser gráfica ou textual com notação matemática e deve permitir a definição de modelos sem ambiguidades. O uso de uma linguagem bem definida permite o entendimento e manipulação de modelos por pessoas e computadores [Lopes 2007].

Normalmente, uma linguagem de modelagem está em conformidade com um metamodelo [Mens e Van Gorp 2006]. Um metamodelo é um modelo que define a linguagem para exprimir um modelo [OMG 2011a]. A linguagem de modelagem é simplesmente um modelo do metamodelo. Ela define a estrutura, semântica e as restrições para uma família

de modelos [Mellor et al. 2004].

A Figura 2.1 mostra a arquitetura de quatro níveis de modelos da MDE. As relações entre os níveis descritos na Figura 2.1 são do tipo *ConformsTo* (em conformidade a) definido por Bézivin [Bézivin 2005]. Os quatro níveis são descritos a seguir de acordo com a definição de Mellor [Mellor et al. 2004]:

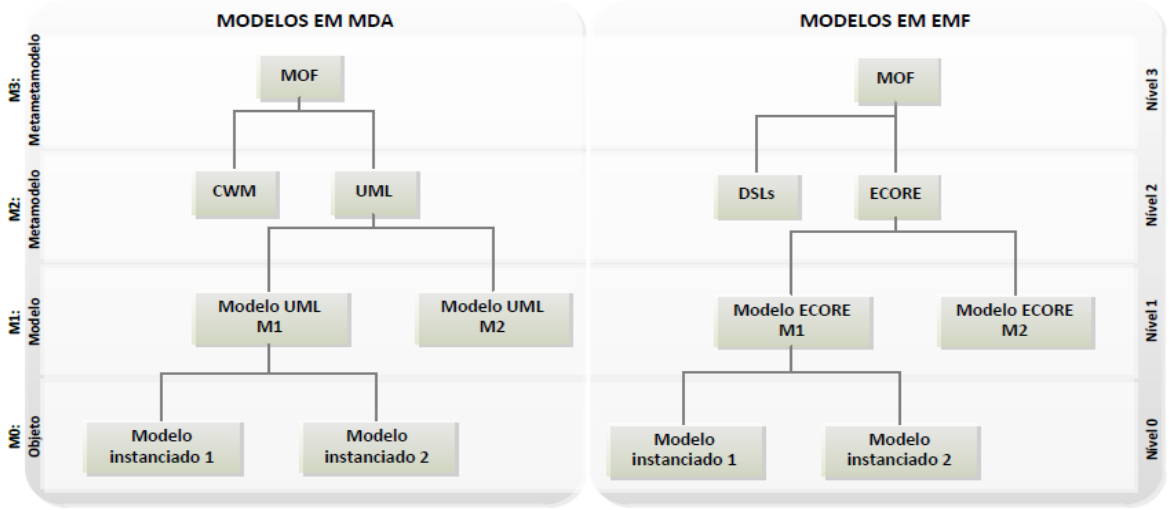

Figura 2.1: Arquitetura de 4 camadas. - adaptado de [Kleppe et al. 2003]

- M3 (metametamodelo): o nível M3 constitui a base da arquitetura de metamodelagem. A função primordial deste nível é definir linguagens para especificar metamodelos. Um metametamodelo define um modelo de mais alto nível de abstração que o metamodelo, e este primeiro é tipicamente mais compacto que o segundo. *Meta-Object Facility* (MOF) e *Eclipse Modeling Framework*(EMF) são exemplos de metametamodelos;
- M2 (metamodelo): um metamodelo está em conformidade a um metametamodelo. A função principal do nível de metamodelo é definir uma linguagem para especificar modelos. Os metamodelos são tipicamente mais elaborados que os metametamodelos. A linguagem UML possui um metamodelo que descreve estruturalmente como devem ser criados os modelos UML;
- M1 (modelo): um modelo está em conformidade a um metamodelo. A função principal do nível de modelo é definir uma linguagem para descrever um domínio da informação.
- M0 (informação): os objetos de usuários representam as informações finais. A principal responsabilidade dos objetos de usuários é descrever um domínio específico em uma plataforma final.

2.1.2 Transformação de Modelos

Uma operação importante na MDE é a transformação de modelos [Kleppe et al. 2003] [Jouault e Kurtev 2006], que consiste de um processo automatizado de conversão de um, ou vários, modelos origem para um, ou mais, modelos destino. Kleppe [Kleppe et al. 2003] define transformação de modelos como a geração automática de um modelo destino a partir de um modelo origem, de acordo com uma definição de transformação. Uma definição de transformação é um conjunto de regras de transformação que, juntas, descrevem como um modelo na linguagem de origem pode ser transformado em um modelo na linguagem de destino. Uma regra de transformação é uma descrição de como um ou mais elementos na linguagem de origem pode ser transformado em um ou mais elementos na linguagem destino. [Kleppe et al. 2003]

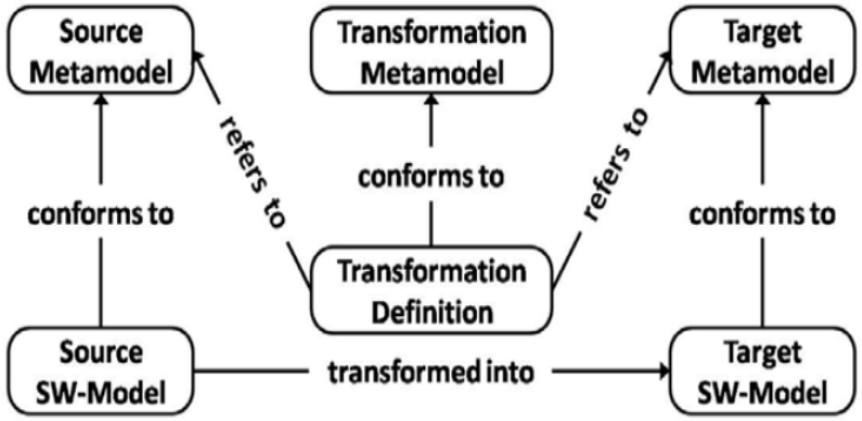

Figura 2.2: Transformação de modelos. - [Calegari e Szasz 2013]

De acordo com a Figura 2.2, uma transformação de modelos recebe basicamente como entrada um modelo *Source SW-Model* em conformidade com um metamodelo de origem *Source Metamodel*, e produz como saída outro modelo *Target SW-Model* em conformidade com um determinado metamodelo destino *Target Metamodel*. A transformação de modelos pode ser definida como um modelo *Transformation Definition* em conformidade com um metamodelo de transformação de modelos *Transformation Metamodel*. Este último metamodelo, junto com os metamodelos *Source SW-Model* e *Target SW-Model*, deve estar em conformidade com um metametamodelo, como o MOF [OMG 2014a]. A definição de transformação é executada por uma máquina de transformação [Calegari e Szasz 2013].

O processo de transformação é a base da MDE, pois após sucessivas transformações o objetivo é alcançado. Entre os principais objetivos da transformação de modelos podem ser destacados: o refinamento de um modelo existente, acrescentando ou subtraindo informações [Calegari e Szasz 2013]; geração de código-fonte; aumento ou diminuição do nível de abstração de um modelo; aumento da reutilização e da manutenção dos sistemas e testes; mudança da linguagem de modelagem para melhoramento da análise e compre-

ensão; e/ou realização de simulações nos modelos [France e Rumpe 2007] [Gerlich et al. 2007] [Feher e Lengyel 2012].

No trabalho de Mens e Van Gorp [Mens e Van Gorp 2006], Czarnecki e Helsen [Czarnecki e Helsen 2006] e Biehl [Biehl 2010], os autores buscam identificar e classificar as transformações de modelos. Algumas das classificações apresentadas por eles são citadas, de forma resumida, a seguir:

- Número de modelos origem e destino - A transformação pode utilizar mais de um modelo origem como entrada e/ou produzir vários modelos destino como saída;
- Vertical ou Horizontal - Os modelos de origem e de destino podem estar em um ou mais níveis de abstração. Uma transformação horizontal mantém modelos origem e destino no mesmo nível de abstração. Na transformação vertical, existe uma mudança de nível de abstração dos modelos. Esta mudança pode ser tanto para aumentar quanto para diminuir o nível de abstração;
- Transformações Endógenas ou Exógenas - Nas transformações Endógenas os modelos envolvidos são expressos na mesma linguagem de modelagem. Nas transformações Exógenas os modelos que participam da transformação são de linguagens diferentes;
- Bidirecionalidade - Uma transformação bidirecional pode tanto gerar modelos destino de modelos origem, quanto gerar modelos origem de modelos destino. Já em transformações unidirecionais, existe apenas um fluxo de execução.

Ainda, pode-se classificar uma transformação quanto aos tipos de modelos envolvidos na transformação. Uma transformação é chamada *Model-to-Model* quando a partir de um modelo origem é gerado um modelo destino ou *Model-To-Code* quando é gerado código-fonte em uma linguagem específica a partir do modelo origem. Como o código-fonte é visto como um modelo em MDE, pode-se visualizar a transformação de modelos *Model-To-Code* como um caso especial de transformações de *Model-to-Model*, sendo necessário apenas fornecer o metamodelo da linguagem de programação para a transformação [Czarnecki e Helsen 2003]. Uma transformação *Code-to-Model* ocorre quando é gerado um modelo a partir do código-fonte de uma linguagem específica. Outra definição, apresentada em [Calegari e Szasz 2013], é a transformação *Model-to-Text*, quando a partir de um modelo fonte é gerado texto sem conformidade com um metamodelo específico.

2.2 *Model Driven Architecture*

A *Model Driven Architecture* [OMG 2003] (MDA) é uma iniciativa desenvolvida pelo *Object Management Group* (OMG) com intuito de promover o uso de modelos no desenvolvimento de software, para fornecer uma solução ao gerenciamento da complexidade do

desenvolvimento, manutenção e evolução de sistemas de software e favorecer a interoperabilidade e portabilidade desses sistemas. De acordo com Favre [marie Favre 2004], o *framework* MDA é um caso particular da abordagem MDE.

O OMG define formalmente a MDA [OMG 2003] como uma abordagem que é bem-definida pela ideia de separar a especificação das operações de um sistema dos detalhes de como este sistema usa as potencialidades de sua plataforma. Isso possibilita que ferramentas ofereçam a especificação de um sistema independente de qualquer plataforma, bem como a transformação da especificação para uma plataforma particular. Os principais objetivos da MDA são a portabilidade, a interoperabilidade e a reusabilidade usando a separação arquitetural dos interesses.

Vários modelos em diferentes níveis de abstração são utilizados para que um sistema modelado possa ser transformado em software. O primeiro modelo a ser criado é o *Platform Idependent Model* (PIM). O PIM é um modelo com alto nível de abstração, que tem como objetivo definir o problema e expressar funcionalidades e restrições de negócio, não contendo referência a nenhuma plataforma ou tecnologia de implementação. O propósito desta independência é exibir apenas especificação que não varia com a alteração da plataforma, possibilitando o uso do mesmo PIM em múltiplas plataformas. Uma plataforma é um conjunto de subsistemas e tecnologias que proporcionam um conjunto coerente de funcionalidades usando interfaces e especificações padrões de uso [OMG 2003].

Após o PIM ser totalmente modelado, ele deve ser transformado em *Platform Specific Model* (PSM). O PSM é um modelo onde características de uma tecnologia de implementação específica são associados ao PIM. Assim, o PSM combina as especificações presentes no PIM com os detalhes relativos ao funcionamento do sistema em uma plataforma específica. Enquanto o PIM define a funcionalidade necessária, o PSM especifica como essa funcionalidade é realizada em uma plataforma específica. A partir dos PSMs o código-fonte do sistema é gerado.

Obviamente é necessária a definição de qual ou quais plataformas serão utilizadas na transformação, para então serem construídos os mapeamentos necessários para cada plataforma. Para gerar esses modelos PSM a partir de um mesmo PIM são necessárias diferentes regras de transformação, uma para cada plataforma escolhida. Vale ressaltar que não são necessárias alterações no modelo de negócio (PIM) para que possa gerar PSM's em diferentes plataformas. Apenas mudanças nas características do negócio forçarão a remodelagem do sistema no nível mais alto que é o PIM.

Pode-se notar que a transformação de modelos, definida na Seção 2.1.2, é a base da MDA. Para realização destas transformações são necessárias as definições de transformação e o motor de transformação. As definições de transformação descrevem como um modelo origem poderá se transformar em um modelo destino. A partir das definições de transformação, o motor de transformação é necessário para executar uma transformação [Kleppe et al. 2003]. Após várias transformações o código-fonte do sistema é gerado.

Para definição das transformações de modelos é usada uma linguagem específica para definir tais transformações. As linguagens de transformação de modelos são definidas na Seção 2.3. O processo de transformação é ilustrado na Figura 2.2 da Seção 2.1.2.

Um objetivo claro da MDA é fornecer um *framework* que integra os padrões existentes da OMG [Kent 2002]. Os principais padrões OMG utilizados na realização do trabalho são:

- *Unified Modeling Language* (UML) - É uma linguagem para especificação, construção, visualização e documentação de artefatos de software. Esta linguagem permite a modelagem de diferentes aspectos ou pontos de vista de um sistema [OMG 2011b]. A UML é abordada na Seção 3.1.
- *Meta Object Facility* (MOF) - Linguagem abstrata e um *framework* para especificação, construção e gerenciamento de metamodelos independentes de plataforma. Esta especificação contém um conjunto de construtores que é utilizado para a definição de metamodelos [OMG 2014a]. Em outras palavras, o MOF é uma linguagem simples para definição de outras linguagens.
- *XML Metadata Interchange* (XMI) - Padrão da OMG para troca de informações baseado em XML (*Extensible Markup Language*). Pode ser usado para trocar qualquer informação cujo metamodelo pode ser expresso usando MOF. O uso mais comum da XMI é como um formato de intercâmbio de modelos UML, embora também possa ser utilizado para a serialização de modelos de outras linguagens. [OMG 2014c].
- *Query/View/Transformation* (QVT)- É uma especificação híbrida padronizada para transformação de modelos no contexto da metamodelagem MOF. Esta linguagem aceita construções declarativas e imperativas [OMG 2011a].
- *Object Constraint Language* (OCL) - É uma linguagem declarativa para descrever regras que se aplicam aos modelos UML. A OCL, inicialmente, era apenas uma extensão da UML para especificações formais de modelos [OMG 2014b]. Hoje em dia, a OCL pode ser utilizada para especificar (pré-)condições de métodos e como linguagem de consulta, podendo ser usada em qualquer modelo cujo metamodelo seja MOF [Barendrecht 2010].
- *System Modeling Language* (SysML) - A linguagem SysML é uma linguagem de modelagem de propósito geral para aplicação em engenharia de sistemas. A SysML suporta a especificação, análise, projeto, verificação e validação de um amplo espectro de sistemas e sistemas de sistemas. A SysML é definida como uma extensão de um subconjunto da UML usando o mecanismo de perfis UML. A SysML é abordada na Seção 3.2.

2.3 Linguagem de transformação

Transformações de modelos desempenham um papel importante na abordagem MDE [Kleppe et al. 2003]. De acordo com Allilarire [Allilaire et al. 2006], é esperado que o desenvolvimento de transformações de modelos se torne uma tarefa comum no desenvolvimento de software utilizando a MDE. Mas para que isso ocorra, os engenheiros de software devem possuir ferramentas e técnicas bem definidas para auxiliar a desempenhar tal tarefa, da mesma forma que atualmente são auxiliados por IDEs clássicos, compiladores e depuradores em seu trabalho de programação diária [Allilaire et al. 2006].

Uma direção para solucionar este problema e prover o apoio necessário para a definição de transformações de modelos é o desenvolvimento de linguagens específicas especialmente projetadas para resolver esta tarefa [Allilaire et al. 2006]. A linguagem de transformação de modelos é usada para definir como um conjunto de modelos de origem é mapeado com o objetivo de criar um conjunto de modelos de destino. A linguagem define como as operações básicas sobre os modelos podem ser realizadas usando um conjunto específico de construções de linguagem (regras declarativas e sequências de instruções imperativas). [Bézivin et al. 2005]

A criação de linguagens específicas para transformação de modelos é uma abordagem que foi tomada pela indústria de software e comunidade de pesquisa. Como resultado, diversas linguagens de transformação de modelos têm sido propostas [Allilaire et al. 2006]. No trabalho de Biehl [Biehl 2010], são citadas várias linguagens de transformações, o que mostra uma dimensão do número de linguagens para transformações de modelos existentes e disponíveis para o usuário atualmente. Algumas das linguagens citadas são: ATL, QVT, EMF Henshin, SmartQVT, ModelMorf, Kermeta, ETL, OpenArchitecureWare (OAW), XML Stylesheet Language Transformations (XSLT), VIATRA, AndroMDA e Fujaba Transformations.

Nos trabalhos [Czarnecki e Helsen 2006], [Mens e Van Gorp 2006] e [Biehl 2010], os autores buscam levantar fatores importantes tanto para classificar as transformações de modelos, como discutido anteriormente, quanto as linguagens para realização destas transformações.

No trabalho [Huber 2008], o autor busca avaliar diferentes ferramentas e linguagens de transformação de modelos. O estudo conclui que nenhuma ferramenta é melhor do que a outra, mas que uma linguagem pode ser mais adequada para um problema específico do que outras linguagens.

Entre os vários fatores utilizados pelos autores na classificação das linguagens de transformação, o fator de maior relevância é quanto ao paradigma da linguagem. Segundo Mens [Mens e Van Gorp 2006], a maior distinção entre os mecanismos de transformação de modelos é quanto ao seu paradigma. Os principais paradigmas das linguagens de transformação, destacados em [Biehl 2010] e [Czarnecki e Helsen 2006], são descritos a

seguir:

- Imperativo - Linguagens imperativas especificam um fluxo de controle sequencial e fornecem meios para descrever a forma como a linguagem de transformação é supostamente executada [Mens e Van Gorp 2006]. Segundo Biehl [Biehl 2010], as construções e conceitos de linguagens de transformação imperativas são semelhantes as de linguagens de programação de propósito geral, como Java ou C/C++. A linguagem imperativa tem como vantagem oferecer um alto nível de controle para o programador, o que proporciona flexibilidade e permite implementações eficientes. A transformação é descrita como uma sequência de ações, o que é especialmente útil se a ordem de um conjunto de regras de transformação precisa ser controlada de forma explícita [Biehl 2010].
- Declarativo - Linguagens declarativas não oferecem um fluxo de controle explícito. Em vez de como a transformação deve ser executada, o foco é sobre o que deve ser mapeado pela transformação [Mens e Van Gorp 2006]. Segundo [Biehl 2010], transformações de modelos declarativas descrevem a relação entre os metamodelos de origem e de destino, onde esta relação pode ser interpretada bidirecionalmente. Como vantagem, as linguagens declarativas são, em geral, compactas e as descrições de transformação são geralmente curtas e concisas [Biehl 2010].
- Híbrido - Linguagens de transformação híbridas oferecem tanto as construções de linguagem imperativa quanto as construções de linguagem declarativa.
- Transformações por grafos - As linguagens de transformações por grafos são construídas sobre fundamentos teóricos das gramáticas algébricas de grafos e são uma subcategoria de linguagens declarativas. Transformações por grafos têm propriedades teóricas interessantes e são frequentemente utilizadas em provas e abordagens formais [Biehl 2010]. Nas transformações por grafos, os modelos são interpretados como grafos e as transformações são manipulações de subgrafos.
- Transformação Direta - Linguagens de programação de uso geral e bibliotecas para ler e gravar os dados dos modelos são usadas para implementar as transformações de modelos [Huber 2008]. A vantagem da transformação direta é que os programadores não precisam aprender uma nova linguagem. Mas por outro lado, as implementações tendem a se tornar maiores e insustentáveis [Biehl 2010].

2.3.1 ATL

A ATLAS *Transformation Language* (ATL) [ATLAS 2006] é uma linguagem de transformação *Model-to-Model* híbrida, ou seja, a linguagem contém uma mistura de construções declarativas e imperativas. O uso do estilo declarativo é encorajado por vários autores, [Allilaire et al. 2006] [Jouault e Kurtev 2006] [Jouault et al. 2008], pois permite uma

implementação mais objetiva e mais simples. No entanto, a definição de transformações complexas utilizando apenas construções declarativas pode ser uma tarefa difícil. Nesse caso, os desenvolvedores podem recorrer aos recursos imperativos da linguagem [Allilaire et al. 2006].

A ATL é uma linguagem de transformação de modelos que tem sua sintaxe abstrata definida usando um metamodelo. Isto significa que cada transformação ATL é de fato um modelo, com todas as propriedades que estão implícitas por isso [Bézivin et al. 2005]. A ATL é aplicada no contexto de transformações apresentada na Figura 2.3. Neste padrão, um modelo origem Ma é transformado em um modelo destino Mb de acordo com uma definição de transformação mma2mmb.atl escrita na linguagem ATL. A transformação definida em ATL é um modelo em conformidade com o metamodelo ATL. Todos os metamodelos pertencentes a uma transformação ATL devem estar em conformidade com o metametamodelo MOF [Allilaire et al. 2006].

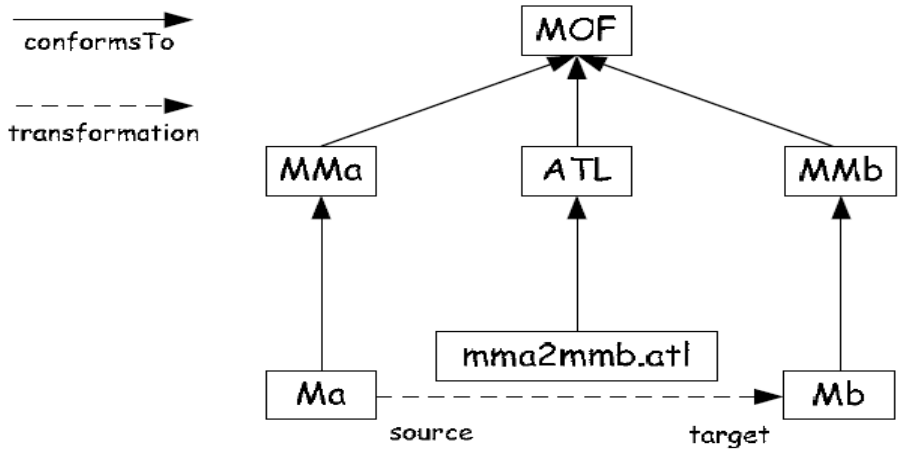

Figura 2.3: Transformação de modelos em ATL. - [Allilaire et al. 2006]

As transformações em ATL geralmente são baseadas na especificação de relações entre modelos origem e modelos destino. Este modo declarativo tende a ser mais próximo da maneira como os desenvolvedores intuitivamente percebem uma transformação. O estilo declarativo enfatiza a codificação dessas relações e esconde os detalhes relacionados com a seleção de elementos de origem, desencadeamento e ordenação de regras, lidando com a rastreabilidade. Portanto, a linguagem esconde atrás de algoritmos de transformações complexos, uma sintaxe simples e fácil [Jouault et al. 2008].

Transformações ATL são unidirecionais, operando somente na leitura de modelos de origem e produzindo modelos de destino somente para gravação. Durante a execução de uma transformação, o modelo origem pode ser navegado mas mudanças não são permitidas (*Read-only*). Já nos modelos destino, é permitida apenas a escrita (*Write-only*) e estes não podem ser navegados. Uma transformação bidirecional é implementada como um par de transformações: uma para cada direção. Modelos fonte e modelos alvo para a ATL

podem ser expressos no formato XMI da OMG. Os metamodelos de origem e de destino podem ser expressos também em XMI ou de forma mais conveniente usando a notação KM3 [Allilaire et al. 2006].

Uma transformação ATL pode ser decomposta em três partes: um *header*, *helpers* e *rules*. O *header* (cabeçalho) é usado para declarar informações gerais, tais como o nome do módulo (nome de transformação que deve coincidir com o nome do arquivo), os metamodelos de entrada e de saída e a importação de bibliotecas necessárias. Os *Helpers* são sub-rotinas que são usados para evitar a redundância de código. Pode-se imaginar um *Helper* para transformar um tipo visibilidade UML para um tipo visibilidade MOF em uma transformação de UML para MOF. Percebe-se a utilidade deste *Helper* quando sabe-se que todos os elementos da UML e MOF definem uma visibilidade. Já as *Rules* (Regras) são as principais definições das transformações ATL porque elas descrevem como elementos de saída (em conformidade com o metamodelo de saída) são produzidos a partir de elementos de entrada (em conformidade com o metamodelo de entrada). Elas são constituídas por ligações, cada uma expressando um mapeamento entre um elemento de entrada e um elemento de saída [Allilaire et al. 2006].

O código ATL é compilado e, em seguida, executado pelo mecanismo de transformação ATL. A ATL oferece suporte dedicado para rastreabilidade. A ordem de execução das regras é determinada automaticamente, com exceção das *Lazy Rules*, que precisam ser chamadas explicitamente. Os *Helpers* fornecem construções imperativas às transformações. A ATL não suporta transformação de modelos incremental, portanto, um modelo origem é lido completamente e o modelo destino é completamente criado. Alterações manuais no modelo destino não são preservadas. A ATL suporta um modo de transformação *in-place* (local), chamado de modo de refino. Ele tem limitações e não pode ser usado em combinação com certas construções, por exemplo, com *Lazy Rules* [Biehl 2010] [Jouault et al. 2008].

A ATL foi escolhida para implementação deste trabalho considerando vários aspectos. A ATL está integrada na plataforma Eclipse, o que provê uma série de recursos padrões para desenvolvimento (*syntax highlighting e debugger*). A ATL é parte do projeto M2M da ferramenta Eclipse (http://iot.eclipse.org/) e possui um grupo de discussão ativo, constante atualização, vários exemplos e diversos estudos de casos aplicados até mesmo na indústria [Selim et al. 2012]. Por se tratar de uma ferramenta de fácil uso [Salem et al. 2008], a partir da premissa de conhecimento da linguagem e do metamodelo, traz como vantagem ao processo: baixo custo, por ser uma ferramenta livre, e alta flexibilidade, por facilitar grandes alterações na transformação diretamente usando a interface do editor de regras ATL. A linguagem também é uma das tecnologias mais maduras no campo de pesquisa dirigida a modelos [Brunéliere et al. 2010].

Capítulo 3

UML e o *Profile* SysML

Neste capítulo são abordadas as linguagens de modelagem UML e SysML, estudadas e utilizadas no desenvolvimento da pesquisa e das transformações. Neste capítulo também são detalhados os diagramas participantes das transformações. A Seção 3.1 apresenta a linguagem de modelagem UML e os diagramas da UML utilizados nas transformações propostas neste trabalho: Diagrama de Classes e Diagrama de Atividades. A Seção 3.2 apresenta o *profile* SysML e os diagramas da SysML utilizados nas transformações propostas neste trabalho: Diagrama de Blocos e o Diagrama de Blocos Internos.

3.1 UML

A *Unified Modeling Language* (UML) [OMG 2011b] é uma linguagem gráfica para visualizar, especificar, construir e documentar os artefatos de sistemas de software. A UML oferece uma maneira padronizada para escrever projetos de um sistema, incluindo partes conceituais, tais como processos de negócios e funções do sistema, bem como partes concretas, tais como instruções de linguagem de programação, esquemas de banco de dados e componentes de software reutilizáveis.

A versão 2.4 da UML define quatorze diagramas que podem ser classificados e subdivididos em duas categorias: diagramas estruturais e diagramas comportamentais. Segundo o OMG [OMG 2011c], diagramas de estrutura definem a arquitetura estática de um modelo. Eles são usados para modelar as partes que compõem um modelo - as classes, objetos, interfaces e componentes físicos. Além disso, eles são utilizados para modelar as relações e dependências entre os elementos. Os diagramas comportamentais mostram o comportamento dinâmico dos objetos em um sistema, incluindo seus métodos, colaborações, atividades e histórico de estados do sistema de software. O comportamento dinâmico de um sistema pode ser descrito como uma série de modificações no sistema ao longo do tempo [OMG 2011c]. A Figura 3.1 apresenta uma visão geral dos diagramas UML, em seguida uma breve introdução de cada diagrama utilizado nesta dissertação é apresentada.

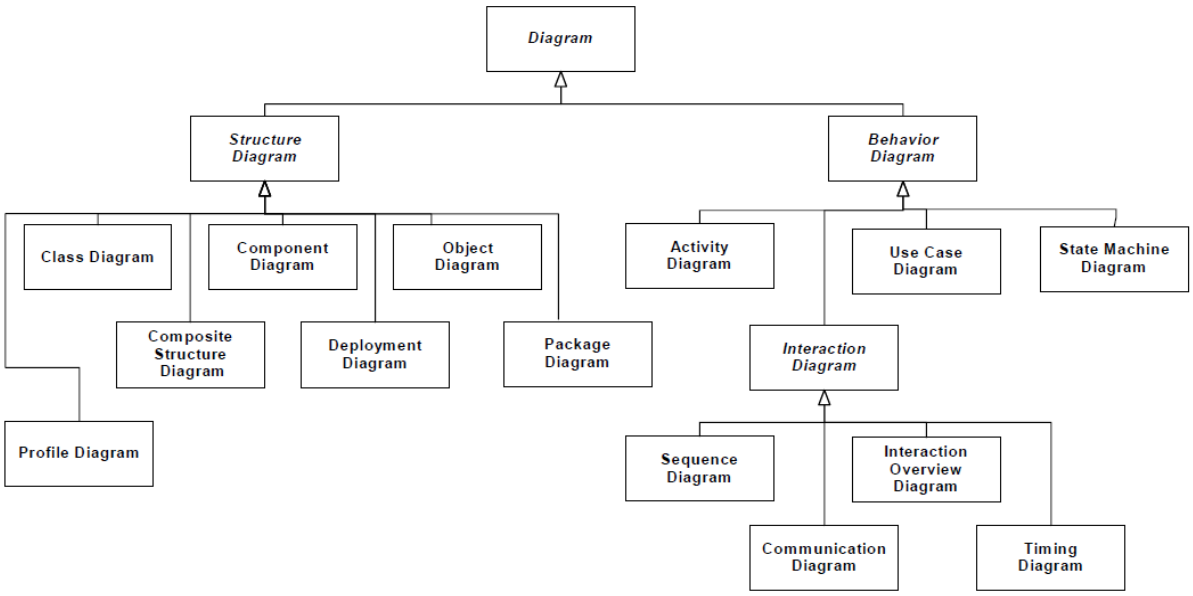

Figura 3.1: Hierarquia dos diagramas UML. - [OMG 2011c]

3.1.1 Diagrama de Classes

O Diagrama de Classes é o diagrama mais utilizado e o mais importante da UML, servindo de apoio para a maioria dos outros diagramas [Booch et al. 2006] [Guedes 2009]. Seu principal enfoque está em permitir a visualização das classes que irão compor o sistema com seus respectivos atributos e métodos, bem como demonstrar como as classes do diagrama se relacionam, complementam e transmitem informações entre si [Guedes 2009]. Este diagrama é utilizado para apresentar uma visão estática de como as classes estão organizadas, preocupando-se em como definir a estrutura lógica das mesmas [Guedes 2009]. A Figura 3.2 apresenta um diagrama de classes modelado para o domínio de um sistema de biblioteca integrada. Os principais elementos que compõem o diagrama de classes são apresentados a seguir.

O diagrama de classes é composto por suas classes e pelos relacionamentos entre elas, ou seja, os associações entre as classes [OMG 2011b]. Uma classe é uma descrição de um conjunto de objetos que compartilham os mesmos atributos, métodos e relacionamentos. Classes são blocos de construções mais importantes de qualquer sistema orientado a objetos [Booch et al. 2006]. Na UML, uma classe é representada por um retângulo que pode ter até três compartimentos. O primeiro armazena o nome pela qual a classe é identificada, no segundo são encontrados os atributos pertencentes a classe e o terceiro compartimento lista os possíveis métodos que a classe contém. Em geral, uma classe possui atributos e métodos, mas é possível encontrar classes que contenham apenas uma dessas características ou mesmo nenhuma delas.

Os atributos representam as características de uma classe, ou seja, as peculiaridades que costumam variar de um objeto para outro. Os atributos são apresentados no segundo

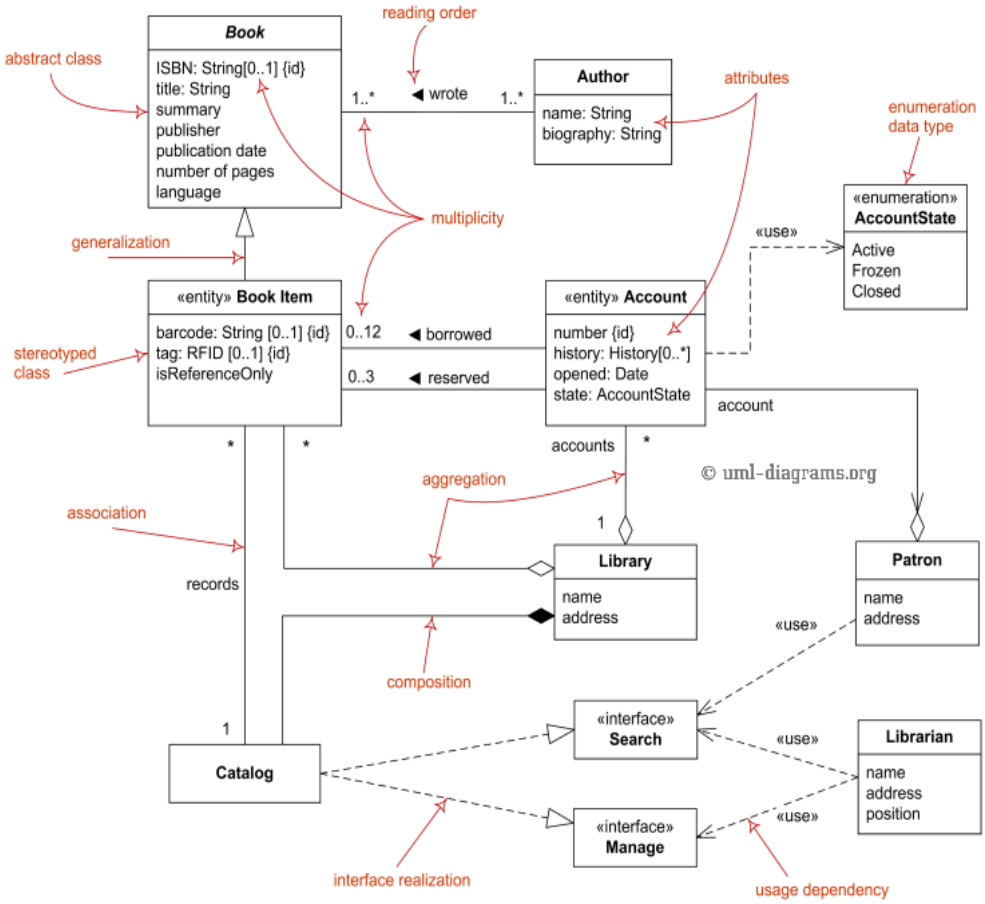

Figura 3.2: Exemplo de um diagrama de classes UML. - Extraído de http://www.uml-diagrams.org/

compartimento e contêm, normalmente, duas informações: nome que identifica o atributo e o tipo de dado que o atributo armazena, sendo que o tipo de dado não é obrigatório no diagrama de classes.

Um método representa uma atividade que um objeto de uma classe pode executar. Os métodos são apresentados no terceiro compartimento e contêm, normalmente, três informações: nome que identifica o método, uma lista de parâmetros necessários para execução do método, sendo representados como os atributos da classe, e o tipo de dado de retorno do método, caso exista.

A UML permite também representar a visibilidade de atributos e métodos usando notação gráfica. A visibilidade de atributos e métodos é aplicada antes da definição destes elementos. Os quatro modos de visibilidades, com sua representação em UML, são: público (+), protegido (#), privado (-) e pacote (∼).

Dificilmente uma classe é utilizada sozinha. Normalmente cada classe funciona em colaboração com outras, para a realização de alguma semântica maior do que cada uma delas individualmente [Booch et al. 2006]. Estes relacionamentos buscam modelar a maneira como as classes estão relacionadas entre si. Estes relacionamentos podem ocorrer

de diversas maneiras, como: associação (conectadas entre si), dependência (uma classe depende ou usa outra classe), especialização (uma classe é uma especialização de outra classe), ou em pacotes (classes agrupadas por características similares).

A associação é o relacionamento mais simples entre as classes. Ela permite que as classes compartilhem informações entre si e colaborem para a execução dos processos pelo sistema. Uma associação descreve um vínculo que ocorre normalmente entre os objetos de uma ou mais classes. Associações são representadas na UML por linhas contínuas ligando as classes envolvidas. Essas linhas podem possuir algumas informações importantes para auxiliar a compreensão do tipo de vínculo estabelecido. Uma informação simples e muito útil para facilitar o entendimento do tipo de associação é vincular um nome a associação.

Outra informação importante que é frequentemente vinculada a associação é a multiplicidade. A multiplicidade indica o número de objetos envolvidos em cada extremidade da associação. A multiplicidade é representada no diagrama de classes como dois números em cada extremidade da linha, que indicam o valor mínimo e máximo de objetos que estão envolvidos na associação.

Uma informação também importante que pode ser vinculada à associação é a navegabilidade. A navegabilidade é representada por uma seta em uma das extremidades da associação, identificando o sentido que as informações são transmitidas entre as classes envolvidas, ou seja, o sentido em que os métodos poderão ser disparados. Outra informação que pode ser útil na associação é a definição de papéis, que são informações extras na associação que podem ajudar explicar a função de um objeto dentro da associação.

A agregação é um tipo especial de associação que tenta demonstrar a relação todo/parte entre os objetos associados. Esta associação indica que a classe que representa o item maior (o "todo") é formada por itens menores (as "partes"). A agregação também é conhecida como um relacionamento do tipo "tem-um", o que significa que um objeto todo contém objetos parte. A agregação é representada no modelo de uma forma diferente, representada por um losango sem preenchimento.

A composição é uma variação da agregação, onde o vínculo entre as classes todo e parte é mais forte. Nesta associação, os objetos das classes parte só existem se o objeto da classe todo, ao qual estão relacionados, existir. Os objetos das classes parte também só podem ser destruídos pelo objeto da classe todo ao qual estão relacionados. A representação da composição diferencia-se graficamente do símbolo de agregação por utilizar um losango preenchido. Em ambos relacionamentos o losango fica representado na classe todo.

A generalização é um tipo de relacionamento especial com o objetivo de representar a ocorrência de herança entre as classes, identificando a classe-mãe (ou superclasse), chamadas também de classes gerais, e classes-filhas (ou subclasses), chamadas também de classes especializadas, demonstrando a hierarquia entre as classes e possivelmente métodos polimórficos nas classes especializadas. A generalização é representada na UML como uma linha com um triângulo apontado para a superclasse.

3.1.2 Diagrama de Atividades

O Diagrama de Atividades é um diagrama para modelagem dos aspectos dinâmicos do software com o objetivo de mostrar o fluxo de uma atividade para outra [OMG 2011b]. Atualmente, é baseado em Redes de Petri e apresenta muitas semelhanças com os antigos fluxogramas, utilizados para desenvolver lógica de programação e determinar o fluxo de controle de um algoritmo [Guedes 2009]. Na maior parte das vezes, o Diagrama de Atividades é utilizado para modelagem de etapas sequenciais (e possivelmente concorrente) em um processo computacional, mas também pode ser usado para modelagem do fluxo de um objeto, a medida que ele passa de um estado para outro em pontos diferentes do fluxo de controle [Booch et al. 2006]. A Figura 3.3 apresenta um diagrama de atividades modelado para a atividade de um pedido em um restaurante. Os principais elementos que compõem o diagrama de atividades são apresentados a seguir.

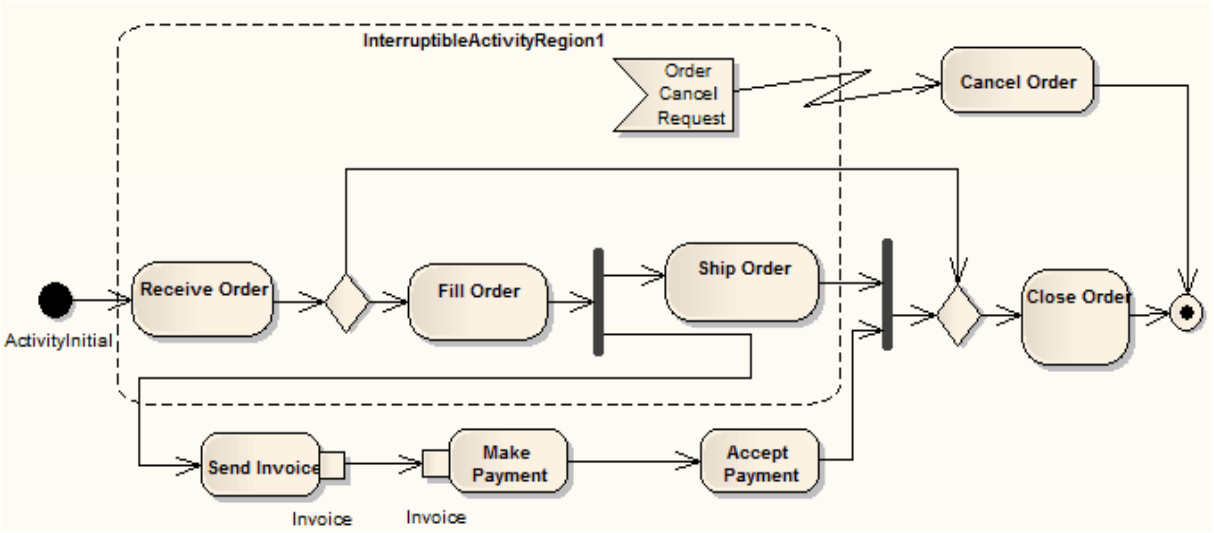

Figura 3.3: Exemplo de um Diagrama de Atividades UML. - Extraído de http://www.sparx-systems.com

O Diagrama de Atividades busca a modelagem de atividades do sistema, que podem ser um método, um algoritmo ou até mesmo um processo completo [Guedes 2009]. Uma atividade é uma sequência estruturada de um comportamento que pode ser composta por ações ou outras sub-atividades, controladas potencialmente por nós de decisão e sincronismo. Uma ação representa um elemento atômico básico de uma atividade que deve ser executado. Ações podem ser de vários tipos, como: cálculo de funções primitivas; invocação de comportamentos e outras atividades; ações de comunicação, como o envio de sinais; e manipulação de objetos, como leitura e gravação de atributos ou mesmo instanciação ou destruição de objetos [OMG 2011c]. As atividades são representadas na UML como retângulos com as bordas arredondadas.

Quando uma ação ou uma sub-atividade é executada completamente, o fluxo de controle passa imediatamente para a próxima ação ou sub-atividade. O fluxo de controle

é um conector que liga dois nós, enviando sinais de controle. O fluxo de controle pode possuir algumas informações adicionais como: uma descrição, uma condição de guarda ou uma restrição.

O Diagrama de Atividades possui alguns nós de controle que têm como objetivo controlar o fluxo de execução do diagrama. Primeiramente, tem-se o nó inicial e final, usados para representar o ponto de início e de fim de uma atividade, respectivamente. Cada atividade pode conter apenas um nó inicial e vários nós finais. Para indicar o final de um fluxo pode-se utilizar o nó final de fluxo, que indica o encerramento de um fluxo mas não de toda atividade. Este nó é representado por um círculo com um X em seu interior.

O nó decisão é um nó de controle utilizado para representar uma escolha entre dois ou mais fluxos possíveis, em que apenas um fluxo de execução será escolhido. Em geral, o nó decisão é acompanhado com uma condição booleana de guarda mutualmente exclusiva, representado por um texto entre colchetes. O nó de decisão também é utilizado para unir um fluxo de execução dividido por um nó de decisão anterior.

O nó de bifurcação/união é um nó de controle que pode tanto dividir um fluxo em dois ou mais fluxos concorrentes, chamado de nó bifurcação, como mesclar dois ou mais fluxos concorrentes em um único fluxo de controle, chamado de nó de união.

Uma nova notação, denominada pino (*Pin*), foi introduzida para nós objetos na UML 2.0 [OMG 2011c]. Os pinos representam portas onde os objetos entram e saem de uma ação, podendo ser comparados a parâmetros em um programa. Um pino de entrada (*InputPin*) significa que um objeto é entrada para uma ação. Já um pino de saída (*OutputPin*) representa que um objeto é saída de uma ação. Pinos são representados na UML como retângulos pequenos, anexados às ações.

Similar aos pinos, um nó de parâmetro de atividade (*Activity Parameter Node*) é um nó de objeto utilizado para representar a entrada (parâmetros) ou saída de um fluxo de objetos em uma atividade.

3.1.3 Mecanismos de extensão

O OMG define duas abordagens possíveis para a definição de linguagens específicas de domínio [Fuentes e Vallecillo 2004]. O primeiro baseia-se na definição de uma nova linguagem. Esta abordagem é uma alternativa à UML e utiliza os mecanismos previstos pelo OMG para a definição de linguagens visuais orientadas a objetos, mecanismos estes que têm sido utilizados para a definição da UML e seu metamodelo. Desta forma, a sintaxe e a semântica dos elementos da nova linguagem são definidos para atender as características específicas do domínio. A segunda alternativa é baseada na especialização da UML, em que alguns dos elementos da linguagem são especializados, adicionando-os novas restrições, sempre respeitando o metamodelo da UML e mantendo a semântica original dos seus elementos. Esta abordagem é chamada de perfil (*Profile*).

De acordo com o OMG [OMG 2011b], *Profile* é o mecanismo que permite que elementos de um metamodelo possam ser estendidos com o objetivo de adaptá-los para diferentes propósitos. Isso inclui a capacidade de adaptar o metamodelo da UML para diferentes plataformas (como J2EE ou. NET) ou domínios (como em tempo real, a modelagem de processos de negócios ou engenharia de sistemas). Um perfil deve fornecer mecanismos para especializar um metamodelo de referência de tal forma que a semântica especializada não contradiz a semântica do metamodelo "original". Os mecanismos de extensão que são fornecidos pela UML são: estereótipos, valores marcados (*tagged values*) e restrições. A menos que haja uma necessidade real de se desviar do metamodelo UML e criar uma nova linguagem, os benefícios do uso de perfis UML, sem dúvida, superam as suas limitações [Fuentes e Vallecillo 2004].

Perfis permitem a personalização de qualquer linguagem definida em conformidade com o metametamodelo MOF, e não apenas com o metamodelo UML. Neste trabalho foi utilizada a SysML, um perfil UML especializado para o domínio da Engenharia de sistemas.

3.2 SysML

A *System Modeling Language* (SysML) [OMG 2012] é uma linguagem de modelagem gráfica de propósito geral para especificar, analisar, projetar e verificar sistemas complexos que podem incluir *hardware*, software, informações, processos, pessoas e instalações. A SysML tem a intenção de unificar as diversas linguagens de modelagem utilizadas por engenheiros de sistemas, o que pode facilitar a comunicação entre times heterogêneos, como por exemplo times compostos de engenheiros mecânicos, eletricistas e analistas de sistemas [Soares e Vrancken 2007].

A SysML é baseada em um subconjunto mínimo da UML que satisfaz as necessidades dos engenheiros de sistemas, adaptando a UML somente quando é necessário [Vanderperren e Dehaene 2005]. Como subconjunto, diagramas UML considerados muito específicos para software (Diagrama de Objetos e Diagrama de Implantação) ou redundantes com outros diagramas (Diagrama de Comunicação e Diagrama de Tempo) não foram incluídos na SysML. Alguns diagramas são derivados de UML sem alterações significativas (Diagrama de Máquina de Estados, Diagrama de Casos de Uso, Diagrama de Sequência e o Diagramas de Pacotes) e outros diagramas são derivados com modificações (Diagrama de Classes, Diagrama de Atividades e o Diagrama Estrutura Composta). Dois novos diagramas são adicionados a SysML, o Diagrama de Requisitos e o Diagrama Paramétrico. Uma das principais melhorias da SysML em relação a UML é o suporte para representar os requisitos e relacioná-los com os modelos do sistema, o projeto real e os testes [Vanderperren e Dehaene 2005]. A Figura 3.4 apresenta a relação entre a SysML e a UML usando um diagrama de Venn.

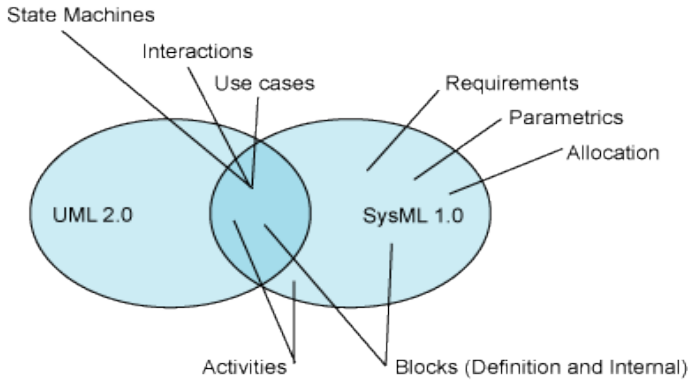

Figura 3.4: Relação entre UML e SysML. - [OMG 2012]

3.2.1 Diagramas da SysML

A SysML possui nove diagramas separados em três categorias: diagramas estruturais, diagramas comportamentais e o Diagrama de Requisitos. As construções estruturais, assim como os diagramas estruturais da UML, definem os elementos estáticos e de estrutura utilizados na SysML. Os diagramas que incluem as construções estruturais são: Diagrama de Pacotes, Diagrama de Definição de Blocos, Diagrama de Blocos Internos e Diagrama Paramétrico. Já as construções comportamentais especificam as partes dinâmicas utilizadas nos diagramas de comportamento da SysML. Os diagramas de comportamento são: Diagrama de Atividades, Diagrama de Sequência, Diagrama de Máquina de Estados e o Diagrama de Casos de Uso. O Diagrama de Requisitos é utilizado para representar hierarquias entre requisitos e/ou mostrar uma exigência individual e suas relações com outros elementos do modelo. Ao invés de representar os requisitos apenas como texto ou mesmo acompanhado por figuras, como é feito tradicionalmente, a SysML permite a representação de requisitos como elementos do modelo. Uma visão geral dos diagramas SysML é apresentada na Figura 3.5 e a seguir é apresentada uma breve introdução de cada diagrama utilizado nesta dissertação.

3.2.2 Diagrama de Blocos

O Diagrama de Definição de Bloco (BDD, *Block Definition Diagram*) [OMG 2012] é usado para definir blocos, suas características e relações com outros blocos estruturais. Os blocos são definidos de acordo com suas características estruturais e comportamentais, e os relacionamentos, usando associações, generalizações e dependências [OMG 2012]. O Diagrama de Blocos é a maneira mais simples de descrever a estrutura do sistema, sendo o diagrama mais comum da SysML [Delligatti 2013]. A Figura 3.6 apresenta um diagrama de blocos completo. Os principais elementos que compõem o diagrama de blocos serão apresentados a seguir.

Um bloco representa uma unidade modular que descreve a estrutura de um sistema,

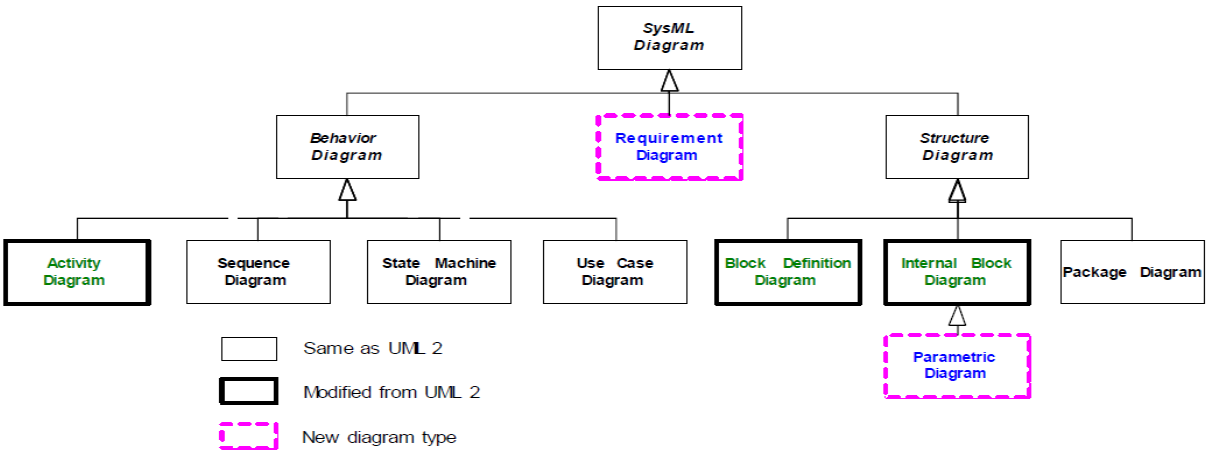

Figura 3.5: Diagramas da SysML - [OMG 2012]

ou um elemento que pode incluir características estruturais e comportamentais, tais como propriedades e operações, que representam o estado do sistema e o comportamento que o sistema pode apresentar [OMG 2012]. Como o Diagrama de Blocos estende o Diagrama de Classes da UML, um bloco da SysML estende a classe da UML com elementos e restrições adicionais. O bloco é representado graficamente por um retângulo dividido em uma série de compartimentos nomeados, onde cada compartimento pode representar propriedades, operações ou restrições.

O primeiro compartimento de qualquer bloco SysML é o compartimento nome. No compartimento nome o bloco é identificado com o nome do bloco precedido do esteriótipo «Block». Após o compartimento nome, uma série de outros compartimentos opcionais podem aparecer em um bloco. Estes compartimentos apresentam as características estruturais e comportamentais do bloco. Os compartimentos opcionais são: *Parts*, *References*, *Values*, *Constraints*, *Operations*, *Receptions*, *Full Ports*, *Proxy Ports*, *Flow Properties* e *Structure*.

As características estruturais, também chamadas de propriedades, são usadas para capturar as relações estruturais e os valores de um bloco. Uma propriedade possui um tipo que complementa a sua definição. O tipo pode ser um bloco ou algum conceito mais básico, como um inteiro. Existem sete tipos de propriedades: Partes (*Parts*), Referências (*References*), Valores (*Values*), Portas (*Ports*), Propriedades de Fluxo (*Flow Properties*), Restrições (*Constraints*) e Estrutura (*Structure*).

Partes Uma propriedade parte (*part*) identifica o uso de uma ou várias instâncias de um bloco no contexto de um outro bloco [Friedenthal et al. 2008]. Esta propriedade representa a estrutura interna de um bloco, ou seja, um bloco é composto por suas partes [Delligatti 2013]. Uma parte pode pertencer a apenas um bloco de cada vez. Uma propriedade parte é uma propriedade de um bloco, e, como tal, pode ser listada em um compartimento do bloco identificado pela palavra-chave *Parts*.

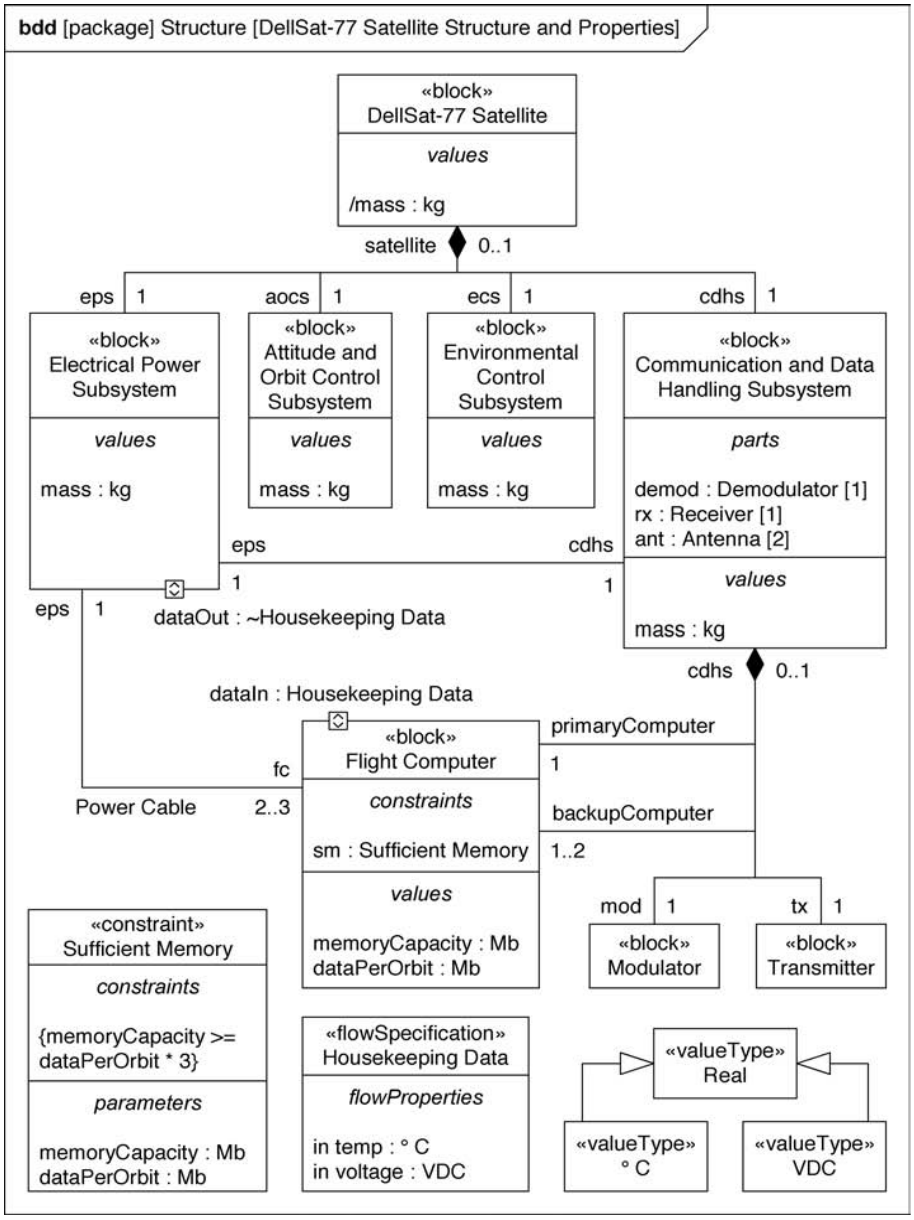

Figura 3.6: Exemplo de um Diagrama de Blocos. - [Delligatti 2013]

Uma parte é identificada em seu compartimento pelo seu nome, definido pelo usuário; seu tipo, que é sempre um bloco; e sua multiplicidade, que é uma informação opcional que representa uma restrição quanto ao número de instâncias que uma parte pode representar dentro da composição, podendo ser representada por um número inteiro ou um par de números inteiros entre colchetes, que indica o valor mínimo e máximo de instâncias.

As partes também podem ser representadas como um relacionamento de composição entre os blocos. A relação de composição, também chamada de relacionamento todo-parte, é um relacionamento onde um bloco, que representa o todo, é associado a um bloco, que representa a parte. O nome da propriedade parte é o nome adicionado como legenda da associação na extremidade do bloco parte. O tipo da parte é definido pelo bloco parte a qual está associado o bloco todo. A multiplicidade é definida em cada extremidade da

associação.

Referências Na propriedade parte, como na composição, tem-se a ideia de destruição, onde ao destruir uma instância do bloco todo também destroem-se as instâncias dos blocos parte. Na propriedade referência (*reference*) essa ideia de destruição, associada a composição, não se aplica [Friedenthal et al. 2008]. A propriedade referência visa atender a necessidade de serviços ou troca de informações de um bloco sem ser uma parte que compõem o bloco.

Como toda propriedade, as referências podem ser listadas em um compartimento separado. O compartimento para armazenamento de referências é identificado com a palavra-chave *References*. Uma referência possui um nome, um tipo, e uma multiplicidade. As entradas do compartimento de referências seguem o mesmo formato das entradas do compartimento *Parts*. Nas referências não existe uma restrição sobre o número de blocos todo que referenciam a mesma instância de um bloco referência.

Como acontece nas partes, as referências também podem ser expressas usando um relacionamento entre blocos. Neste caso, a propriedade referência é representada como uma relação de agregação entre o bloco todo e o bloco referência. O nome da propriedade referência, o tipo e multiplicidade são dados da mesma forma que no relacionamento de composição da propriedade parte.

Propriedade Valor A Propriedade valor (*Value Property*) é usada para modelar as características quantitativas associadas a um bloco [Friedenthal et al. 2008]. Na maioria das vezes, uma propriedade valor é algo que pode-se atribuir um número [Delligatti 2013]. As propriedades Valor são particularmente úteis em conjunto com propriedades de restrição para a construção de um modelo matemático para um sistema. As propriedades Valor são adicionadas a um compartimento identificado com a palavra-chave *values*.

Uma propriedade Valor possui as mesmas características que as outras propriedades, como um nome, um tipo e uma multiplicidade. As principais diferenças das propriedades valor estão quanto ao seu tipo, a possibilidade de associar um valor inicial e a possibilidade de definir uma distribuição probabilística para seus valores. Uma propriedade Valor definida como uma distribuição probabilística possui um nome para a distribuição, um conjunto de valores que caracterizam a distribuição probabilística, o nome e o tipo da propriedade desta propriedade valor.

Diferente das partes e referências, que possuem como tipo um bloco, a definição de uma propriedade Valor possui como tipo um *Value Type* (Tipo de valor). O *Value Type* é um elemento de definição, como os blocos, que na maioria das vezes fornece uma definição uniforme de uma quantidade [Delligatti 2013]. Os *valuesType* normalmente são utilizados como tipo de propriedades Valor, mas podem ser usados como tipo de outros elementos, como: Tipo de retorno de operações, parâmetros de operações e recepções e itens de

fluxo nos conectores. Um *valueType* pode ser de três tipos: Primitivo, Estruturado e Enumeração.

A SysML define também os conceitos de unidade e dimensão que podem ser usados para caracterizar um *Value Type*. O uso destes elementos é opcional na definição de um *Value Type*. A dimensão identifica uma quantidade física, como comprimento, cujo valor pode ser expresso em termos de unidades definidas (por exemplo, como metros ou pés). A unidade tem de estar sempre relacionada com uma dimensão, mas uma dimensão não precisa ter qualquer unidade associada, e muitas vezes as equações podem ser expressas em termos de quantidades que incluem as dimensões sem especificar unidades [Friedenthal et al. 2008].

Portas Portas (*Ports*) são pontos de interação onde entidades externas podem se conectar e interagir com um bloco de uma maneira diferente e/ou limitada se comparado a conexão direta com o próprio bloco [OMG 2012]. A porta permite que comportamentos de um bloco possam ser acessados por entidades externas por meio de conectores. Um bloco pode ter várias portas que especificam diferentes pontos de interação [Friedenthal et al. 2008]. Apesar das portas serem definidas no Diagrama de Blocos, sua principal aplicação é apresentar a interação entre as partes que compõem um bloco, informação esta que é apresentada em um Diagrama de Blocos Internos [Friedenthal et al. 2008].

A SysML identifica dois tipos de portas: Portas *Proxy* e Portas *Full*. As Portas *Proxy* definem o limite de um bloco, especificando quais características do bloco são visíveis usando conectores externos, enquanto as portas *Full* definem a fronteira com seus próprios recursos. As portas que não são especificadas como *Proxy* ou *Full* são simplesmente chamados de Portas (*Ports*), sendo adicionadas no compartimento identificado como *Ports*.

A Interface Necessária (*Required Interface*) associada a uma porta especifica uma ou mais operações necessárias para o bloco (ou suas partes) realizar o seu comportamento. Uma Interface Fornecida (*Provided Interface*) associada à uma porta especifica uma ou mais operações que um bloco (ou uma ou mais partes) fornecem a blocos externos.

Propriedade de Fluxo Propriedades de Fluxo (*Flow Properties*) [OMG 2012] especificam os tipos de itens que podem fluir entre um bloco e o meio externo. Uma propriedade fluxo representa o que pode fluir para dentro e/ou para fora de um bloco por meio de uma porta. As propriedades de fluxo são inseridas no compartimento identificado com a palavra-chave «flowProperties» e possuem uma direção, que pode ser *in*, *out* ou *inout*, um nome e um tipo, que deve ser um *Value Type*, um Bloco ou um sinal. O tipo representa o que irá fluir pela porta.

Restrições Uma propriedade de restrição (*Constraint Property*) geralmente representa uma relação matemática (uma equação ou desigualdade) que é imposta a um conjunto de propriedades valor [Delligatti 2013]. A propriedade de restrição é uma parte essencial da construção de modelos matemáticos do sistema. Este modelo matemático pode ser modelado usando Diagramas Paramétricos. As propriedades de restrição são adicionadas a um compartimento identificado com a palavra-chave *constraints* e possuem um nome e um tipo, que deve ser o nome de um bloco de restrição.

Um bloco de restrição (*Constraint Block*) é um tipo especial de bloco que é criado para encapsular uma expressão de restrição reutilizável [OMG 2012]. Na maioria das vezes, uma expressão de restrição é uma equação ou uma desigualdade. Os blocos de restrição são representados graficamente como um retângulo de bordas sólidas identificado com o esteriótipo «constraint» precedendo o nome do bloco. O bloco de restrição possui dois compartimentos nomeados. O primeiro, chamado *constraint*, inclui a expressão de restrição, enquanto o segundo, chamado de *parameters*, apresenta os parâmetros necessários para a restrição.

Estrutura O compartimento Estrutura (*structure*) é o único compartimento que não lista recursos. Pelo contrário, é um compartimento gráfico que mostra a estrutura interna de um bloco. Este compartimento pode utilizar qualquer anotação e elementos de um Diagrama de Blocos Internos (IBD). Este compartimento raramente é utilizado pelos usuários da SysML [Delligatti 2013]. Este compartimento é identificado pela palavra-chave *structure*.

Junto com as características estruturais, os blocos também podem possuir características comportamentais. Características comportamentais descrevem como os blocos respondem a determinadas solicitações. Existem dois tipos de características comportamentais: operações e recepções.

Operação Uma operação é uma característica comportamental que geralmente representa uma solicitação síncrona, isto é, o solicitador espera por uma resposta da operação antes de continuar sua execução. No entanto, a SysML permite que o modelador represente também comportamentos assíncronos como uma operação [Delligatti 2013]. Operações definem um conjunto de parâmetros que descrevem os argumentos passados com a chamada da operação e/ou valores que são retornados da operação, dependendo da direção indicada no parâmetro. Essa direção pode ser: *in*, *out* ou *inout*. Operações são definidas em um compartimento identificado com a palavra-chave *Operations* e são descritas pelo seu nome, uma lista de parâmetros, um tipo opcional de retorno, que pode ser um *valueType* ou um bloco, e a multiplicidade, que é uma restrição no número de instâncias do tipo de retorno que a operação irá retornar quando completa.

Recepção A recepção (*Receptions*) é um comportamento assíncrono que o bloco realiza quando é acionado por um sinal enviado por um usuário. Neste caso, o usuário que acionou a recepção não aguarda a resposta e continua sua execução. Diferente das operações, as recepções não possuem tipo de retorno.

Como o sinal é um elemento do modelo, é possível utilizar um sinal para representar qualquer tipo de matéria, energia ou dados que uma parte-cliente envia para outra parte-alvo com a finalidade de provocar um comportamento de recepção. Como um bloco, um sinal pode possuir propriedades. Na maioria das vezes, essas propriedades representam os dados que o sinal carrega do cliente para um alvo. Quando o sinal chega ao alvo, a recepção é disparada utilizando as propriedades do sinal como parâmetros [Delligatti 2013]. Um sinal é representado graficamente semelhante a um bloco, sendo identificado com o esteriótipo «signal» precedendo o nome do sinal.

Recepções são definidas em um compartimento nomeado com a palavra-chave *Receptions* e são descritos pelo seu nome, precedido da palavra-chave «signal», e uma lista de atributos. Como uma recepção é associada a um sinal, é definido que o nome de recepção deve corresponder ao nome do sinal modelado que irá acioná-la.

3.2.3 Diagrama de Blocos Internos

O Diagrama de Blocos Internos (IBD, *Internal Block Diagram*) [OMG 2012] tem como objetivo apresentar a estrutura interna de um bloco específico. O IBD normalmente apresenta como as partes e referências de um bloco estão relacionadas no contexto deste bloco. O Diagrama de Blocos Internos é baseado no Diagrama de Estrutura Composta da UML, com restrições e extensões definidas pela SysML. A Figura 3.7 apresenta um Diagrama de Blocos Internos extraído da documentação da SysML. Os principais elementos que compõem o Diagrama de Blocos Internos são apresentados a seguir.

O Diagrama de Blocos Internos busca complementar o Diagrama de Blocos, permitindo modelar informações que não podem ser representadas em um BDD. Algumas dessas informações são: as conexões entre partes e referências; os tipos de itens que fluem por meio destas conexões (matéria, energia ou dados); e os serviços que são prestados e/ou necessários de cada parte ou referência.

A estrutura interna de um bloco específico é normalmente apresentada usando a relação entre as partes e referências que o compõem. Cada parte associada ao bloco todo é representada no Diagrama de Blocos Internos como um retângulo, semelhante um bloco, com as bordas sólidas. A parte é identificada pelo seu nome, dois pontos (:) e o tipo da parte. A multiplicidade da parte pode ser exibida no canto superior direito da parte ou entre colchetes após o tipo da parte. As referências são representadas de forma semelhante, se diferenciando apenas pelo seu formato gráfico, sendo representadas por um retângulo de borda tracejada.

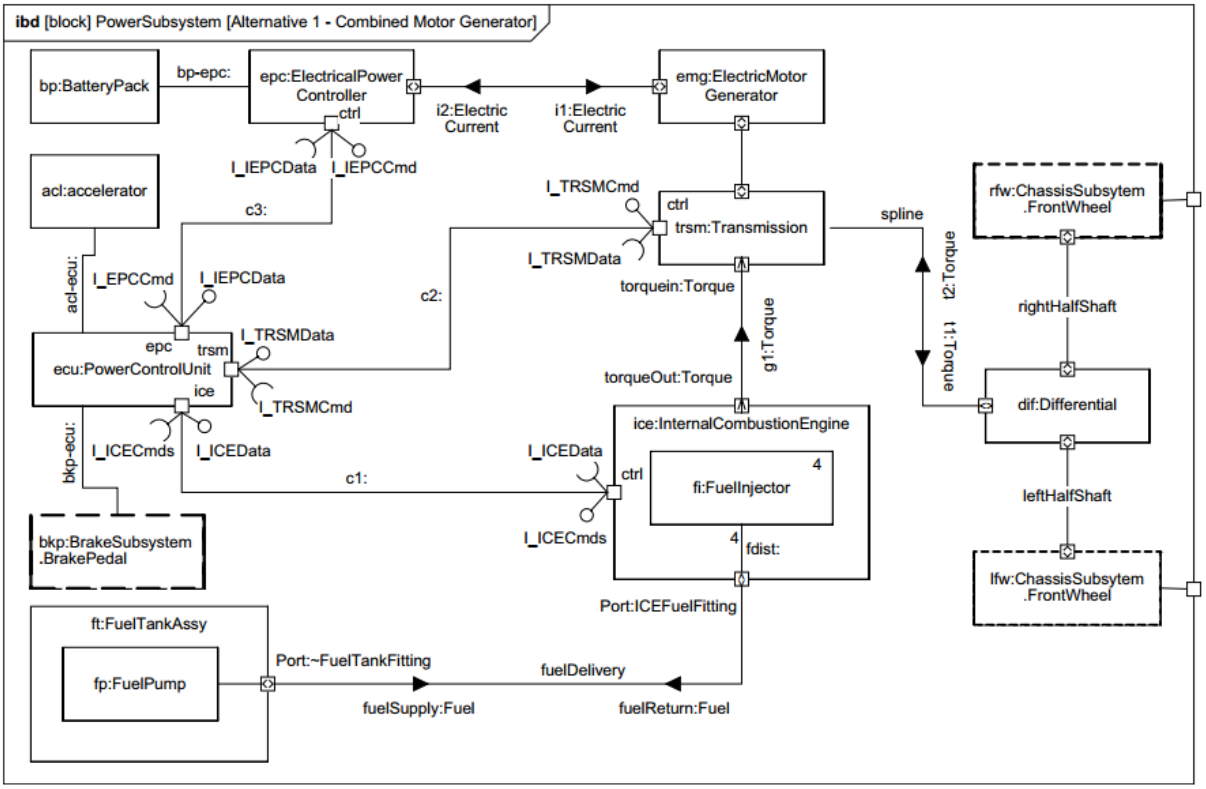

Figura 3.7: Exemplo de um Diagrama de Blocos Internos. - [OMG 2012]

Um conector (*Connector*) é usado para ligar duas propriedades, normalmente partes e/ou referências, possibilitando que estas propriedades interajam entre si, embora o conector não diz nada sobre a natureza da interação [Friedenthal et al. 2008]. Conectores normalmente são conectados às portas que definem uma interação mais detalhada entre as partes e/ou referências.

A interação entre duas propriedades pode se caracterizar pelo fluxo de entradas e saídas entre as partes, a invocação de serviços das partes, o envio e recebimento de mensagens entre as partes ou restrições entre as partes que compõem a relação [Friedenthal et al. 2008]. Sempre que necessário, a natureza e a direção de itens que fluem em um conector pode ser representada utilizando itens de fluxos (*Item Flows*).

Um item de fluxo (*Item Flow*) especifica o que flui entre blocos e/ou partes através de associações e conectores [OMG 2012]. Enquanto propriedades de fluxo especificam o que pode fluir (entrar ou sair do bloco), itens de fluxo especificam o que de fato flui entre os blocos e as partes.

Capítulo 4

Transformação Estrutural usando ATL (Block2Class)

Software-Intensive Systems [Tiako 2009] [Hinchey et al. 2008] são sistemas grandes e complexos em que o software é um componente essencial, interagindo com outros elementos, tais como outros *softwares*, sistemas, dispositivos, atuadores, sensores e com pessoas. Como o software é uma parte essencial destes sistemas, ele influencia a concepção, implementação e evolução do sistema como um todo. Exemplos de *Software-Intensive Systems* podem ser encontrados em diversos setores, tais como fábricas, transportes, telecomunicações e saúde.

Projetar *Software-Intensive Systems* é uma atividade desafiadora por muitas razões. O próprio ambiente em que os *Software-Intensive Systems* agem apresenta grandes desafios. *Software-Intensive Systems* são frequentemente usados para controlar infra-estruturas críticas em que qualquer erro, não-conformidade ou mesmo atrasos na resposta podem causar enormes prejuízos financeiros ou até mesmo colocar em risco a vida humana. Projetar e criar modelos são atividades importantes para melhorar a comunicação entre as equipes e diminuir significativamente ambiguidades da linguagem natural [Ludewig 2003]. Normalmente, em Engenharias de Sistemas e de Software, um artefato é considerado um modelo se tem uma representação gráfica, formal ou matemática [Bézivin 2006].

Atualmente, há uma variedade de linguagens de modelagem, métodos e técnicas aplicadas a todas as fases do desenvolvimento de sistemas de software. Uma extensa lista de técnicas para as atividades de projeto de software é apresentada em [Jiang et al. 2008]. Por exemplo, os elementos estruturais de software têm sido modelados utilizando o modelo Entidade-Relacionamento ou diagramas de blocos simples com semântica pouco clara [Edwards e Lee 2003]. Atualmente, não há dúvidas que a UML tem sido amplamente aplicada para o desenvolvimento de software na indústria. Apesar de seu relativo sucesso, a linguagem tem sido fortemente criticada [Bell 2004] [France et al. 2006] [André et al. 2007] [Soares e Vrancken 2009]. A crítica mais relevante sobre a UML em relação a este trabalho é que a UML apresenta deficiências para modelagem de elementos de um

Software-Intensive System que não são software. Esta é a principal razão pela qual a SysML foi proposta. A SysML é uma linguagem de modelagem de sistemas que suporta a especificação, análise, projeto, verificação e validação de uma ampla gama de sistemas complexos. A linguagem é derivada da UML, tendo em conta os aspectos de sistemas, tais como *hardware*, informação, processos e pessoas.

O foco deste trabalho está na fase de projeto de *Software-Intensive Systems*, mais especificamente no projeto da visão estrutural de uma arquitetura de *Software-Intensive Systems*. Portanto, é de extrema importância que não só o software desses sistemas seja modelado, mas também todos os elementos estruturais que compõem tal sistema.

Este capítulo tem dois objetivos principais. O primeiro deles é descrever uma introdução da SysML como uma linguagem de modelagem do processo de desenvolvimento de *Software-Intensive Systems*. Mais especificamente, o Diagrama de Blocos da SysML é aplicado na prática para descrever a arquitetura estrutural no domínio da gestão do tráfego rodoviário. A SysML foi aplicada a uma série de projetos [Viehl et al. 2006] [Laleau et al. 2010] em vários campos de atuação, tais como grandes telescópios [Karban et al. 2008], fabricação de automóveis [Balmelli et al. 2006], aplicações de controle de processos industriais [Hästbacka et al. 2011], e sistemas de gestão do tráfego rodoviário [Soares et al. 2011].

O segundo objetivo é propor um mapeamento entre metamodelos que busca descrever a relação entre o Diagrama de Blocos da SysML e o Diagramas de Classes da UML, que não foi bem descrita na especificação da SysML [OMG 2012]. Esta relação foi, então, implementada usando uma abordagem orientada a modelos. Uma transformação automática usando a linguagem ATL foi realizada com base no mapeamento descrito.

Outras abordagens baseadas em modelos que combinam SysML e ATL não são frequentes na literatura devido à novidade destas linguagens. Um exemplo pode ser encontrado em [Colombo et al. 2012], em que a transformação implementada em ATL é realizada considerando apenas o metamodelo SysML, ou seja, os autores propuseram uma transformação a partir de modelos de análise para modelos de projetos, apenas refinando os diagramas SysML. A ATL foi escolhida nesta pesquisa uma vez que tem sido aplicada com sucesso para as transformações em aplicações reais, como descrito na literatura [Jouault et al. 2008] [Kim et al. 2012] [Goknil et al. 2014]. Além disso, a ATL fornece uma ferramenta de apoio adequada, visto que a linguagem é parte do projeto Eclipse. A abordagem foi aplicada a uma aplicação prática no domínio da gestão do tráfego rodoviário, em que os *Software-Intensive Systems* importantes são implementados com o objetivo de auxiliar a vida moderna.

4.1 Relacionando Diagrama de Blocos SysML com Diagrama de Classes UML

Uma vez que o ponto de vista estrutural do sistema foi definido a partir do ponto de vista da engenharia de sistemas, engenheiros de software têm que mapear os elementos do sistema modelados como Blocos da SysML para elementos de software como classes e objetos da UML. A escolha foi usar o Diagrama de Classes da UML para representar o modelo de classes de software. Esta escolha é natural, dado que a SysML e a UML são linguagens de modelagem com raízes no mesmo metamodelo. Portanto, os elementos físicos de um sistema, modelados como Blocos SysML, são posteriormente implementados em um sistema de software como objetos correspondentes, uma vez que os elementos físicos estão incluídos na arquitetura de software.

Blocos SysML são transformados em uma ou mais Classes da UML durante as fases projeto de software e implementação. No entanto, esta transformação não é automática. Esta é uma atividade de modelagem que, por sua própria natureza, não tem regras rígidas. Algumas orientações úteis e um mapeamento entre metamodelos, apresentado na Figura 4.1, são propostos nesta seção.

As propriedades, operações e restrições são compartimentos dos blocos propostos na especificação SysML. As propriedades valor e restrições são mapeadas como atributos de uma classe, pois são características estruturais usadas para capturar as relações estruturais e os estados de um bloco. Operações e recepções são mapeados como métodos, pois eles estão relacionados ao comportamento do bloco a um determinado evento. Todas as propriedades da SysML têm visibilidade pública. Durante o mapeamento, boas práticas de projeto do paradigma orientado a objetos são aplicadas ao Diagrama de Classes que está sendo gerado, como por exemplo a ocultação de informações (*information hiding*). Assim, as propriedades são mapeadas como atributos privados ou protegidos de uma classe.

As operações são mapeadas em pelo menos um método de software, normalmente como um elemento da interface pública da classe. A razão é que uma operação do sistema pode de fato ser implementada utilizando um conjunto de métodos de software, em vez de apenas um. O fato é que engenheiros de sistemas e de software devem trabalhar juntos a fim de decidir a melhor solução para esse mapeamento.

Blocos SysML se relacionam uns com os outros por meio de associações. Estas podem ser associações simples, indicando que existe uma relação entre os blocos associados, ou associações especiais como a composição ou agregação. A escolha semântica entre os dois tipos de associações especiais depende do tempo de vida e da força da relação que a propriedade parte, ou referência, exerce no bloco todo.

Partes e referências são representadas no Diagrama de Blocos da SysML como associações de composição e agregação entre blocos, respectivamente. Partes representam que o

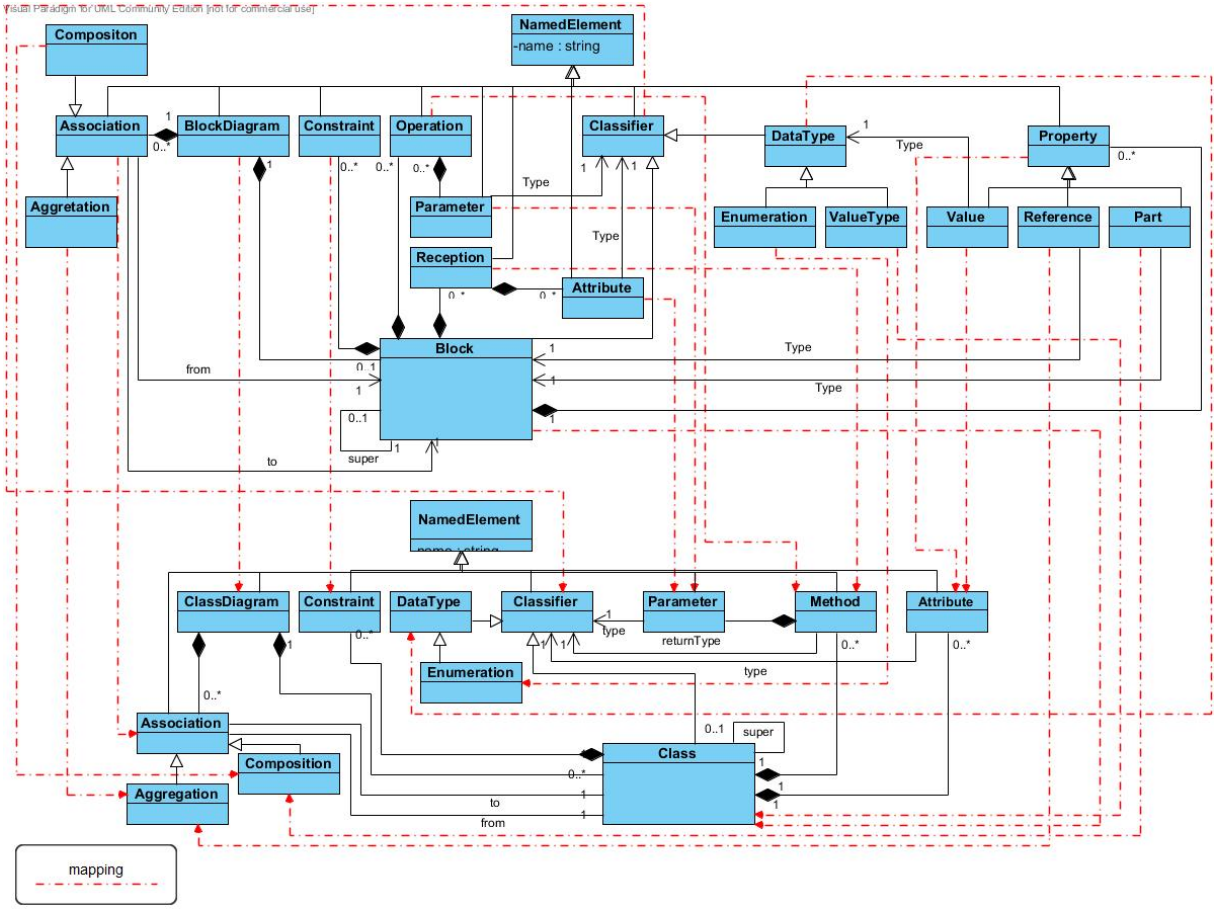

Figura 4.1: Mapeamento da transformação de Diagrama de Blocos da SysML para Diagrama de Classes da UML.

bloco a ser implementado é composto por outros blocos. Esta característica é mapeada no Diagrama de Classes da UML usando a associação de composição. As referências indicam que o bloco a ser mapeado utiliza outros blocos, mas não é composto de outros blocos. Esta característica é mapeada no Diagrama de Classes da UML usando a associação de agregação. Para o restante do mapeamento, as associações e as suas cardinalidades são mantidas inalteradas.

O *Value Type* é um elemento de definição, como os blocos, que na maioria das vezes fornece uma definição uniforme de uma quantidade. O bloco *Value Type* do Diagrama de Blocos é mapeado para o Diagrama de classes como um *DataType*. O *Enumeration* é um tipo de *Value Type* que define um conjunto de valores literais que podem ser associados aos elementos que possuem este tipo. O *Enumeration* do Diagrama de Blocos da SysML é mapeado com um *Enumeration* do Diagrama de Classes da UML.

O *Constraint Block* é um tipo especial de bloco criado para encapsular uma expressão de restrição. Como os blocos, o *Constraint Block* é mapeado como uma classe do Diagrama de Classes da UML.

Para implementação da relação, apresentada na Figura 4.1 e discutida nesta seção, é

utilizada a linguagem de transformação ATL, uma linguagem de transformação *Model-to-Model* híbrida. A implementação da relação usando ATL é apresentada na seção seguinte.

4.2 Regras especificadas em ATL

A ATL foi escolhida como a linguagem para implementar a transformação de Diagramas de Blocos SysML para Diagrama de Classes UML. Para a implementação e execução de uma transformação usando a ATL, é necessário definir os metamodelos dos modelos que irão participar da transformação. Para a transformação do Diagrama de Blocos da SysML para o Diagrama de Classes da UML foram utilizados dois metamodelos: o metamodelo do Diagrama de Blocos da SysML e o metamodelo do Diagrama de Classes da UML. Estes metamodelos são exatamente os mesmos propostos nas especificações da OMG. Usando estes metamodelos, foram definidas as regras ATL com o propósito de transformar blocos da SysML em classes da UML. No total onze regras ATL foram implementadas para criar a transformação automática. Todas as regras são apresentadas, de forma resumida, na Tabela 4.1.

A primeira regra implementada na transformação foi a Model2Model. Esta regra tem o objetivo de transformar o elemento modelo do Diagrama de Blocos da SysML em um elemento modelo do Diagrama de Classes da UML. O elemento modelo apresenta o nome do diagrama e é o elemento responsável por organizar todos outros elementos do diagrama. A regra Model2Model é apresentada na Figura 4.2.

```
rule Model2Model {
    from
        s : SysML!Model (s.oclIsTypeOf(SysML!Model))
    to
        t : UML2!Model (
            name <- s.name,
            packagedElement <- s.packagedElement
        )
}
```

Figura 4.2: Regra ATL Model2Model.

A regra Block2Class, apresentada na Figura 4.3, é a principal regra implementada para esta transformação. Ela tem como objetivo transformar cada Bloco do Diagrama de Blocos da SysML em sua respectiva Classe no Diagrama de Classes UML gerado. Para cada Bloco encontrado, é criada uma Classe correspondente com o mesmo nome do bloco. Essa regra também é responsável por transformar as propriedades dos blocos em atributos das classes, usando de forma indireta a regra Property2Attribute. A regra Property2Attribute, apresentada na Figura 4.4, é responsável por transformar, principalmente, as propriedades dos blocos da SysML em atributos das Classes da UML.

As *Lazy rules* Operation2Method e Reception2Method, apresentadas na Figura 4.5,

Tabela 4.1: Regras implementadas usando a ATL.

Regras	Descrição
Model2Model	Transforma o modelo de Diagrama de Blocos SysML para um modelo de Diagrama de Classes UML
Block2Class	Transforma cada Bloco do modelo SysML para uma Classe do modelo UML
Property2Attribute	Transforma as propriedades do Bloco SysML para atributos da Classe UML
Operation2Method	Transforma as operações dos Blocos SysML em métodos das Classes UML
Reception2Method	Transforma as recepções dos Blocos SysML em métodos das Classes UML
ParameterAttribute2Parameter	Transforma parâmetros das operações e atributos das recepções dos blocos SysML em parâmetros dos métodos das Classes UML
AssociationPartReference2Association	Transforma partes, referências (representado como associações) e associações dos blocos SysML em associações entre Classes UML
DataType2DataType	Transforma um *DataType* de um diagrama de Blocos SysML em um *DataType* de um Diagrama de Classes UML
Enumeration2Enumeration	Transforma um *Enumeration* de um Diagrama de Blocos SysML em um *Enumeration* de um Diagrama de Classes UML
EnumerationLiteral2EnumerationLiteral	Transforma um *EnumerationLiteral* de um Diagrama de Blocos SysML em um *EnumerationLiteral* de um Diagrama de Classes UML
Generalization2Generalization	Transforma a relação de generalização entre Blocos SysML para a relação de generalização entre Classes UML

```
rule Block2Class {
    from
        s : SysML!Class
    to
        t : UML2!Class(
                name <- s.name,
                generalization <- s.generalization->collect(e | thisModule.Generalization2Generalization(e)),
                ownedAttribute <- s.ownedAttribute,
                ownedOperation <- s.ownedOperation->collect(e | thisModule.Operation2Method(e)),
                ownedOperation <- s.ownedReception->collect(e | thisModule.Reception2Method(e))
            )
}
```

Figura 4.3: Regra ATL Block2Class.

são chamadas na regra Block2Class e tem como objetivo transformar cada operação ou recepção, respectivamente, dos Blocos da SysML em métodos das Classes da UML. Ope-

```
rule Property2Attribute {
    from
        s : SysML!Property
    to
        t : UML2!Property(
                name <- s.name,
                association <- s.association,
                lower <- s.lower,
                upper <- s.upper,
                visibility <- 'private',
                isOrdered <- s.isOrdered,
                type <- s.type,
                isUnique <- s.isUnique,
                aggregation <- s.aggregation,
                clientDependency <- s.clientDependency
        )
}
```

Figura 4.4: Regras ATL Property2Attribute.

rações e recepções são elementos simples de um bloco SysML, contendo em suas especificações apenas um nome, visibilidade e parâmetros. Assim, as transformações foram bem simples, sendo necessária apenas uma regra de transformação adicional denominada ParameterAttribute2Parameter, a fim de transformar os parâmetros das operações e atributos das recepções em parâmetros dos métodos gerados. A regra ParameterAttribute2Parameter é apresentada na Figura 4.6.

```
lazy rule Operation2Method{
        from
            s : SysML!Operation
        to
            t : UML2!Operation(
                name <- s.name,
                visibility <- 'public',
                ownedParameter <- s.ownedParameter
            )

}

lazy rule Reception2Method{
        from
            s : SysML!Reception
        to
            t : UML2!Operation(
                name <- s.name,
                visibility <- 'public',
                ownedParameter <- s.ownedParameter
            )
}
```

Figura 4.5: Regras ATL Operation2Method e Reception2Method.

A *Lazy Rule* Generalization2Generalization, chamada pela regra Block2Class, é uma regra simples e tem como objetivo a transformação do relacionamento de generalização

```
rule ParameterAttribute2Parameter{
    from
        s : SysML!Parameter

    to
        t : UML2!Parameter(
            name <- s.name,
            type <- s.type
        )
}
```

Figura 4.6: Regras ATL ParameterAttribute2Parameter.

entre blocos da SysML no relacionamento de generalização entre Classes da UML. A generalização é um relacionamento especial do Diagrama de Blocos da SysML e do Diagrama de Classes da UML, e a única informação relevante deste elemento é a referência que o bloco que está em processo de transformação faz ao seu superbloco. A regra Generalization2Generalization é apresentada na Figura 4.7.

```
lazy rule Generalization2Generalization{
    from
        s : SysML!Generalization
    to
        t : UML2!Generalization(
                general <- s.general
        )

}
```

Figura 4.7: Regra ATL Generalization2Generalization.

Partes e referências de um bloco da SysML são representadas no modelo como associações de composição e agregação, respectivamente. Assim, a regra implementada para transformação de associações entre os blocos em associações entre as classes também irá realizar a transformação de partes e referências em associações de composição e agregação entre as classes. A regra criada para esta transformação, apresentada na Figura 4.8, é uma regra simples e direta que mantem todas as cardinalidades e legendas vinculadas à associação entre os blocos.

```
rule Association2Association{
    from
        s : SysML!Association
    to
        t : UML2!Association(
                name <- s.name,
                memberEnd <- s.memberEnd,
                ownedEnd <- s.ownedEnd
        )
}
```

Figura 4.8: Regra ATL Association2Association.

4.3 Estudo de caso realizado

O Diagrama de Blocos SysML é aplicado neste trabalho com o objetivo de representar a visão estrutural da arquitetura de sistemas. O estudo de caso apresentado nesta seção é uma arquitetura para um sistema de gestão do tráfego rodoviário (RTMS, *Road Traffic Management System*) [Almejalli et al. 2008] [Almejalli et al. 2009], que são *Software-Intensive Systems* utilizados em atividades como controle, previsão, visualização e monitoramento do tráfego rodoviário. A visão estrutural da arquitetura descreve, em alto nível, quais elementos cooperam uns com os outros, sem preocupações sobre como essa interação é feita. Estes elementos são apresentados na Tabela 4.2 e descritos a seguir.

- Gerenciadores de Origem-Destino (ODMGR, *Origin-Destination Managers*) - representam a relação entre uma origem e um destino e compreendem uma ou mais rotas;
- Gerenciadores de Rota (*Route Managers*) - controlam o conjunto de rotas de uma origem a um destino;
- Ligação (*Link*) - podem ser de dois tipos principais, *Links* principais e *Links* acessadores. O *Link* principal é a ligação do ponto de fusão para o ponto de escolha e o *Link* acessador é a ligação a partir do ponto de escolha para o ponto de fusão;
- Junções (*Junctions*) - compreendem o *Link* principal de saída e os *Links* Acessadores de entrada de um cruzamento ou entroncamento da auto-estrada. A junção é um

Tabela 4.2: Elementos da arquitetura de Software.

Camada	Elemento Geográfico	Monitoração	Controle
Network	*Network Route*	*OD-matrix Route travel time*	ODMGR. *Route Mgr.*
Link	*Link merge/choice point*	*Capacity Avg. speed, turn fractions*	*Link Mgr. Junction Mgr.*
Point	*Sensor/actuator position*	*Speed, flow, occupancy*	*Actuator Mgr.*

local onde o tráfego pode mudar suas rotas, direções e às vezes até mesmo o modo de viajar;

- Esquemas de Controle (*Control Schemes*) - é um conjunto coerente de medidas disparadas por padrões recorrentes no estado de tráfego, tais como os horários de pico da manhã ou o êxodo de final de semana.

A representação da visão estrutural da arquitetura é apresentada na Figura 4.9 usando um Diagrama de Blocos da SysML. Os componentes distribuídos têm que se comunicar uns com os outros, pois eles trabalham em cooperação. Estes componentes continuamente medem o estado do tráfego e comunicam sobre este estado uns com os outros em tempo real. Por exemplo, os *links* têm que comunicar com outros *links*, com o objetivo de medir com sucesso o estado do tráfego. Rotas participam de pelo menos um *link*, mas podem participar de mais.

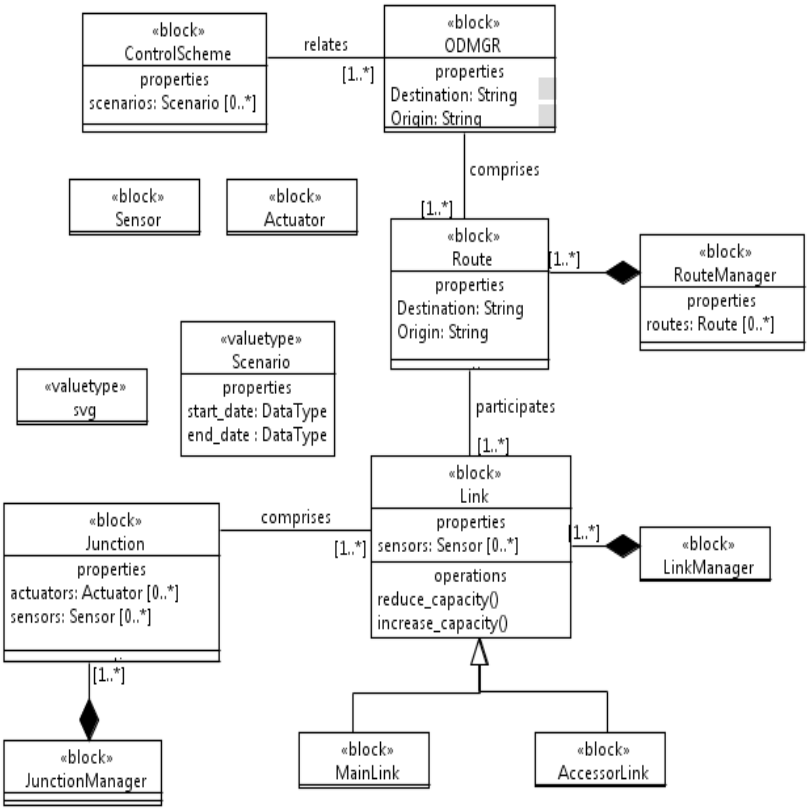

Figura 4.9: Visão estrutural da arquitetura do RTMS usando diagrama de blocos SysML.

Após todas regras ATL implementadas foi realizada a modelagem do Diagrama de Blocos de entrada da transformação. O Diagrama de Blocos de entrada foi criado usando a ferramenta Papyrus, que está integrada à ferramenta TopCased. O Papyrus oferece todo o suporte para criar modelos SysML, incluindo a exportação do diagrama gerado na forma gráfica para um arquivo no formato XMI, que é o formato de entrada para execução das transformações em ATL. O Diagrama de Blocos inicial para a transformação é apresentado

na forma gráfica na Figura 4.9 e parte do arquivo XMI que representa este diagrama é apresentado na Figura 4.10.

```
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_8mWbEho5EeOL9_bOzv28hQ" value="*"/>
      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_8mWbExo5EeOL9_bOzv28hQ" value="1"/>
    </ownedAttribute>
  </packagedElement>
  <packagedElement xmi:type="uml:Association" xmi:id="_8mVM8Bo5EeOL9_bOzv28hQ" name="Association9" memberEnd="_8mWbERo5EeOL9_b
    <eAnnotations xmi:id="_8mV0ABo5EeOL9_bOzv28hQ" source="org.eclipse.papyrus">
      <details xmi:id="_8mWbEBo5EeOL9_bOzv28hQ" key="nature" value="SysML_Nature"/>
    </eAnnotations>
  </packagedElement>
  <packagedElement xmi:type="uml:Class" xmi:id="_Ar26YCn9EeOwc_6KHRWQqA" name="Sensor"/>
  <profileApplication xmi:id="_yiXSEBSxEeOAW4VNWaoDDg">
    <eAnnotations xmi:id="_yibjgBSxEeOAW4VNWaoDDg" source="http://www.eclipse.org/uml2/2.0.0/UML">
      <references xmi:type="ecore:EPackage" href="http://www.eclipse.org/papyrus/0.7.0/SysML#/"/>
    </eAnnotations>
    <appliedProfile href="pathmap://SysML_PROFILES/SysML.profile.uml#_TZ_nULU5EduiKqCzJMWbGw"/>
  </profileApplication>
  <profileApplication xmi:id="_yrVn4BSxEeOAW4VNWaoDDg">
  <profileApplication xmi:id="_yrh1IBSxEeOAW4VNWaoDDg">
  <profileApplication xmi:id="_yricMBSxEeOAW4VNWaoDDg">
  <profileApplication xmi:id="_yricMhSxEeOAW4VNWaoDDg">
  <profileApplication xmi:id="_yrjDOBSxEeOAW4VNWaoDDg">
  <profileApplication xmi:id="_yrjqUBSxEeOAW4VNWaoDDg">
  <profileApplication xmi:id="_yrkRYBSxEeOAW4VNWaoDDg">
  <profileApplication xmi:id="_yrk4cBSxEeOAW4VNWaoDDg">
  <profileApplication xmi:id="_yrk4chSxEeOAW4VNWaoDDg">
  <profileApplication xmi:id="_yrlfgRSxEeOAW4VNWaoDDg">
</uml:Model>
<Blocks:ValueType xmi:id="_2jV0MBSxEeOAW4VNWaoDDg" base_DataType="_2drREBSxEeOAW4VNWaoDDg"/>
<Blocks:ValueType xmi:id="_BXUesBSzEeOAW4VNWaoDDg" base_DataType="_BXRbYBSzEeOAW4VNWaoDDg"/>
<Blocks:Block xmi:id="_ZcmKsBS1EeOAW4VNWaoDDg" base_Class="_ZcigUBS1EeOAW4VNWaoDDg"/>
<Blocks:Block xmi:id="_by82sBS4EeOAW4VNWaoDDg" base_Class="_by6acBS4EeOAW4VNWaoDDg"/>
<Blocks:Block xmi:id="_gyLv0BTDEeOyhY1Qsz49GA" base_Class="_gwxakBTDEeOyhY1Qsz49GA"/>
<Blocks:Block xmi:id="_M9r1IBTEEeOyhY1Qsz49GA" base_Class="_M9pY4BTEEeOyhY1Qsz49GA"/>
<Blocks:Block xmi:id="_cbXrIBTEEeOyhY1Qsz49GA" base_Class="_cbVO4BTEEeOyhY1Qsz49GA"/>
<Blocks:Block xmi:id="_riqlgBoWEeOL9_bOzv28hQ" base_Class="_riXDgBoWEeOL9_bOzv28hQ"/>
<Blocks:Block xmi:id="_2sCGoBoWEeOL9_bOzv28hQ" base_Class="_2r4VoBoWEeOL9_bOzv28hQ"/>
<Blocks:Block xmi:id="_P3SOIRoXEeOL9_bOzv28hQ" base_Class="_P3SOIBoXEeOL9_bOzv28hQ"/>
```

Figura 4.10: Parte do arquivo XMI do Diagrama de Blocos da SysML.

Com o Diagrama de Blocos criado e exportado para o formato XMI, o ambiente de execução da ATL, integrado à ferramenta TopCased, é configurado para a execução da transformação. A tela de configuração da transformação é apresentada na Figura 4.11 e uma breve explicação de cada parâmetro é apresentada a seguir.

- ATL Module: Nome do arquivo que contém as regras de transformação. Este arquivo possui a extensão .atl;
- Metamodels: Neste campo são configurados os metamodelos que irão participar da transformação. Neste caso é passado o metamodelo do Diagrama de Blocos da SysML e o metamodelo da UML, onde será utilizado o metamodelo do Diagrama de Classes.
- Source Models: Caminho do arquivo XMI que contém o(s) modelo(s) de entrada da transformação, neste caso o caminho do arquivo XMI do Diagrama de Blocos de entrada é configurado;
- Target Models: Caminho do arquivo XMI que irá conter o(s) modelo(s) de saída da transformação, neste caso o nome do Diagrama de Classes de saída é configurado.

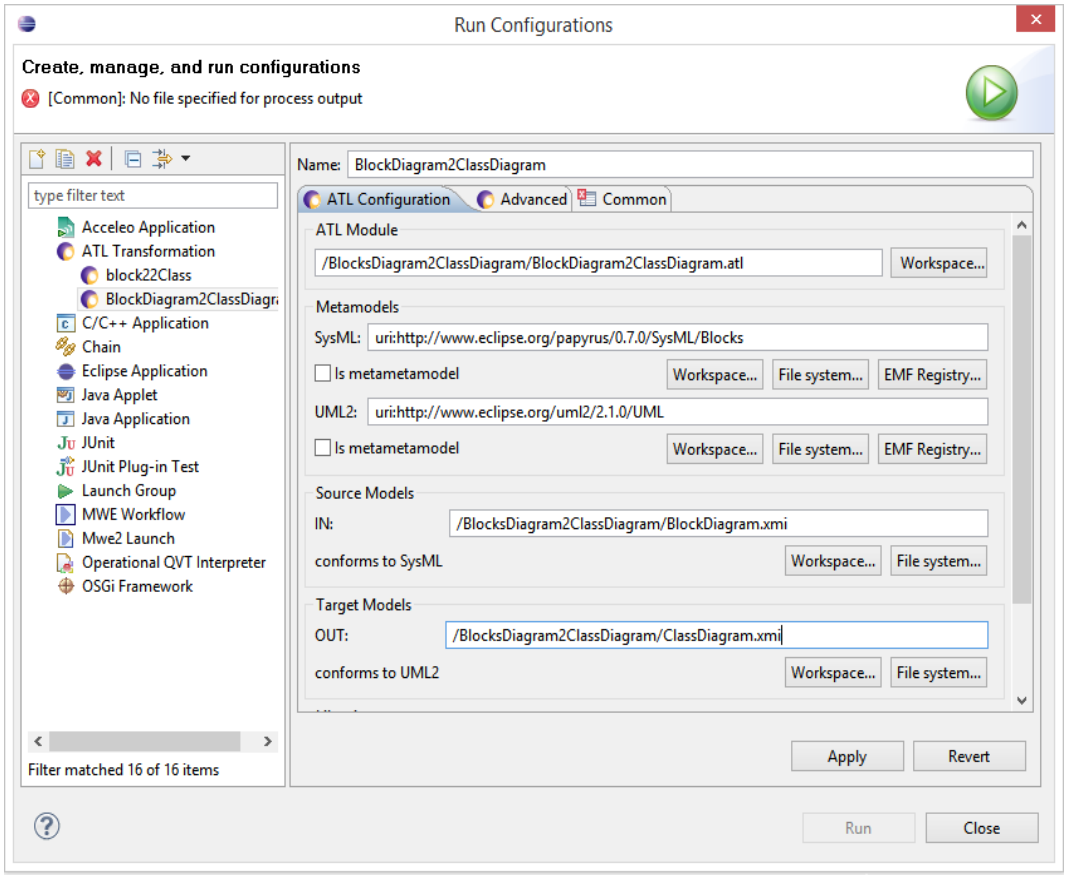

Figura 4.11: Configuração ATL para execução da transformação BlockDiagram2ClassDiagram.

Após a execução da transformação, um arquivo XMI representando o Diagrama de Classes da UML é gerado. Parte do arquivo XMI gerado pela transformação é apresentado na Figura 4.12.

O arquivo XMI gerado pela transformação precisa então ser importado por uma ferramenta de modelagem com o objetivo de exibir o Diagrama de Classes na forma gráfica. O Papyrus, ferramenta utilizada para a modelagem do Diagrama de Blocos inicial, não possui a funcionalidade de importação de arquivos XMI. Esta é uma funcionalidade facilmente encontrada em ferramentas comerciais. Para realizar a importação foram testadas duas ferramentas comerciais: Astah Professional e Enterprise Architect. Nestas duas ferramentas o Diagrama de Classes foi importado e exibido em seu formato gráfico. O Diagrama de Classes gerado pela transformação, e importado pelas ferramentas comerciais, é apresentado na Figura 4.13.

Com o propósito de avaliar o reuso de transformações existentes no diagrama gerado pela transformação proposta, o Diagrama de Classes da UML gerado por essa transformação foi submetido a uma transformação *Model-to-Code*, já implementada na ferramenta TopCased. O objetivo desta transformação é gerar código-fonte de forma automática. Foram testadas duas transformações no diagrama de classes gerado: Transformação de

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<uml:Model xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:uml="http://www.eclipse.org/uml2/3.0.0/UML"
  <packagedElement xmi:type="uml:DataType" xmi:id="_YEHscQeHEeSEIelKiudLTw" name="Scenario">
    <ownedAttribute xmi:id="_YEHscgeHEeSEIelKiudLTw" name="start_date" visibility="private">
      <type xmi:type="uml:Class" href="pathmap://UML_METAMODELS/UML.metamodel.uml#DataType"/>
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_YEHscweHEeSEIelKiudLTw" value="1"/>
      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_YEHsdAeHEeSEIelKiudLTw" value="1"/>
    </ownedAttribute>
    <ownedAttribute xmi:id="_YEHsdQeHEeSEIelKiudLTw" name="end_date " visibility="private">
      <type xmi:type="uml:Class" href="pathmap://UML_METAMODELS/UML.metamodel.uml#DataType"/>
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_YEHsdgeHEeSEIelKiudLTw" value="1"/>
      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_YEHsdweHEeSEIelKiudLTw" value="1"/>
    </ownedAttribute>
  </packagedElement>
  <packagedElement xmi:type="uml:DataType" xmi:id="_YEHseAeHEeSEIelKiudLTw" name="svg"/>
  <packagedElement xmi:type="uml:Class" xmi:id="_YEHseQeHEeSEIelKiudLTw" name="ODMGR">
    <ownedAttribute xmi:id="_YEHsegeHEeSEIelKiudLTw" name="Destination" visibility="private" aggregation="composite">
      <type xmi:type="uml:PrimitiveType" href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_YEHseweHEeSEIelKiudLTw" value="1"/>
      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_YEHsfAeHEeSEIelKiudLTw" value="1"/>
    </ownedAttribute>
    <ownedAttribute xmi:id="_YEHsfQeHEeSEIelKiudLTw" name="Origin" visibility="private" aggregation="composite">
      <type xmi:type="uml:PrimitiveType" href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_YEHsfgeHEeSEIelKiudLTw" value="1"/>
      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_YEHsfweHEeSEIelKiudLTw" value="1"/>
    </ownedAttribute>
    <ownedAttribute xmi:id="_YEHsgAeHEeSEIelKiudLTw" name="controlscheme" visibility="private" type="_YEHshgeHEeSEIelKiudLTw"
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_YEHsgQeHEeSEIelKiudLTw" value="1"/>
```

Figura 4.12: Parte do arquivo XMI do Diagrama de Classes da UML.

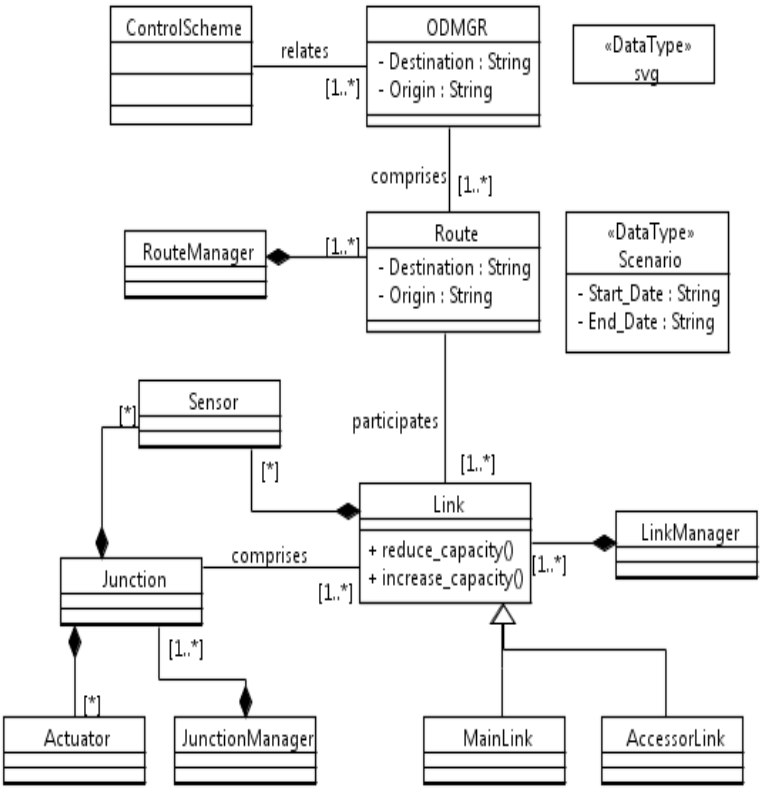

Figura 4.13: Visão lógica usando diagrama de classes UML.

Diagrama de Classes da UML para código-fonte Java e a transformação de Diagrama de Classes UML para código-fonte Python. Parte do código-fonte Java e Python da classe Link, gerada pela ferramenta TopCased, é apresentada na Figura 4.14.

```java
public class Link{

        private Sensor[] sensors = {};
        private Route route ;
        private Junction junction ;
        private LinkManger linkmanager ;
        // Start of user code to add fields for Link

        /**
         * Return sensors.
         * @return sensors
         */
        public Sensor[] getSensors() {
            return sensors;
        }
```

```python
class Link:
    def __init__ (self):
        # sensors : Sensor
        self.sensors = SysMLmodel.Sensor()
        # route : Route
        self.route = SysMLmodel.Route()
        # junction : Junction
        self.junction = SysMLmodel.Junction()
        # linkmanager : LinkManger
        self.linkmanager = SysMLmodel.LinkManger()
        #Start of user code for Link.__init__
        pass
        #End of user code
```

Figura 4.14: Código-fonte gerado automaticamente pela ferramenta TopCased.

Em relação ao sistema de exemplo apresentado nesta seção, os objetos sensores e atuadores existem apenas pois pertencem a um objeto de junção. O mesmo vale para os objetos sensores relacionados com objetos Links. Como resultado, eles são todos representados com uma associação de composição. Particularmente neste domínio específico, outros compartimentos de um bloco SysML não tiveram um mapeamento simples. Assim, cada caso deve ser analisado cuidadosamente. Por exemplo, o Controle de Esquema é realmente refinado para uma classe de controle, responsável por implementar os cenários propostos. Cada cenário apresentado no "compartimento cenário" é projetado como um Caso de Uso, e implementado em software usando as Classes UML.

O mapeamento proposto, que é a base para o a transformação e, posteriormente, da implementação desta transformação usando a ATL está representado na Figura 4.1, e o resultado final do mapeamento de blocos SysML para classes UML usando o mapeamento proposto é apresentado na Figura 4.13.

4.4 Discussão e Resultados

Depois de mais de uma década de uso em uma variedade de domínios, tanto na academia quanto na indústria, o número de sistemas legados modelados utilizando a UML é considerável. Portanto, mesmo com alguns problemas bem conhecidos, a linguagem ainda tem sido aplicada consideravelmente a novos projetos. De fato, a introdução de uma linguagem completamente diferente seria um desafio por muitas razões. Novas ferramentas de modelagem teriam de ser integradas na metodologia de desenvolvimento. O processo de aprendizagem pela equipe de desenvolvimento e o treinamento devem ser considerados. Por esta razão, o valor acrescentado de uma nova linguagem de modelagem deve ser claro.

É difícil encontrar uma única linguagem de modelagem que é capaz de modelar tanto o software quanto os elementos do sistema. O objetivo deste trabalho é descrever uma pesquisa e sua aplicação na qual o Diagrama de Blocos da SysML é introduzido para criar modelos de *Software-Intensive Systems* em combinação com o Diagrama de Classes da UML. Como o Diagrama de Blocos da SysML é útil para modelar componentes de um sistema, tais como os equipamentos e suas partes, este diagrama pode ser aplicado para modelar outros elementos que não são software. Um mapeamento entre metamodelos que descreve a relação entre o Diagrama de Blocos da SysML e o Diagrama de Classes da UML é apresentado. Uma abordagem dirigida a modelos usando a linguagem ATL é usada para implementar o mapeamento, afim de transformar automaticamente um Diagrama de Blocos da SysML em um Diagrama de Classes da UML. Esta abordagem traz melhor separação de interesses durante o projeto do sistema, e fornece uma maneira simples de rastrear elementos de sistemas para elementos de software. Além disso, o conhecimento sobre o mapeamento de elementos do sistema para elementos de software pode aproximar o trabalho de engenheiros de sistemas e engenheiros de software. A avaliação baseou-se na aplicação prática no desenvolvimento de um *Software-Intensive System* na área de gestão do tráfego rodoviário.

Capítulo 5

Transformação Dinâmica usando ATL (IBD2Activity)

Este capítulo tem como objetivo transformar um diagrama da SysML específico, o Diagrama de Blocos Internos, em um diagrama da UML específico, o Diagrama de Atividades. Assim, engenheiros de sistemas podem trabalhar em conjunto com os engenheiros de software para projetar sistemas complexos da engenharia, usando linguagens complementares com o mesmo metamodelo. Um resultado importante deste trabalho foi propor um mapeamento para descrever a relação entre o Diagrama de Blocos Internos da SysML e o Diagrama de Atividades da UML, relação esta que não foi descrita na especificação da SysML. Além disso, o outro resultado foi a implementação dessa relação por meio de uma abordagem orientada a modelos. Uma transformação automática, usando a linguagem ATL, foi realizada com base no mapeamento descrito. Esta transformação deve ser realizada de tal forma a preservar todas as informações do Diagrama de Blocos Internos no Diagrama de Atividades.

5.1 Relacionando Diagrama de Blocos Internos SysML com Diagrama de Atividades UML

A transformação foi proposta após uma análise do Diagrama de Blocos Internos da SysML e a identificação de um fluxo de informações entre as partes que compõem o bloco principal e/ou entre as partes e o próprio bloco principal. A hipótese é que, usando as partes, o bloco principal e os seus fluxos, é possível criar uma transformação automática que gera um Diagrama de Atividades da UML a partir de um Diagrama de Blocos Internos da SysML, preservando todas as informações contidas no diagrama original.

A transformação baseia-se na identificação das entradas e saídas de informações de cada bloco (partes ou bloco principal) através das portas e das *Flow Properties* definidas nas portas. Usando esta identificação é possível definir um fluxo de informações entre

cada bloco integrante do Diagrama de Blocos Internos e transformá-lo em um Diagrama de Atividades relevante.

Portas de entrada e de saída definidas no bloco principal do IBD podem ser consideradas estados inicial e final, respectivamente, de um Diagrama de Atividades, e se transformam em *ActivityParameterNodes* do Diagrama de Atividades. Cada parte é transformada em uma ação do Diagrama de Atividades, e cada porta relacionada com cada parte é transformada em um pino de entrada ou um pino de saída da ação, dependendo da propriedade de fluxo relacionada com a porta do bloco. As ações são conectadas usando os pinos e os conectores de acordo com as conexões existentes entre as partes do Diagrama de Blocos Internos da SysML. Se existe uma ligação entre uma porta de saída de uma parte e uma porta de entrada de outra parte, então existe uma ligação entre um pino de saída de uma ação e um pino de entrada de outra ação. Em alguns casos, pode ocorrer a ligação entre os pinos e *ActivityParameterNodes*.

O mapeamento entre os metamodelos da transformação é apresentado na Figura 5.1, onde é possível observar quais elementos do Diagrama de Atividades são gerados automaticamente a partir dos elementos do Diagrama de Blocos Internos.

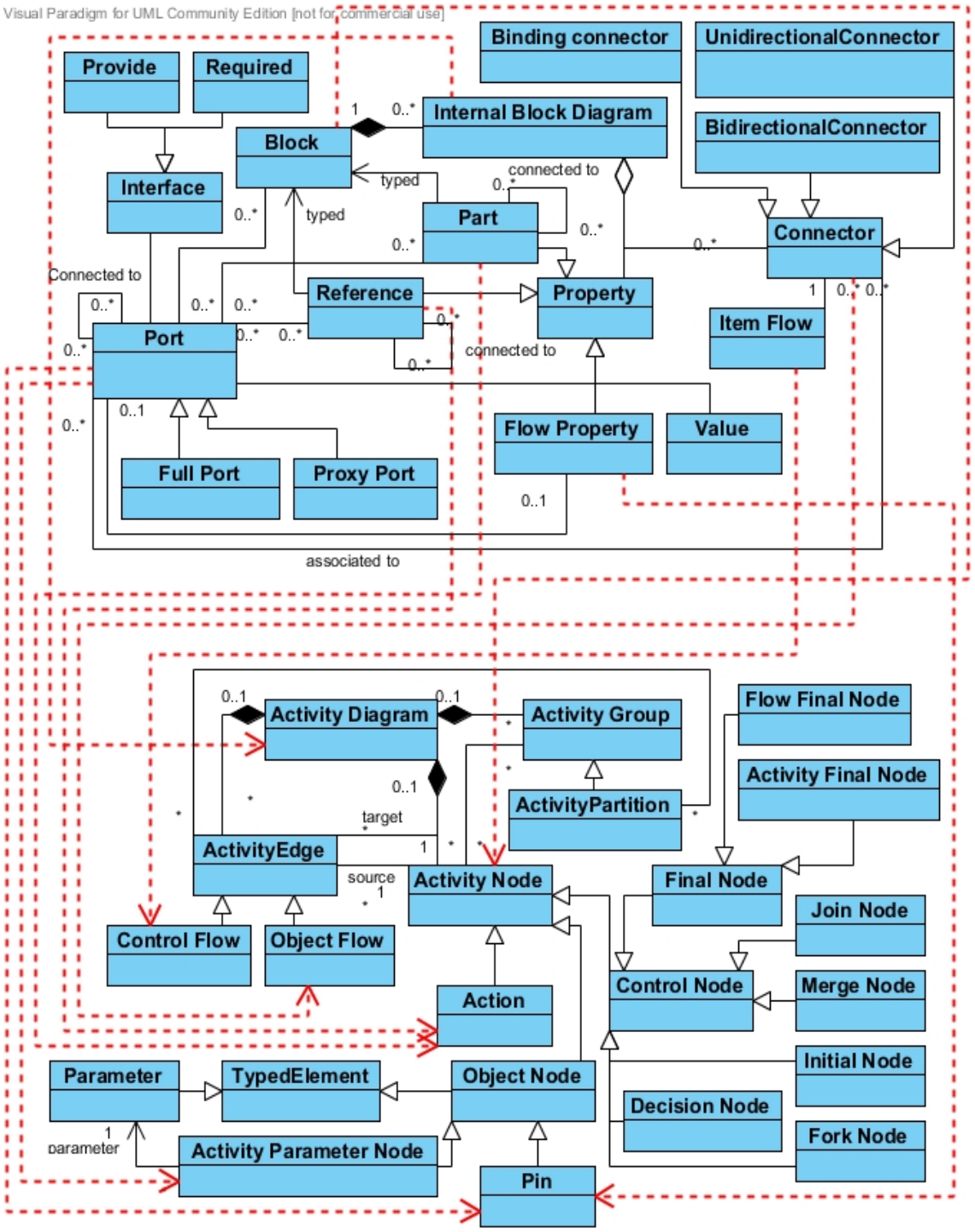

Figura 5.1: Mapeamento da transformação de Diagrama de Blocos Internos da SysML para Diagrama de Atividades da UML.

5.2 Regras especificadas em ATL

A ATL é escolhida como a linguagem de transformação de modelos para implementar a transformação de Diagramas de Blocos Internos da SysML para Diagramas de Atividades da UML. Esta transformação se mostrou mais complexa que a anterior, assim somente regras não foram suficientes para implementar esta transformação. Logo, foi necessária a

Tabela 5.1: *Helpers* criados em ATL

Rules	Descrição
AllINFlowPorts	Busca no IBD todas portas de entrada de dados de todos os blocos
AllOUTFlowPorts	Busca no IBD todas portas de saída de dados de todos os blocos
isINPort	Verifica se uma determinada porta, passada por parâmetro, é uma porta de entrada de dados
isOUTPort	Verifica se uma determinada porta, passada por parâmetro, é uma porta de saída de dados
getAllInputPin	Busca no diagrama de atividades todos Pinos de entrada já transformados
getAllOutputPin	Busca no diagrama de atividades todos Pinos de saída já transformados
getAllActivityParameterNode	Busca no diagrama de atividades todos ActivityParameterNodes já transformados
getActivityParameterNode	Recupera o ActivityParameterNode gerado pela transformação da Porta 'p' passada como parâmetro
getInputPin	Recupera os Pinos de entrada gerado pela transformação da Porta 'p' passada como parâmetro
getOutputPin	Recupera os Pinos de saída gerado pela transformação da Porta 'p' passada como parâmetro
getPortsByConnector	Busca todas portas vinculadas a uma parte através das informações contidas nos conectores

implementação de alguns *Helpers* para auxiliar a implementação das regras ATL.

Helpers em ATL podem ser vistos como métodos em uma linguagem de programação orientada a objetos. Esses *Helpers* são usados para definir o código-fonte ATL que pode ser chamado em diferentes partes da transformação, mesmo dentro de outros *Helpers*. Para a transformação de um Diagrama de Blocos Internos para um Diagrama de Atividades, foram criados onze *Helpers*. Cada *Helper* é apresentado, de forma breve, na Tabela 5.1 e de uma forma mais detalhada a seguir.

Alguns *Helpers* implementados nesta transformação são relativamente simples e têm como objetivo recuperar um conjunto de elementos específicos do IBD. O *Helper* AllINFlowPorts tem como objetivo a busca de todas as portas de entrada de todos os blocos do Diagrama de Blocos Internos da SysML. Este *Helper* busca cada porta presente no IBD verificando se a *Flow Property* associada a porta é do tipo *IN*, ou seja, que a porta é de fato uma porta de entrada de dados. Este *Helper* foi criado para auxiliar um outro *Helper*, o isINPort. O *Helper* isINPort tem como objetivo verificar se uma porta específica, passada como parâmetro, é uma porta de entrada. Na Figura 5.2, os *Helpers* AllINFlowPorts e isINPort são apresentados.

Na implementação da transformação, para cada *Helper* criado para lidar com as portas

```
helper def: AllINFlowPorts: Set(IBD!FlowPort) =
    IBD!FlowPort->allInstances()->
        select(e | e.direction.toString().equals('in') )
;

helper def: isINPort(p:IBD!Port): Boolean =
    if thisModule.AllINFlowPorts->
       select(e | e.base_Port = p).size() > 0 then
        true
    else
        false
    endif;
```

Figura 5.2: *Helpers* AllInFlowPorts e isINPort implementados em ATL.

de entrada, um *Helper* semelhante é criado para lidar com as portas de saída. Neste caso, foi criado o *Helper* AllOUTFlowPorts para recuperar todas portas de saída de todos blocos do IBD e o *Helper* isOUTPort para verificar se uma porta específica, passada como parâmetro, é uma porta de saída. Os *Helpers* AllOUTFlowPorts e isOUTPort são apresentados na Figura 5.3.

```
helper def: AllOUTFlowPorts: Set(IBD!FlowPort) =
    IBD!FlowPort->allInstances()->
        select(e | e.direction.toString().equals('out') )
;

helper def: isOUTPort(p:IBD!Port): Boolean =
    if thisModule.AllOUTFlowPorts->
        select(e | e.base_Port = p).size() > 0 then
        true
    else
        false
    endif;
```

Figura 5.3: *Helpers* AllOUTFlowPorts e isOUTPort implementados em ATL.

Foram necessários também *Helpers* que buscam informações de elementos no Diagrama de Atividades que está sendo gerado pela transformação. Estes *Helpers* são necessários para identificação dos pinos de entrada, saída ou *ActivityParametersNodes* gerado pela respectiva porta. O objetivo é identificar os pinos que fazem parte da ligação entre as ações do Diagrama de Atividades, de acordo com as ligações existentes entre as portas do Diagrama de Blocos Internos da SysML.

O *Helper* getAllInputPin foi criado para buscar todos os pinos de entrada já criados no Diagrama de Atividades. Este *Helper* é usado por outro *Helper*, o getInputPin, que tem como objetivo identificar o pino de entrada (*InputPin*) gerado pela transformação de uma porta específica, passada como parâmetro. Os *Helpers* getAllInputPin e getInputPin são apresentados na Figura 5.4.

Para cada *Helper* criado para tratar um pino de entrada, foi necessário um novo *Helper* para tratar um pino de saída. O *Helper* getAllOutputPin foi criado para buscar todos os pinos de saída do Diagrama de Atividades gerado. O *Helper* getOutputPin utiliza

```
helper def: getAllInputPin(): Set(ACT!InputPin)=
    ACT!InputPin->allInstances();

helper def: getInputPin(p:IBD!FlowPort):ACT!InputPin =
    thisModule.getAllInputPin()->
        select(c | c.name = p.name+':'+p.type.name).first();
```

Figura 5.4: *Helpers* getAllInputPin e getInputPin implementados em ATL.

o *Helper* getAllOutputPin e tem como objetivo identificar o pino de saída (*OutputPin*) gerado pela transformação de uma porta específica, passada como parâmetro. Os *Helpers* getAllOutputPin e getOutputPin são apresentados na Figura 5.5.

```
helper def: getAllOutputPin(): Set(ACT!InputPin)=
    ACT!OutputPin->allInstances();

helper def: getOutputPin(p:IBD!FlowPort):ACT!InputPin =
    thisModule.getAllOutputPin()->
        select(c | c.name = p.name+':'+p.type.name).last();
```

Figura 5.5: *Helpers* getAllOutputPin e getOutputPin implementados em ATL.

Como portas podem também ser transformadas em *Activity Parameter Nodes*, caso elas estejam vinculadas ao bloco principal do IBD, são necessários *Helpers* que tratam esse caso específico. Assim, foi criado o *Helper* getAllActivityParameterNode para busca de todos *Activity Parameter Nodes* já transformados no Diagrama de Atividades e o *Helper* getActivityParameterNode que recupera o *Activity Parameter Node* gerado pela transformação de uma porta específica, passada como parâmetro. O *Helper* getAllActivityParameterNode é utilizado pelo *Helper* getActivityParameterNode que por sua vez auxilia a *Lazy Rule* Connector2Edge. Os *Helpers* getAllActivityParameterNode e getActivityParameterNode são apresentados na Figura 5.6.

```
helper def: getAllActivityParameterNode() : Set(ACT!ActivityParameterNode)=
    ACT!ActivityParameterNode->allInstances();

helper def: getActivityParameterNode(p:IBD!Port):ACT!ActivityParameterNode =
    thisModule.getAllActivityParameterNode()->  select(c | c.name.toString()
              .equals(p.name.toString()+':'+p.type.name.toString())).first();
```

Figura 5.6: *Helpers* getAllActivityParameterNode e getActivityParameterNode implementados em ATL.

O *Helper* getPortsByConnector tem como objetivo pesquisar todas portas vinculadas a uma parte usando as informações contidas nos conectores. Essa pesquisa é necessária pois as informações das portas estão vinculadas a um bloco genérico e não a parte específica. Assim, se duas partes possuem o mesmo tipo (um bloco genérico), as informações de portas das partes se misturam, não sendo possível recuperar as portas de uma

determinada parte diretamente. Esta informação é obtida apenas usando a identificação de quais portas são utilizadas por um determinado conector que realiza a ligação entre as partes. Se o conector conecta uma parte com outra parte, essa conexão é feita usando duas portas específicas. Em seguida, utilizando o conector, é possível recuperar as portas que pertencem a cada uma das partes e realizar a transformação usando as regras INPort2InputPin ou OUTPort2OutputPin, dependendo se a porta é de entrada ou de saída. A regra Part2Action utiliza o *Helper* getPortsByConnector para auxiliar na transformação. O Helper getPortsByConnector é apresentado na Figura 5.7.

```
helper def: getPortsByConnector(b:IBD!Property): Set(IBD!Port)=
    IBD!Connector->allInstances()->iterate(i; res: Set(IBD!Port) = Set{} |
    if i.end.first().partWithPort = b then
        res.including(i.end.first().role)
    else
        if i.end.last().partWithPort = b then
            res.including(i.end.last().role)
        else
            res
        endif
    endif

);
```

Figura 5.7: *Helper* getPortsByConnector implementado em ATL.

Helpers são usados basicamente para auxiliar regras, que são de fato quem realizam as transformações da ATL. Nelas especificamos a transformação de como um elemento de origem gera um elemento de destino. Para a transformação de um Diagrama de Blocos Internos da SysML para um Diagrama de Atividades da UML, foram criadas sete regras (uma regra e seis *Lazy Rules*). Todas regras são apresentadas de forma resumida na Tabela 5.2 e de forma detalhada a seguir.

A primeira regra criada é novamente a regra Model2Model, que realiza a transformação do modelo do Diagrama de Blocos Internos para o modelo do Diagrama de Atividades. Esta é uma regra simples que basicamente possui dois objetivos: adicionar no nome do Diagrama de Atividades o nome do Diagrama de Blocos Internos e chamar a Lazy Rule MainBlock2Activity. A regra Model2Model é apresentada na Figura 5.8.

```
rule Model2Model {
    from
        i : IBD!Model (i.oclIsTypeOf(IBD!Model))
    to
        a : ACT!Model (
            name <- i.name,
            packagedElement <- thisModule.MainBlock2Activity(
                                    i.packagedElement.first())
        )
}
```

Figura 5.8: Regra Model2Model implementada em ATL.

Tabela 5.2: Regras criadas em ATL

Rules	Descrição
Model2Model	Transforma o modelo de IBD da SysML para um modelo de Diagrama de Atividades da UML
MainBlock2Activity	Transforma o primeiro bloco encontrado (Bloco Principal) na atividade principal do Diagrama de Atividades
Part2Action	Transforma atributos que representam as partes em ações do Diagrama de Atividades
Port2ActivityParameterNode	Transforma as portas do bloco principal em *Activivy Parameter Nodes*
INPort2InputPin	Transforma as portas de entrada de cada parte em Pinos de entrada associado a parte
OUTPort2OutputPin	Transforma as portas de saída de cada parte em Pinos de saída associado a parte
Connector2Edge	Transforma conectores do IBD em ligações entre ações do Diagrama de Atividades

A *Lazy Rule* MainBlock2Activity é uma das principais regras da transformação. Esta *Lazy Rule* é responsável por transformar o bloco principal do IBD na Atividade principal do Diagrama de Atividades. Esta é uma regra complexa que chama outras três *Lazy Rules*: Part2Action, Port2ActivityParameterNode e Connector2Edge. A *Lazy Rule* MainBlock2Activity é apresentada na Figura 5.9.

```
lazy rule MainBlock2Activity{
    from
        i : IBD!Block
    to
        a : ACT!Activity(
                name <- i.name,
                node <- i.ownedAttribute->
                        select(o| not o.oclIsTypeOf(IBD!Port))->
                        collect(e | thisModule.Part2Action(e)),
                node <- i.ownedAttribute->
                        select(o| o.oclIsTypeOf(IBD!Port))->
                        collect(e | thisModule.Port2ActivityParameterNode(e)),
                edge <- i.ownedConnector->
                        collect(c | thisModule.Connector2Edge(c))
        )
}
```

Figura 5.9: *Lazy Rule* MainBlock2Activity implementada em ATL.

A *Lazy Rule* Part2Action, apresentada na Figura 5.10, tem como objetivo gerar as ações no Diagrama de Atividades a partir das partes do Diagrama de Blocos Internos. É uma regra mais complexa pois exige uma análise mais detalhada dos elementos envolvidos. Três *Helpers* são usados para auxiliar a implementação desta regra: getPortsByConnector, isINPort e isOUTPort. Após selecionar as portas de entrada ou de saída, as *Lazy Rules* InPort2InputPin e OUTPort2OutputPin são chamadas para transformar estas portas em pinos de entrada ou pinos de saída, respectivamente.

```
lazy rule Part2Action{
        from
            i : IBD!Property
        to
            a : ACT!OpaqueAction(
                name <- i.name +':'+i.type.name ,
                inputValue <- thisModule.getPortsByConnector(i)->
                              select(p | thisModule.isINPort(p))->
                              collect(e | thisModule.INPort2InputPin(e)),
                outputValue <- thisModule.getPortsByConnector(i)->
                               select(p | thisModule.isOUTPort(p))->
                               collect(e | thisModule.OUTPort2OutputPin(e))
            )
}
```

Figura 5.10: *Lazy Rule* Part2Action implementada em ATL.

As *Lazy Rules* INPort2InputPin e OUTPort2OutputPin buscam transformar portas de um determinado bloco (parte) em pinos na ação correspondente. A *Lazy Rule* INPort2InputPin transforma portas de entrada do bloco em pinos de entrada na ação correspondente e a *Lazy Rule* OUTPort2OutputPin transforma portas de saída de um determinado bloco em pinos de saída da ação correspondente. Essas transformações são realizadas de forma simples e direta, pois cada porta é transformada em um pino, de entrada ou de saída, com o mesmo nome e valores de limite inferior e superior da porta. A lógica necessária para selecionar a porta e a *lazy rule* corretas para a transformação é realizada em outra *Lazy Rule*, a Part2Action. As *Lazy Rules* INPort2InputPin e OUTPort2OutputPin são apresentadas na Figura 5.11.

```
lazy rule INPort2InputPin{
    from
        i : IBD!Port
    to
        a : ACT!InputPin (
            name <- i.name+':'+i.type.name,
            upper <- i.upper,
            lower <- i.lower
        )
}
```

```
lazy rule OUTPort2OutputPin{
    from
        i : IBD!Port
    to
        a : ACT!OutputPin (
            name <- i.name+':'+i.type.name,
            upper <- i.upper,
            lower <- i.lower
        )
}
```

Figura 5.11: *Lazy Rule* INPort2InputPin e OUTPort2OutputPin implementadas em ATL.

A *Lazy Rule* Port2ActivityParameterNode tem como objetivo transformar as portas do bloco principal em *Actitivy Parameter Nodes* da Atividade correspondente. Esta *Lazy Rule* é chamada pela *Lazy Rule* MainBlock2Activity, onde também é realizada a lógica necessária para selecionar as portas corretas. A *Lazy Rule* Port2ActivityParameterNode é apresentada na Figura 5.12.

Por fim, a *Lazy Rule* Connector2Edge tem como objetivo transformar os conectores do IBD em ligações entre ações no Diagrama de Atividades da UML. Nesta regra são utilizados os *Helpers* getActivityParameterNode, getInputPin e getOutputPin para iden-

```
lazy rule Port2ActivityParameterNode{
        from
            i : IBD!Port
        to
            a : ACT!ActivityParameterNode(
                name <-  i.name+':'+i.type.name
            )

}
```

Figura 5.12: *Lazy Rule* Port2ActivityParameterNode implementada em ATL.

tificar o início e o fim de cada conector para que possam ser geradas as ligações entre as ações perfeitamente. A busca realizada pelos *Helpers* é feita por meio do nome da porta de entrada ou de saída onde cada conector do IBD é ligado, por isso a necessidade de manter o mesmo nome das portas nos pinos de entrada, de saída e nos *Activity Parameter Nodes*. A *Lazy Rule* Connector2Edge é apresentada na Figura 5.13.

```
lazy rule Connector2Edge{
    from
        i : IBD!Connector
    to
        a : ACT!ObjectFlow (
            name <-  i.name+':'+i.type.name,
            source <- thisModule.getActivityParameterNode(i.end.first().role),
            target <- thisModule.getActivityParameterNode(i.end.last().role),
            source <- thisModule.getOutputPin(i.end.first().role),
            target <- thisModule.getInputPin(i.end.last().role)
        )
}
```

Figura 5.13: *Lazy Rule* Connector2Edge implementada em ATL.

A próxima seção busca aplicar todas regras implementadas usando ATL em um estudo de caso onde um Diagrama de Atividades da UML é gerado a partir de um Diagrama de Blocos Internos da SysML já existente.

5.3 Estudo de caso realizado

Um estudo de caso é descrito nesta seção com o objetivo principal de apresentar o funcionamento da transformação proposta. O Diagrama de Blocos Internos selecionado é o diagrama criado para o problema de um destilador de água. Este problema foi proposto para a Equipe SysML por D. Oliver em Dezembro de 2005 e sua solução apresentada na Incose (*International Council on Systems Engineering*) de 2006. Sua solução também é apresentada em [Friedenthal et al. 2008], no capítulo 16. Este estudo de caso foi escolhido por ser um diagrama já conhecido dos usuários SysML, por ser um diagrama gerado pela Equipe SysML e estar disponível em sites oficiais da SysML. O estudo de caso também possui um pré-requisito importante: possuir um fluxo de dados bem definido. Esta ca-

racterística é necessária para que o Diagrama de Atividades gerado seja um diagrama relevante, mantendo as informações do diagrama original.

O destilador de água é um sistema para purificação de água suja. Este sistema é descrito a seguir, e seu comportamento é ilustrado na Figura 5.14:

- O aquecimento da água suja e a condensação do vapor são realizadas por um *Counter Flow Heat Exchanger*.
- Fervura de água suja é realizada por uma caldeira (*Boiler*).
- A drenagem de resíduo é realizada pela drenagem (*Drain*)
- A água tem propriedades: vol = 1 litro, a densidade de 1 gm/cm3, temperatura de 20°C, calor específico 1cal/gm °C, o calor de vaporização 540 cal/gm.

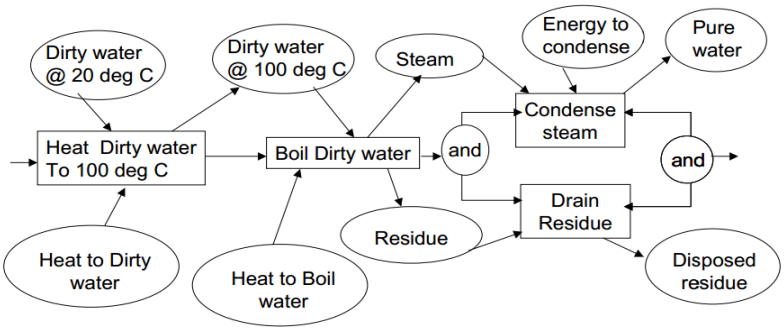

Figura 5.14: Comportamento do sistema waterDistiller.

O Diagrama de Blocos Internos selecionado mostra a estrutura interna do bloco destilador (*Distiller*) com um elemento adicional, a alimentação do tipo válvula (*Valve*). O bloco destilador é composto de quatro partes: hx1:HeatExchange, feed:Valve, bx1:Boiler e drain:Valve, conforme apresentado na Figura 5.15. O Diagrama de Blocos Internos gerado pela Equipe SysML mostra como essas partes estão relacionadas, quais são as entradas e saídas de cada parte envolvida e o fluxo de informações entre as partes que compõem o destilador.

Novamente, a ferramenta Papyrus, integrada à ferramenta TopCased, foi utilizada para a modelagem do Diagrama de Blocos Internos inicial da transformação. O IBD gerado no formato gráfico, apresentado na Figura 5.15, foi exportado para um arquivo com extensão XMI necessário para execução da transformação. Parte do arquivo XMI do Diagrama de Blocos Internos gerado é apresentado na Figura 5.16.

A ferramenta TopCased também foi utilizada para programação e execução da transformação em ATL. Na transformação de modelos usando a ATL, o arquivo XMI é interpretado pela transformação que executa as regras já implementadas, gerando o diagrama resultante, neste caso, gerando o Diagrama de Atividades equivalente. A configuração

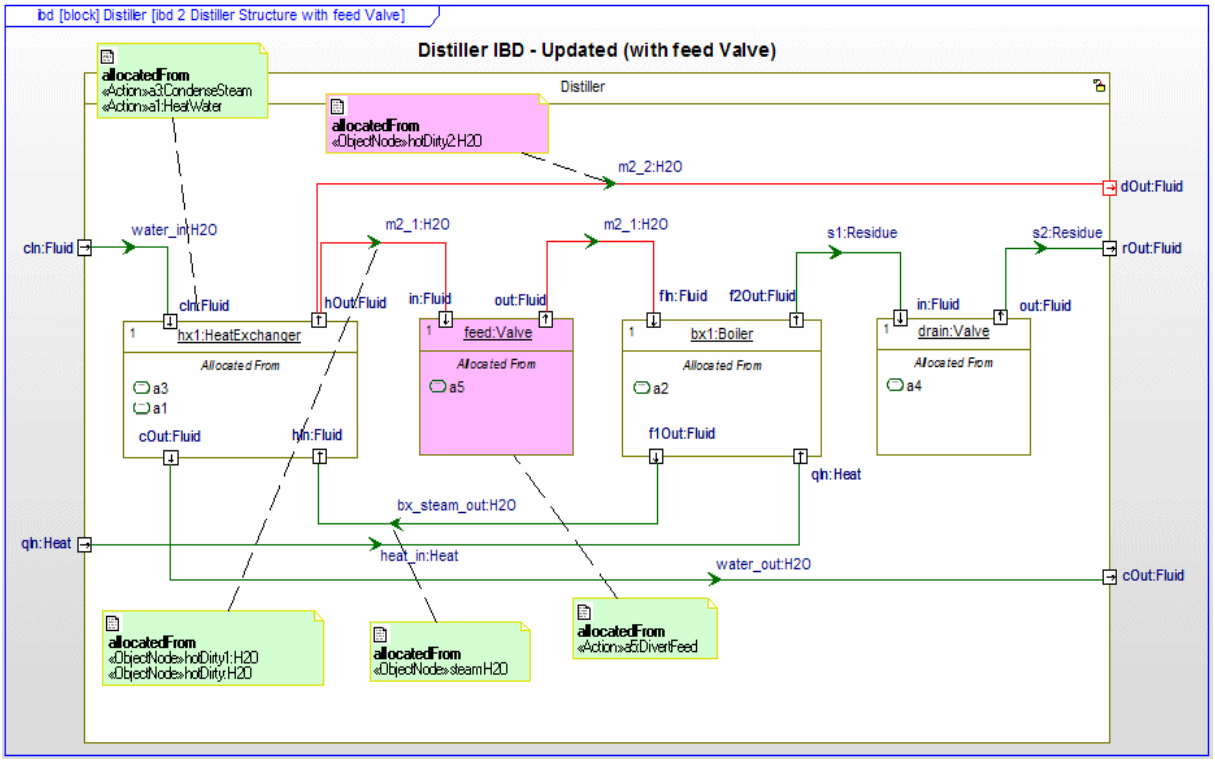

Figura 5.15: Diagrama de Blocos Internos da SysML do Bloco Destilador.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  <uml:Model xmi:id="_kp0BMZokEeOmno2e_3D-nA" name="SysMLmodel">
    <packagedElement xmi:type="uml:Class" xmi:id="_lLE7MJokEeOmno2e_3D-nA" name="Distiller">
      <ownedAttribute xmi:type="uml:Port" xmi:id="_zNP1cJokEeOmno2e_3D-nA" name="cIn" type="_WxuvsJonEeOmno2e_3D-nA" a
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_zNUG4JokEeOmno2e_3D-nA" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_zNTf0JokEeOmno2e_3D-nA" value="1"/>
      </ownedAttribute>
      <ownedAttribute xmi:type="uml:Port" xmi:id="_U0JeIJomEeOmno2e_3D-nA" name="qIn" type="_EOjvcJonEeOmno2e_3D-nA" a
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_U0KFMZomEeOmno2e_3D-nA" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_U0KFMJomEeOmno2e_3D-nA" value="1"/>
      </ownedAttribute>
      <ownedAttribute xmi:type="uml:Port" xmi:id="_YitlIJonEeOmno2e_3D-nA" name="dOut" type="_WxuvsJonEeOmno2e_3D-nA"
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_YiwocJonEeOmno2e_3D-nA" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_YiwBYJonEeOmno2e_3D-nA" value="1"/>
      </ownedAttribute>
      <ownedAttribute xmi:type="uml:Port" xmi:id="_Y1LOIJonEeOmno2e_3D-nA" name="rOut" type="_WxuvsJonEeOmno2e_3D-nA"
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_Y1L1MZonEeOmno2e_3D-nA" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_Y1L1MJonEeOmno2e_3D-nA" value="1"/>
      </ownedAttribute>
      <ownedAttribute xmi:type="uml:Port" xmi:id="_ZHLkIJonEeOmno2e_3D-nA" name="cOut" type="_WxuvsJonEeOmno2e_3D-nA"
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_ZHMLMZonEeOmno2e_3D-nA" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_ZHMLMJonEeOmno2e_3D-nA" value="1"/>
      </ownedAttribute>
      <ownedAttribute xmi:id="_pEul8JonEeOmno2e_3D-nA" name="hx1" type="_v3QpsJonEeOmno2e_3D-nA" aggregation="composit
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_v3QCoJonEeOmno2e_3D-nA" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_v3PbkJonEeOmno2e_3D-nA" value="1"/>
```

Figura 5.16: Parte do arquivo XMI do Diagrama de Blocos Internos da SysML.

para execução da transformação de Diagramas de Blocos Internos da SysML para Diagrama de Atividades da UML é apresentada na Figura 5.17 e cada parâmetro é explicado brevemente a seguir.

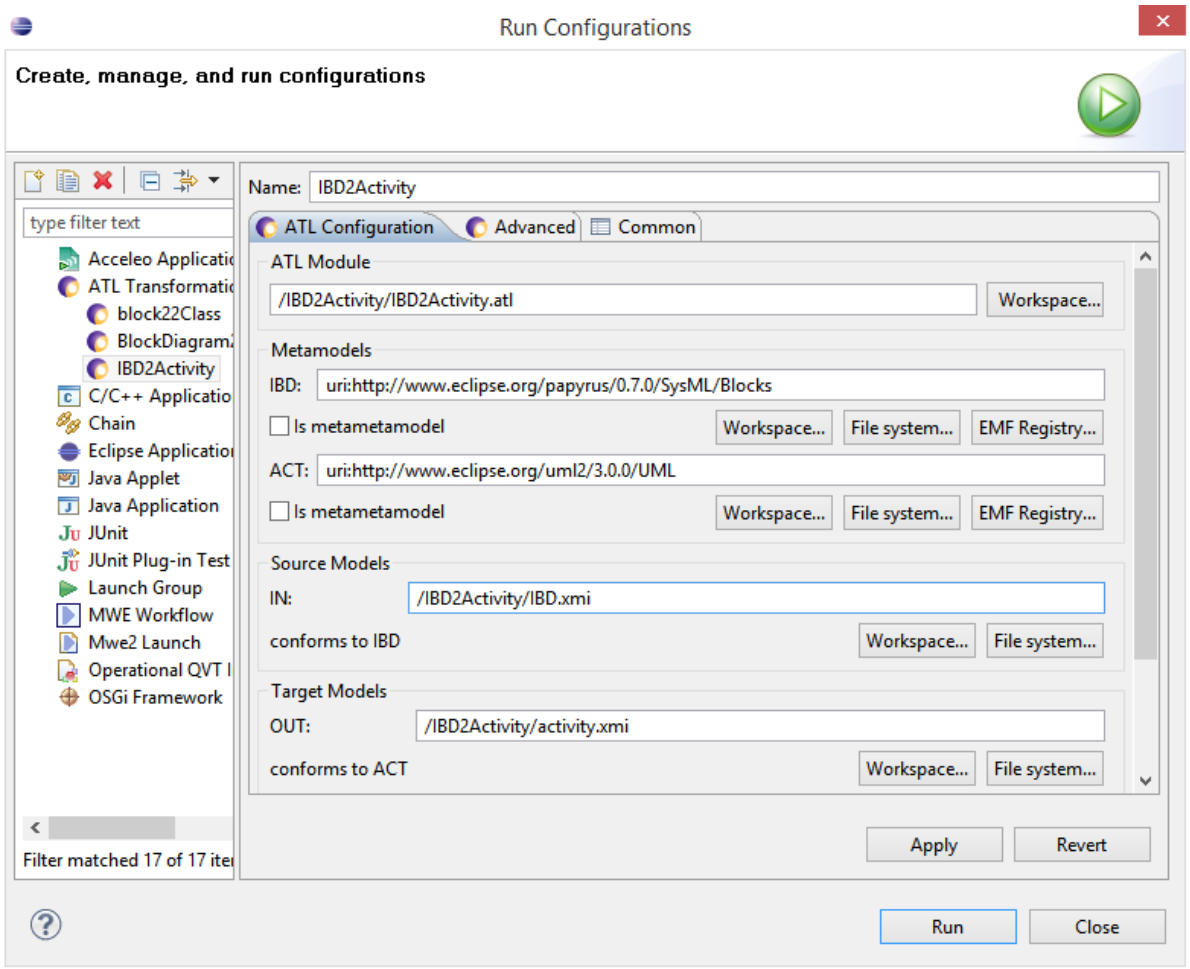

Figura 5.17: Configuração ATL para execução da transformação IBD2ActivityDiagram.

.

- ATL Module: Nome do arquivo que contém as regras de transformação. Este arquivo possui a extensão .atl;
- Metamodels: Neste campo são configurados os metamodelos que irão participar da transformação. Neste caso é passado o metamodelo do Diagrama de Blocos Internos da SysML e o metamodelo da UML, onde será utilizado o metamodelo do Diagrama de Atividades.
- Source Models: Caminho do arquivo XMI que contém o(s) modelo(s) de entrada da transformação, neste caso o caminho do arquivo XMI do Diagrama de Blocos Internos de entrada é configurado;
- Target Models: Caminho do arquivo XMI que irá conter o(s) modelo(s) de saída da transformação, neste caso o nome do Diagrama de Atividades de saída é configurado.

O Diagrama de Atividades é gerado após a execução da transformação, como um arquivo com extensão XMI. Parte do arquivo XMI gerado pela transformação é apresentado na Figura 5.18. Para que este diagrama seja apresentado no seu formato gráfico, é necessário importá-lo por uma ferramenta de modelagem. A ferramenta Papyrus, usada para modelar o diagrama de entrada, exporta o diagrama de entrada do modo gráfico para o formato XMI, necessário para a execução da transformação. No entanto, a ferramenta não tem a funcionalidade para importar o diagrama resultante no formato XMI para o formato gráfico.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<uml:Model xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:uml="http://www.eclips
  <packagedElement xmi:type="uml:Activity" xmi:id="_JDVsMZ2MEeOnROr6anhNwg" name="Distiller">
    <node xmi:type="uml:OpaqueAction" xmi:id="_JDVsMp2MEeOnROr6anhNwg" name="hx1:HeatExchanger">
      <inputValue xmi:id="_JDVsM52MEeOnROr6anhNwg" name="cIn:Fluid" incoming="_JDVsX52MEeOnROr6anhNwg">
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_JDVsNJ2MEeOnROr6anhNwg" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_JDVsNZ2MEeOnROr6anhNwg" value="1"/>
      </inputValue>
      <inputValue xmi:id="_JDVsNp2MEeOnROr6anhNwg" name="hIn:Fluid" incoming="_JDVsZJ2MEeOnROr6anhNwg">
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_JDVsN52MEeOnROr6anhNwg" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_JDVsOJ2MEeOnROr6anhNwg" value="1"/>
      </inputValue>
      <outputValue xmi:id="_JDVsOZ2MEeOnROr6anhNwg" name="hOut:Fluid" outgoing="_JDVsYJ2MEeOnROr6anhNwg _
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_JDVsOp2MEeOnROr6anhNwg" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_JDVsO52MEeOnROr6anhNwg" value="1"/>
      </outputValue>
      <outputValue xmi:id="_JDVsPJ2MEeOnROr6anhNwg" name="cOut:Fluid" outgoing="_JDVsZp2MEeOnROr6anhNwg"
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_JDVsPZ2MEeOnROr6anhNwg" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_JDVsPp2MEeOnROr6anhNwg" value="1"/>
      </outputValue>
    </node>
    <node xmi:type="uml:OpaqueAction" xmi:id="_JDVsP52MEeOnROr6anhNwg" name="feed:Valve">
      <inputValue xmi:id="_JDVsQJ2MEeOnROr6anhNwg" name="in:Fluid" incoming="_JDVsYJ2MEeOnROr6anhNwg">
        <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_JDVsQZ2MEeOnROr6anhNwg" value="1"/>
        <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_JDVsQp2MEeOnROr6anhNwg" value="1"/>
      </inputValue>
```

Figura 5.18: Parte do arquivo XMI do Diagrama de Blocos Internos da SysML.

O Diagrama de Atividades gerado pela transformação é apresentado, no formato gráfico, na Figura 5.19. A geração deste diagrama no formato gráfico foi realizada de forma totalmente manual. Visto que a ferramenta Papyrus e outras ferramentas comerciais testadas não importaram o diagrama gerado pela transformação, o arquivo XMI foi interpretado e o Diagrama de Atividades resultante gerado manualmente. Este resultado é discutido na próxima seção.

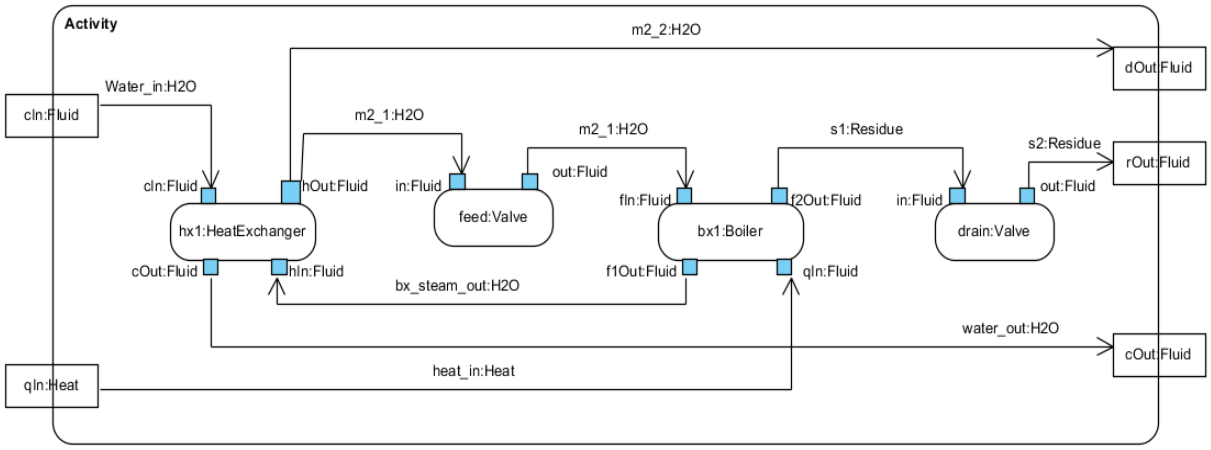

Figura 5.19: Diagrama de Atividades gerado automaticamente pela transformação.

5.4 Discussão e Resultados

A transformação é interessante devido a dois fatos. Primeiro, a transformação proposta transforma um diagrama estrutural, especificamente a parte estrutural com fluxo bem definido, em um diagrama comportamental do sistema. Em segundo lugar, a transformação proposta transforma um diagrama da SysML, que tem como foco engenheiros de sistemas, para um diagrama da UML, que tem como foco engenheiros de software e desenvolvedores.

A transformação não pode ser totalmente automatizada, isto é, não há uma correspondência de um-para-um entre os elementos dos diagramas envolvidos. Isso ocorre porque o Diagrama de Blocos Internos de entrada para a transformação deve ser necessariamente um diagrama que possui algumas características bem definidas, incluindo portas vinculadas ao bloco principal indicando o início e o fim do fluxo e fluxos de dados entre os blocos (partes) bem definido. Existem também alguns elementos presentes no Diagrama de Blocos Internos, como a representação interfaces (fornecidas e necessárias) e conectores simples entre os blocos, que não apresentam nenhuma informação de fluxo de informações e não possuem relevância suficiente para ser transformados em algum elemento do Diagrama de Atividades, onde as ações e informações de ações de fluxo são importantes.

No caso de haver um fluxo bem definido de dados e não existirem pontos iniciais e finais bem definidas para o fluxo, a transformação pode ser efetuada de uma forma semi-automática. Neste caso, o engenheiro de sistema deve identificar o início e o fim do fluxo no diagrama de atividades gerado. Devido a estes problemas, um desafio interessante é identificar quais Diagramas de Blocos Internos da SysML poderiam ser transformados para Diagramas de Atividades da UML, onde produziriam Diagramas de Atividades relevantes e com informações satisfatórias.

Um problema gerado pela ferramenta utilizada, o Papyrus, é a abertura do diagrama resultante automaticamente em seu formato gráfico. A ferramenta utilizada permite ao

usuário exportar o diagrama de entrada, gerado na forma gráfica, para um arquivo com extensão XMI (ou UML), necessário para execução da transformação. No entanto, a ferramenta não permite que o usuário importe o diagrama gerado pela transformação, com a extensão XMI, para seu formato gráfico de forma automática, tornando o processo de abertura do diagrama resultante totalmente manual.

Este problema poderia ser resolvido usando uma ferramenta que permite ao usuário importar arquivos em uma extensão XMI e gerar o diagrama gráfico automaticamente, mas durante a pesquisa as ferramentas testadas não conseguiram importar o Diagrama de Atividades gerado. Três ferramentas comerciais foram analisadas, Enterprise Architect, Astah Professional e Visual Paradigm, e nenhuma delas conseguiu importar o arquivo no formato XMI e gerar o Diagrama de Atividade no formato gráfico. Assim, o Diagrama de Atividades no formato gráfico foi gerado de forma manual, onde o arquivo XMI foi analisado e o Diagrama de Atividades foi sendo gerado.

5.5 Considerações Finais

Um mapeamento entre metamodelos da transformação e, posteriormente, a implementação da transformação do Diagrama de Blocos Internos SysML para o Diagrama de Atividades da UML é apresentado nesta seção. Em seguida, aplicou-se a proposta em um estudo de caso descrito pela Equipe SysML. Este estudo de caso é escolhido para este trabalho pois este diagrama é conhecido dos usuários SysML e está disponível em sites oficiais da SysML. O foco do trabalho é facilitar os trabalhos realizados por engenheiros de sistemas, com modelagem SysML, e engenheiros de software, com modelagem UML. Em seguida, os sistemas de software são projetados, tanto do ponto de vista de sistemas, como também do ponto de vista do software. A desvantagem da transformação é que não pode ser totalmente automatizada, isto é, não há uma correspondência de um-para-um entre os elementos dos diagramas envolvidos. Em alguns casos, o engenheiro de sistemas deve apontar o início e o fim do fluxo no diagrama de atividades gerado. No entanto, os problemas encontrados nesta transformação não são considerados problemas graves.

Um resultado interessante é que outras transformações podem ser reusadas no diagrama de atividades gerado. Por exemplo, a partir do diagrama de atividades UML gerado, o desenvolvedor pode propor outras transformações, como *Model-to-Code*, ou mesmo gerar um modelo formal, como por exemplo, Redes de Petri. Assim, o modelo inicial projetado como um Diagrama de Blocos Internos pode ser simulado e verificado formalmente. Estas atividades serão realizadas em trabalhos futuros.

Capítulo 6

Comparação com trabalhos relacionados

Este trabalho será comparado a outros trabalhos semelhantes encontrados na literatura. A obtenção dos trabalhos utilizados para validação foi feita por meio de uma busca pelas bibliotecas digitais da IEEE e ACM. Foi realizada uma busca por trabalhos que definam uma transformação de modelos, preferencialmente implementada utilizando a linguagem de transformação ATL, onde a partir do modelo SysML de origem, um modelo UML deveria ser gerado. Assim foram utilizadas como palavras-chave: *Model Transformation*, *ATL*, *UML* e *SysML*; filtrando por trabalhos de 2008 a 2014.

Durante as buscas, não foi encontrado um grande número de trabalhos que atendiam os critérios estabelecidos. Assim, foi necessário expandir a consulta para transformações que geravam modelos de qualquer linguagem de modelagem a partir de modelos SysML. Neste caso, foi eliminada a palavra-chave *UML* dos critérios de busca e o número de trabalhos encontrados melhorou consideravelmente. A Tabela 6.1 apresenta uma comparação os principais trabalhos encontrados durante as pesquisas. Estes trabalhos são descritos brevemente logo em seguida. nesta dissertação.

Após a fase de buscas, o único trabalho encontrado propondo a transformação de modelos SysML para modelos UML foi o trabalho [Lasalle et al. 2011]. Este trabalho é o que mais se assemelha ao trabalho proposto. O trabalho [Lasalle et al. 2011] é baseado na geração e execução de casos de testes de forma automática a partir de diagramas SysML. Os autores utilizam uma ferramenta já existente, a *Smartesting Test Design*, que realiza o trabalho de gerar e executar casos de testes de diagramas UML4MBT, um *profile* UML com restrições para desenvolvimento de diagramas voltados para a geração de teste. Para gerar casos de testes a partir de diagramas SysML, os autores definiram uma transformação *Model-to-Model* dos diagramas SysML para diagramas UML4MBT, com o objetivo de utilizar esta ferramenta para geração e execução dos testes. Para realizar a transformação, foi necessária a criação de um *profile* SysML, chamado SysML4MBT, que inclui algumas restrições na SysML voltados para geração de testes e limita os diagra-

Tabela 6.1: Tabela comparativa de trabalhos relacionados

Autor	Origem	Destino	Tipo Transformação	Linguagem Transformação
[Lasalle et al. 2011]	SysML	UML	Model-to-Model	Java
[Bouquet et al. 2012]	SysML	VHDL-AMS	*Model-to-Model* e *Model-to-Code*	ATL e Xpand
[McGinnis e Ustun 2009]	SysML	Arena	*Model-to-Model*	ATL
[Batarseh e McGinnis 2012]	SysML	Arena	*Model-to-Model*	ATL
[Foures et al. 2011]	SysML	Redes de Petri	*Model-to-Model*	ATL
[Foures et al. 2012]	SysML	VHDL-AMS	*Model-to-Model* e *Model-to-Code*	ATL
[Huang et al. 2012]	SysML	*Timed Automata*	*Model-to-Model*	-
[Hammad et al. 2013]	SysML	Modelica	*Model-to-Model* e *Model-to-Code*	ATL e Acceleo
[Berrani et al. 2013]	SysML	Modelica	*Model-to-Model* e *Model-to-Code*	ATL e Acceleo

mas da SysML a um subconjunto de diagramas que são importantes para a criação dos diagramas da UML4MBT. Os diagramas presentes na SysML4MBT são: Diagrama de Blocos, o Diagrama de Blocos Internos, o Diagrama de Máquina de Estados e o Diagrama de Requisitos da SysML. Os diagramas SysML4MBT são transformados em diagramas UML4MBT, usando a transformação proposta, e estes diagramas são passados como entrada para a ferramenta *Smartesting Test Design*, onde os casos de testes são gerados e executados de forma automática. A transformação foi implementada de forma direta, utilizando a linguagem JAVA.

A transformação de modelos proposta por Lasalle é voltada para geração de casos de teste automáticos a partir de diagramas SysML. Apesar de algumas semelhanças serem notadas entre o trabalho proposto por Lasalle e este trabalho, principalmente na transformação de Diagrama de Blocos para Diagrama de Classes, em vários outros aspectos os dois trabalhos são bem diferentes. No trabalho proposto por Lasalle, os autores buscam transformar o Diagrama de Blocos Internos em partes do Diagrama de Classes, além de implementar a transformação proposta de forma direta, utilizando a linguagem Java. No trabalho proposto nesta dissertação, o Diagrama de Blocos Internos , que é um diagrama estrutural, é transformado em um Diagrama de Atividades, que é diagrama comportamental, o que torna a transformação mais interessante. Todas as transformações propostas nesta dissertação foram implementadas utilizando uma linguagem específica para transformação de modelos, a ATL. Também não foi necessária a criação de um *profile* para realização das transformações, todos modelos utilizados no trabalho seguem o metamodelo definido pela OMG das linguagens envolvidas.

Grande parte dos trabalhos encontrados na literatura propõem transformações de modelos SysML para modelos que possam ser simulados e verificados formalmente de forma automática. Alguns trabalhos se assemelham entre si por possuírem duas etapas para realização da transformação. A primeira etapa é a transformação *Model-to-Model*, onde o modelo SysML é transformado em um modelo de uma linguagem específica. A segunda etapa, que acontece em alguns trabalhos, é a transformação *Model-to-Code*. Nesta etapa, o modelo gerado na primeira etapa é transformado para código de uma linguagem específica. Este é o caso do trabalho [Bouquet et al. 2012].

No trabalho [Bouquet et al. 2012], foi desenvolvida uma transformação de diagramas estruturais SysML, mais especificadamente o Diagrama de Blocos e o Diagrama de Blocos Internos, para código VHDL-AMS. Para implementação da transformação, foi utilizada a linguagem ATL para transformação *Model-to-Model* e a linguagem Xpand para geração do código-fonte. Para validar o trabalho foi proposto um estudo de caso, onde uma *Smart Surface* foi modelada com SysML e seu código VHDL-AMS foi gerado automaticamente. Para geração de código são necessários dois passos. O primeiro é realizar uma transformação *Model-to-Model* dos diagramas SysML para modelos VHDL-AMS utilizando a linguagem ATL. O segundo passo é gerar o código VHDL-AMS a partir dos modelos gerados na transformação anterior utilizando a linguagem Xpand.

Já nos trabalhos [McGinnis e Ustun 2009] e [Batarseh e McGinnis 2012] são propostas transformações de diagramas SysML para a linguagem de simulação Arena. Em todos estes trabalhos foi utilizada a linguagem de transformação ATL para realização da transformação. O objetivo destes trabalhos é automatizar a criação de modelos de simulação de eventos discretos a sistemas modelados com SysML.

O trabalho [McGinnis e Ustun 2009] foi o primeiro trabalho encontrado que aborda a transformação de modelos SysML para modelos de simulação Arena. Os autores buscam definir as regras para transformar uma linguagem de domínio específico, neste caso foi utilizada a linguagem SysML, para uma linguagem de simulação de eventos discretos, onde a linguagem escolhida foi a Arena. A transformação foi implementada de forma automática usando a linguagem de transformação ATL e um pequeno estudo de caso foi aplicado à transformação proposta.

No trabalho [Batarseh e McGinnis 2012] é proposto um *profile* SysML para sistemas de montagem eletrônica. Este *profile* também foi implementado em uma ferramenta com o objetivo de ajudar os *Stakeholders* a manipular e criar os modelos SysML para seus sistemas de forma mais fácil e rápida. A transformação de SysML para Arena foi implementada usando a linguagem de transformação ATL, e permite criar modelos de simulação Arena a partir dos modelos SysML de forma automática. Esta transformação automática reduziu tempo e custo para obter os resultados da simulação e ajudou na tomada de decisões pelos *stakeholders*. Para validação do trabalho, a transformação foi aplicada a um estudo de casos na área de sistemas de montagem eletrônica, fornecido

por um parceiro da indústria. Também foi realizada uma comparação entre a proposta apresentada pelos autores e as práticas comuns usadas pela indústria.

O trabalho [Foures et al. 2011] é um pouco diferente dos demais, pois neste trabalho os autores buscam transformar o Diagrama de Atividades SysML para uma linguagem formal, as Redes de Petri. A transformação é realizada de forma automática, sendo implementada utilizando a linguagem de transformação ATL. Uma parte importante do trabalho foi o uso do formalismo TINA que permite verificar formalmente se as propriedades do Diagrama de Atividades são preservadas. Esta transformação permite também a possibilidade de reusar outras transformações já existentes, como a transformação de Redes de Petri para VHDL-AMS, proposta por outro autor, para simular todo o modelo. Segundo os autores, a adição destas duas abordagens permite validar a parte discreta e contínua dos diagramas de atividade, e, portanto, prever as características funcionais do sistema. Os autores utilizaram um estudo de caso simples para validar apenas a transformação proposta pelo trabalho.

No trabalho [Foures et al. 2012], os autores propõem reusar duas transformações de modelos com o objetivo de propor uma abordagem para simular diagramas de Atividades SysML. Para isso, os autores transformam um Diagrama de Atividades para um modelo da linguagem formal Redes de Petri. Esta transformação foi proposta no trabalho [Foures et al. 2011], apresentado anteriormente. As Redes de Petri geradas pela primeira transformação são transformadas para VHDL-AMS, por meio da transformação proposta no trabalho [Albert et al. 2005]. Todas as transformações propostas pelos autores foram implementadas utilizando a linguagem de transformação ATL. Neste trabalho foi aplicado um estudo de caso mais completo e complexo, mais próximo da realidade da indústria, para validar todo processo necessário para a simulação de Diagrama de Atividades SysML proposto pelos autores.

No trabalho [Huang et al. 2012], os autores buscam transformar um diagrama de comportamento da SysML, o Diagrama de Máquina de Estados, para *Timed Automata* (TA). Segundo os autores, por se tratar de uma linguagem semi-formal, a SysML não permite ser verificada convenientemente e eficientemente, especialmente em sistemas embarcados de tempo real. Para resolver este problema, os autores estendem o Diagrama de Máquina de Estados com elementos de tempo e probabilidade extraídos do MARTE, um *profile* UML especificado para modelagem de sistemas embarcados de tempo real. Com o objetivo de verificar formalmente o Diagrama de Máquina de Estados da SysML, os autores propõem uma metodologia automatizada utilizando transformação de modelos. A solução proposta no trabalho é realizar uma transformação de modelos do Diagrama de Máquina de Estados SysML, com extensões de tempo e probabilidade extraídas do MARTE, para a linguagem formal *Timed Automata* (TA). Após a transformação, o modelo formal *Timed Automata* (TA) é simulado e verificado formalmente utilizando uma ferramenta especializada existente. Para realizar tal verificação formal é utilizado o Uppaal, um ambiente

integrado para modelagem, validação e verificação de sistemas de tempo real modelados como redes de *Timed Automata* (TA). Para implementação das regras de transformação entre SysML e *Timed Automata* (TA) foi utilizada a linguagem de transformação ATL. Para validação do trabalho, os autores utilizam um estudo de caso de um sistema atual, chamado de *Telephone System* (TS). No estudo de caso, é mostrado o Diagrama de Máquina de Estados SysML que representa o sistema proposto e o modelo formal em *Timed Automata* (TA), gerado pela transformação, sendo verificado formalmente pela Uppaal.

No trabalho, apresentado brevemente em [Hammad et al. 2013] e de uma forma mais completa em [Berrani et al. 2013], os autores buscam desenvolver uma método para modelar e validar propriedades, como consumo de energia, em redes de sensores sem fio. Essa abordagem busca combinar os benefícios e facilidades da linguagem SysML com a possibilidade de simular e verificar propriedades do modelo gerado, oferecido pela linguagem Modelica. Essa abordagem foi separada em duas etapas. A primeira é oferecer uma transformação de modelos, onde diagramas estáticos, dinâmicos e de requisitos da SysML são transformados em um modelo Modelica correspondente, e a partir do modelo Modelica realizar uma transformação de modelo para código Modelica. A segunda etapa é realizar uma verificação virtual de propriedades do modelo, no trabalho é verificado o consumo de energia das redes de sensores sem fio. A abordagem busca modelar a rede de sensores sem fio utilizando um subconjunto de diagramas da SysML. Os diagramas utilizados para modelar o sistema são: Diagrama de Requisitos, Diagrama de Definição de Blocos, Diagrama de Blocos Internos, Diagrama Paramétrico e Diagrama de Máquina de Estados. A partir dos diagramas SysML modelados, é realizada a transformação *Model-to-Model*, utilizando a linguagem de transformação ATL, gerando o modelo Modelica no formato XMI, que então é transformado para código Modelica por meio de uma transformação *Model-to-Code*, utilizando a linguagem Acceleo. Assim, propriedades do sistema podem ser simuladas e verificadas por meio de testes, já que a linguagem Modelica possui essas operações, usando a ferramenta OpenModelica. Para validar todo trabalho, os autores aplicam essa abordagem a um estudo de caso de densidade de tráfego motorizado em áreas urbanas. No estudo de caso os autores aplicam todos os passos da abordagem: modelagem, transformação automática e simulação e verificação formal focando na propriedade de consumo de energia da rede de sensores.

Como pode-se notar nos trabalhos apresentados anteriormente, o uso da SysML está cada vez mais importante nas mais diferentes áreas de aplicação. Assim, trabalhos que buscam a realização de novas transformações a partir de modelos SysML estão cada vez mais frequentes e importantes para a MDE. Os trabalhos apresentados anteriormente buscam resolver um problema em uma área específica, e, na maioria das vezes, acrescentar formalismo nos diagramas semi-formais da SysML. O principal objetivo da maioria dos trabalhos é simular e verificar os diagramas de forma automática. Para isso, em alguns casos é necessária a criação de *profiles* SysML para atender a um problema específico.

Estes trabalhos apesar de serem úteis para as áreas nos quais são desenvolvidos, podem não contribuir significantemente para outras áreas.

As transformações propostas nesta dissertação, apesar de serem aplicadas a um contexto específico, são definidas de forma geral, podendo ser aplicadas a qualquer contexto. Não foi utilizado nenhum *profile* para criação das transformações, sendo utilizados os metamodelos SysML e UML definidos pela OMG. Pelo fato de utilizar o metamodelo UML definido pela OMG, pode-se reusar qualquer transformação já definida para os diagramas UML diretamente nos diagramas produzidos pela transformação automática. Atualmente, existe um grande número de transformações e ferramentas desenvolvidas para modelos UML, já que esta é *de facto* a linguagem mais popular para modelagem de software. Tais transformações e ferramentas podem ser reusadas nos modelos gerados pelas transformações propostas neste trabalho, atingindo assim um objetivo mais específico.

Capítulo 7

Conclusão

7.1 Resumo do trabalho

O trabalho foi proposto com dois objetivos principais. O primeiro foi apresentar a relação entre diagramas das linguagens de modelagem SysML e UML, considerando que esta relação não foi bem definida na documentação da SysML. Em alguns casos, essa relação ocorre de maneira simples e de fácil percepção, como no caso do Diagrama de Blocos da SysML e o Diagrama de Classes da UML (Capítulo 4), pois, como apresentado no Capítulo 2, o Diagrama de Blocos da SysML é derivado do Diagrama de Classes da UML. Porém, em outros casos essa relação não é tão aparente e pode ser bem mais complexa, como é o caso da relação entre o Diagrama de Blocos Internos da SysML e o Diagrama de Atividades da UML (Capítulo 5). O Diagrama de Blocos Internos é derivado do Diagrama de Estrutura Composta da UML mas, após alguns estudos, foi identificado um fluxo de dados entre as partes que compõem o Diagrama de Blocos Internos. Assim, o Diagrama de Blocos Internos da SysML foi relacionado com o Diagrama de Atividades da UML.

O segundo objetivo deste trabalho foi usar uma abordagem orientada a modelos para implementar esta relação como uma transformação de modelos automática. Para essa implementação foi utilizada uma linguagem específica para transformação de modelos, a ATL. A técnica de estudo de casos foi aplicada em cada transformação proposta com o objetivo de validar as transformações em aplicações reais. Para a transformação de Diagrama de Blocos da SysML para o Diagrama de Classes UML, um Diagrama de Classes foi gerado automaticamente a partir de um Diagrama de Blocos da SysML utilizado para modelar uma aplicação no domínio da gestão de tráfego rodoviário. Para a segunda transformação foi utilizado um Diagrama de Blocos Internos proposto pela Equipe da SysML no domínio de um destilador de água. O Diagrama de Atividades da UML foi gerado de forma automática. Em ambos os casos, os diagramas gerados mantiveram informações importantes dos diagramas originais.

Um estudo foi realizado em ambas as transformações com o objetivo de avaliar o reuso de transformações existentes nos diagramas gerados pelas transformações automáticas. Na transformação do Diagrama de Blocos para o Diagrama de Classes, foi gerado códigos Java e Python de forma automática do Diagrama de Classes gerado. Estas transformações estão implementadas na ferramenta TopCased, utilizada para modelagem e implementação das transformações. Na segunda transformação, de Diagramas de Blocos Internos para Diagrama de Atividades, foi apenas levantada a hipótese de geração de Redes de Petri de forma automática do Diagrama de Atividades gerado. Essa transformação foi apresentada no trabalho [Foures et al. 2011].

7.2 Contribuições

Após a realização deste trabalho, algumas contribuições foram realizadas para a comunidade científica. As principais foram:

- Aplicação da SysML como a linguagem de modelagem do processo de desenvolvimento de *Software-Intensive Systems* distribuídos de tempo-real, com foco do uso da SysML na fase de projetos, mais especificamente no projeto da visão estrutural de uma arquitetura de *Software-Intensive Systems*, onde é de extrema importância que não só o software desses sistemas seja modelado, mas também todos os elementos estruturais que compõem tal sistema.
- Definição de um mapeamento entre metamodelos com o objetivo de descrever a relação entre o Diagrama de Definição de Blocos da SysML e o Diagrama de Classes da UML e, posteriormente, a implementação, usando a linguagem ATL, dessa relação usando uma abordagem orientada a modelos.
- Definição de um mapeamento entre metamodelos com o objetivo de descrever a relação entre o Diagrama de Blocos Internos da SysML e o Diagrama de Atividades da UML e, posteriormente, a implementação, usando a linguagem ATL, dessa relação usando uma abordagem orientada a modelos.
- Realização de um estudo introdutório sobre o reuso de transformações existentes nos modelos UML gerados automaticamente a partir dos modelos da SysML. O reuso permite aplicar várias abordagens definidas aos modelos da UML em modelos da SysML, aumentando assim os domínios de aplicações das transformações propostas.

Além dos resultados diretos da dissertação, o artigo "*Model-Driven Structural Design of Software-Intensive Systems using SysML Blocks and UML Classes*" foi submetido e aceito para publicação na 16th *International Conference on Enterprise Information Systems* (ICEIS 2014) [da Silva Melo e Soares 2014].

Um segundo artigo "*A Model-Driven Approach to Transform SysML Internal Block Diagrams to UML Activity Diagrams*" está sendo finalizado e será submetido a uma conferência internacional. Um artigo inicial foi criado com o intuito de realizar uma introdução a SysML. Neste artigo, a SysML foi aplicada a um software Web da Engenharia de software. O intuito deste artigo é a avaliação da SysML na modelagem de um software, servindo assim como opção a UML na modelagem deste tipo de sistema. Este artigo está sendo melhor estruturado para submissão em uma conferência ou revista nacional.

Outros trabalhos estão sendo elaborados para publicação em conferências de forma a disseminar os resultados obtidos com a dissertação para a comunidade acadêmica de engenharia dirigida a modelos.

7.3 Trabalhos Futuros

Como a transformação de modelos representa uma atividade de extrema importância na Engenharia dirigida a modelos, e a criação de novas transformações representa uma peça chave para o aumento do uso da MDE, pretende-se como trabalhos futuros o reuso de transformações já propostas na literatura e a criação de novas transformações para os vários domínios de aplicação.

Devido a popularidade da UML, existem várias transformações que possuem como entrada modelos da UML. Criar transformações de uma linguagem específica, como a SysML, que é menos utilizada e que possui um número de transformações reduzidos em comparação com a UML, para modelos da UML, permite aplicar de forma direta qualquer transformação já proposta para a UML nessa outra linguagem, aumentando o reuso e a aplicabilidade das transformações existentes. Assim, um trabalho futuro é a definição de novas transformações de outros diagramas da SysML para diagramas da UML.

Tão importante quanto definir novas transformações para outros diagramas é o reuso de transformações existentes com o intuito de aumentar a aplicabilidade de diagramas SysML nos diversos domínios de aplicação. Assim, um outro trabalho futuro é a realização de um estudo mais aprofundado sobre o reuso de transformações de modelos. Este trabalho busca apresentar que o reuso de transformações de modelos já existentes traz grandes benefícios para a MDE e *Model Driven Architecture*, como diminuição do tempo de desenvolvimento de novos sistemas e aumento da qualidade final do sistema desenvolvido. Algumas operações como verificações e validações formais, aumento ou diminuição do nível de abstração, mudança de linguagem de modelagem para melhor entendimento e geração automática de código-fonte poderão ser aplicadas a modelos de forma indireta usando o reuso de transformações existentes. Um breve exemplo deste estudo foi apresentado neste trabalho onde diagramas SysML poderão ser transformados em código-fonte Java ou Python ou serem verificados formalmente usando Redes de Petri de forma indireta. Para isso, os diagramas SysML são transformados para diagramas UML e a partir

destes diagramas serão reusadas transformações apresentadas por outros autores para os diagramas da UML.

Referências Bibliográficas

[Albert et al. 2005] Albert, V., Nketsa, A., e Pascal, J. (2005). Towards a Metalmodel Based Approach for Hierarchical Petri Net Transformations to VHDL. In *European Simulation and Modelling Conference, Porto.*

[Allilaire et al. 2006] Allilaire, F., Bézivin, J., Jouault, F., e Kurtev, I. (2006). ATL – Eclipse Support for Model Transformation. In *Proc. of the Eclipse Technology Exchange Workshop (ETX) at ECOOP.*

[Almejalli et al. 2009] Almejalli, K., Dahal, K., e Hossain, A. (2009). An Intelligent Multi-Agent Approach for Road Traffic Management Systems. In *Control Applications, (CCA) Intelligent Control, (ISIC), 2009 IEEE*, pp. 825–830.

[Almejalli et al. 2008] Almejalli, K., Dahal, K., e Hossain, M. (2008). Real Time Identification of Road Traffic Control Measures. In Fink, A. e Rothlauf, F. (editores), *Advances in Computational Intelligence in Transport, Logistics, and Supply Chain Management*, volume 144 de *Studies in Computational Intelligence*, pp. 63–80. Springer Berlin Heidelberg.

[Anastasakis et al. 2007] Anastasakis, K., Bordbar, B., Georg, G., e Ray, I. (2007). UML2Alloy: A Challenging Model Transformation. In Engels, G., Opdyke, B., Schmidt, D., e Weil, F. (editores), *Model Driven Engineering Languages and Systems*, volume 4735 de *Lecture Notes in Computer Science*, pp. 436–450. Springer Berlin Heidelberg.

[André et al. 2007] André, C., Mallet, F., e de Simone, R. (2007). Modeling Time(s). In *10th International Conference on Model Driven Engineering Languages and Systems (MODELS '07)*, pp. 559–573.

[ATLAS 2006] ATLAS, G. (2006). *ATL User Manual.* version 0.7.

[Balmelli et al. 2006] Balmelli, L., Brown, D., Cantor, M., e Mott, M. (2006). Model-Driven Systems Development. *IBM Systems Journal*, 45(3):569–585.

[Barendrecht 2010] Barendrecht, P. (2010). Modeling Transformations Using QVT Operational Mappings. Technical report, Eindhoven University of Technology - Department of Mechanical Engineering.

[Batarseh e McGinnis 2012] Batarseh, O. e McGinnis, L. F. (2012). SysML to Discrete-Event Simulation to Analyze Electronic Assembly Systems. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, TMS/DEVS '12, pp. 48:1–48:8, San Diego, CA, USA. Society for Computer Simulation International.

[Bell 2004] Bell, A. E. (2004). Death by UML Fever. *Queue*, 2(1):72–80.

[Berrani et al. 2013] Berrani, S., Hammad, A., e Mountassir, H. (2013). Mapping SysML to Modelica to Validate Wireless Sensor Networks Non-Functional Requirements. In *Programming and Systems (ISPS), 2013 11th International Symposium on*, pp. 177–186.

[Bézivin 2005] Bézivin, J. (2005). On the Unification Power of Models. *Software & Systems Modeling*, 4(2):171–188.

[Bézivin 2006] Bézivin, J. (2006). Model Driven Engineering: An Emerging Technical Space. In Lämmel, R., Saraiva, J. a., e Visser, J. (editores), *Generative and Transformational Techniques in Software Engineering*, volume 4143 de *Lecture Notes in Computer Science*, pp. 36–64. Springer Berlin Heidelberg.

[Bézivin e Gerard 2002] Bézivin, J. e Gerard, S. (2002). A Preliminary Identification of MDA Components. In *Generative Techniques in the Context of Model Driven Architecture*.

[Bézivin et al. 2005] Bézivin, J., Jouault, F., Rosenthal, P., e Valduriez, P. (2005). Modeling in the Large and Modeling in the Small. In Abmann, U., Aksit, M., e Rensink, A. (editores), *Model Driven Architecture*, volume 3599 de *Lecture Notes in Computer Science*, pp. 33–46. Springer Berlin Heidelberg.

[Biehl 2010] Biehl, M. (2010). Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology.

[Booch 2004] Booch, G. (2004). *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[Booch et al. 2006] Booch, G., Rumbaugh, J., e Jacobson, I. (2006). *UML: Guia do Usuário*. CAMPUS - RJ.

[Bouquet et al. 2012] Bouquet, F., Gauthier, J.-M., Hammad, A., e Peureux, F. (2012). Transformation of SysML Structure Diagrams to VHDL-AMS. In *Proceedings of the 2012 Second Workshop on Design, Control and Software Implementation for Distributed MEMS*, DMEMS '12, pp. 74–81, Washington, DC, USA. IEEE Computer Society.

[Brown e McDermid 2007] Brown, A. W. e McDermid, J. A. (2007). The Art and Science of Software Architecture. In *Proceedings of the First European Conference on Software Architecture*, ECSA'07, pp. 237–256. Springer-Verlag.

[Brunéliere et al. 2010] Brunéliere, H., Cabot, J., Jouault, F., Tisi, M., Bézivin, J., et al. (2010). Industrialization of Research Tools: the ATL Case. In *Third International Workshop on Academic Software Development Tools and Techniques-WASDeTT-3 (co-located with the 25th IEEE/ACM International Conference on Automated Software Engineering-ASE'2010)*.

[Calegari e Szasz 2013] Calegari, D. e Szasz, N. (2013). Verification of Model Transformations. *Electronic Notes in Theoretical Computer Science*, 292:5–25.

[Colombo et al. 2012] Colombo, P., Khendek, F., e Lavazza, L. (2012). Bridging the Gap Between Requirements and Design: An Approach Based on Problem Frames and SysML. *Journal of Systems and Software*, 85(3):717–745.

[Czarnecki e Helsen 2003] Czarnecki, K. e Helsen, S. (2003). Classification of Model Transformation Approaches.

[Czarnecki e Helsen 2006] Czarnecki, K. e Helsen, S. (2006). Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645.

[da Silva Melo e Soares 2014] da Silva Melo, M. e Soares, M. d. S. (2014). Model-driven Structural Design of Software-intensive Systems Using SysML Blocks and UML Classes. In *16th International Conference on Enterprise Information Systems (ICEIS 2014)*.

[Delligatti 2013] Delligatti, L. (2013). *SysML Distilled: A Brief Guide to the Systems Modeling Language*. Addison-Wesley Professional, 1st edition.

[Deursen et al. 2007] Deursen, A. V., Visser, E., e Warmer, J. (2007). Model-Driven Software Evolution: A Research Agenda. In *Proceedings of the International Workshop on Model-Driven Software Evolution Held with the ECSMR'07*.

[Douglass 1997] Douglass, B. P. (1997). *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Edwards e Lee 2003] Edwards, S. A. e Lee, E. A. (2003). The Semantics and Execution of a Synchronous Block-Diagram Language. *Science of Computer Programming*, 48(1):21–42.

[Feher e Lengyel 2012] Feher, P. e Lengyel, L. (2012). The Challenges of a Model Transformation Language. In *Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on*, pp. 324–329.

[Foures et al. 2011] Foures, D., Albert, V., e C., P. J. (2011). ActivityDiagram2PetriNet: Transformation-Based Model in accordance with the OMG SysML specifications. In *Eurosis, The 2011 European Simulation and Modelling Conference, France*.

[Foures et al. 2012] Foures, D., Albert, V., Pascal, J.-C., e Nketsa, A. (2012). Automation of SysML Activity Diagram Simulation with Model-Driven Engineering Approach. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, TMS/DEVS '12, pp. 11:1–11:6, Orlando, Florida, San Diego, CA, USA. Society for Computer Simulation International.

[France et al. 2006] France, R., Ghosh, S., Dinh-Trong, T., e Solberg, A. (2006). Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer*, 39(2):59–66.

[France e Rumpe 2007] France, R. e Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pp. 37–54, Washington, DC, USA. IEEE Computer Society.

[Friedenthal et al. 2008] Friedenthal, S., Moore, A., e Steiner, R. (2008). *A Practical Guide to SysML: Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Fuentes e Vallecillo 2004] Fuentes, L. e Vallecillo, A. (2004). An Introduction to UML Profiles. *UPGRADE, The European Journal for the Informatics Professional*, 5(2):5–13.

[Gerlich et al. 2007] Gerlich, R., Sigg, D., e Gerlich, R. (2007). Model Transformation in Practice. In *DASIA 2007 - Data Systems In Aerospace*, volume 638 de *ESA Special Publication*.

[Goknil et al. 2014] Goknil, A., Kurtev, I., e Van Den Berg, K. (2014). Generation and Validation of Traces Between Requirements and Architecture Based on Formal Trace Semantics. *Journal of Systems and Software*, 88:112–137.

[Grobshtein e Dori 2009] Grobshtein, Y. e Dori, D. (2009). Creating SysML Views from an OPM Model. In *International Conference on Model-Based Systems Engineering, 2009. MBSE '09.*, pp. 36–45.

[Guedes 2009] Guedes, G. (2009). *UML 2 - Uma Abordagem Prática*. Novatec.

[Hammad et al. 2013] Hammad, A., Mountassir, H., e Chouali, S. (2013). An Approach Combining SysML and Modelica for Modelling and Validate Wireless Sensor Networks. In *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*, SESoS '13, pp. 5–12, Montpellier, France, New York, NY, USA. ACM.

[Hästbacka et al. 2011] Hästbacka, D., Vepsäläinen, T., e Kuikka, S. (2011). Model-Driven Development of Industrial Process Control Applications. *Journal of Systems and Software*, 84(7):1100 – 1113.

[Hinchey et al. 2008] Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J. P., e Margaria, T. (2008). Software Engineering and Formal Methods. *Communications of the ACM*, 51(9):54–59.

[Huang et al. 2012] Huang, X., Sun, Q., Li, J., Pan, M., e Zhang, T. (2012). An MDE-Based Approach to the Verification of SysML State Machine Diagram. In *Proceedings of the Fourth Asia-Pacific Symposium on Internetware*, Internetware '12, pp. 9:1–9:7. ACM.

[Huber 2008] Huber, P. (2008). The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches. Master's thesis, Technische Universität Wien,, Vienna, Austria.

[Jiang et al. 2008] Jiang, L., Eberlein, A., Far, B., e Mousavi, M. (2008). A Methodology for the Selection of Requirements Engineering Techniques. *Software & Systems Modeling*, 7(3):303–328.

[Jouault et al. 2008] Jouault, F., Allilaire, F., Bézivin, J., e Kurtev, I. (2008). ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39.

[Jouault e Kurtev 2006] Jouault, F. e Kurtev, I. (2006). Transforming Models with ATL. In *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, pp. 128–138. Springer-Verlag.

[Karban et al. 2008] Karban, R., Zamparelli, M., Bauvir, B., Koehler, B., Noethe, L., e Balestra, A. (2008). Exploring Model Based Engineering for Large Telescopes: Getting Started with Descriptive Models. In *SPIE Astronomical Telescopes+ Instrumentation*, pp. 70171I–70171I. International Society for Optics and Photonics.

[Kent 2002] Kent, S. (2002). Model Driven Engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM 02, pp. 286–298, London, UK. Springer-Verlag.

[Kim et al. 2012] Kim, S.-K., Myers, T., Wendland, M.-F., e Lindsay, P. A. (2012). Execution of Natural Language Requirements Using State Machines Synthesised from Behavior Trees. *Journal of Systems and Software*, 85(11):2652–2664.

[Kleppe et al. 2003] Kleppe, A. G., Warmer, J., e Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[K.Ranjini et al. 2011] K.Ranjini, A.Kanthimathi, e Y.Yasmine (2011). Design of Adaptive Road Traffic Control System through Unified Modeling Language. *International Journal of Computer Applications*, 14(7):36–41.

[Laleau et al. 2010] Laleau, R., Semmak, F., Matoussi, A., Petit, D., Hammad, A., e Tatibouet, B. (2010). A First Attempt to Combine SysML Requirements Diagrams and B. *Innovations in Systems and Software Engineering*, 6(1-2):47–54.

[Langer et al. 2014] Langer, P., Mayerhofer, T., Wimmer, M., e Kappel, G. (2014). On the Usage of UML: Initial Results of Analyzing Open UML Models. In *Modellierung*, pp. 289–304.

[Lasalle et al. 2011] Lasalle, J., Bouquet, F., Legeard, B., e Peureux, F. (2011). SysML to UML Model Transformation for Test Generation Purpose. *SIGSOFT Software Engineering Notes*, 36(1):1–8.

[Lima et al. 2007] Lima, B., de Sousa, J., e Lopes, D. (2007). Using MDA to Support Hypermedia Document Sharing. In *International Conference on Software Engineering Advances, 2007. ICSEA 2007*, pp. 65–65.

[Lopes 2007] Lopes, D. (2007). Introdução a Engenharia Dirigida por Modelos. In *Escola Regional de Computacao Ceará - Maranhão - Piauí (ERCEMAPI)*.

[Lopes 2008] Lopes, D. (2008). Engenharia Dirigida por Modelos: Abordagem e Aplicações. In *Escola Regional de Computação Ceará Maranhão - Piauí (ERCEMAPI)*.

[Ludewig 2003] Ludewig, J. (2003). Models in Software Engineering – An Introduction. *Software and Systems Modeling*, 2(1):5–14.

[marie Favre 2004] marie Favre, J. (2004). Towards a Basic Theory to Model Model Driven Engineering. In *Proceedings of the UML2004 International Workshop on Software Model Engineering*.

[McGinnis e Ustun 2009] McGinnis, L. e Ustun, V. (2009). A Simple Example of SysML-driven Simulation. In *Winter Simulation Conference*, WSC '09, pp. 1703–1710, Austin, Texas. Winter Simulation Conference.

[Mellor et al. 2004] Mellor, S. J., Kendall, S., Uhl, A., e Weise, D. (2004). *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[Meng et al. 2010] Meng, F., Chu, D., e Zhan, D. (2010). Transformation from Data Flow Diagram to UML2.0 activity diagram. In *Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on*, volume 2, pp. 1010–1014.

[Mens e Van Gorp 2006] Mens, T. e Van Gorp, P. (2006). A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142.

[Muller e Gaertner 2004] Muller, P.-A. e Gaertner, N. (2004). *Modélisation Objet avec UML*. Eyrolles, Paris.

[Ngo e Soriano 2012] Ngo, V. e Soriano, T. (2012). A Model Transformation Process to Realize Controllers of Ship Autopilot Systems by the Specialized MDA's Features with UML/SysML. In *Mechatronics (MECATRONICS) , 2012 9th France-Japan 7th Europe-Asia Congress on and Research and Education in Mechatronics (REM), 2012 13th Int'l Workshop on*, pp. 20–26.

[OMG 2003] OMG (2003). *Model Driven Architecture (MDA) Guide*. OMG doc. ab/2003-06-01.

[OMG 2011a] OMG (2011a). *Query/View/Transformation 1.1*. OMG doc. ptc/11-01-01.

[OMG 2011b] OMG (2011b). *Unified Modeling Language Specification Infrastructure v2.4.1*. OMG doc. formal/11-08-05.

[OMG 2011c] OMG (2011c). *Unified Modeling Language Specification Superstructure v2.4.1*. OMG doc. formal/11-08-06.

[OMG 2012] OMG (2012). *System Modeling Language (SysML) 1.3*. OMG doc. formal/12-06-01.

[OMG 2014a] OMG (2014a). *Meta Object Facility (MOF) 2.4.2 Core*. OMG doc. ab/2014-04-06.

[OMG 2014b] OMG (2014b). *Object Constraint Language (OCL) 2.4*. OMG doc. formal/14-02-03.

[OMG 2014c] OMG (2014c). *XML Metadata Interchange (XMI) 2.4.2*. OMG doc. formal/14-04-04.

[Paredis et al. 2010] Paredis, C. J., Bernard, Y., Burkhart, R. M., de Koning, H.-P., Friedenthal, S., Fritzson, P., Rouquette, N. F., e Schamai, W. (2010). An Overview of the SysML-Modelica Transformation Specification. In *2010 INCOSE International Symposium*.

[Qamar et al. 2009] Qamar, A., During, C., e Wikander, J. (2009). Designing Mechatronic Systems, a Model-Based Perspective, an Attempt to Achieve SysML-Matlab/Simulink Model Integration. In *Advanced Intelligent Mechatronics, 2009. AIM 2009. IEEE/ASME International Conference on*, pp. 1306–1311.

[Rothenberg 1989] Rothenberg, J. (1989). Artificial Intelligence, Simulation &Amp; Modeling. chapter The Nature of Modeling, pp. 75–92. John Wiley & Sons, Inc.

[Runeson e Höst 2009] Runeson, P. e Höst, M. (2009). Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164.

[Salem et al. 2008] Salem, R. B., Grangel, R., e Bourey, J.-P. (2008). A Comparison of Model Transformation Tools: Application for Transforming {GRAI} Extended Actigrams into {UML} Activity Diagrams. *Computers in Industry*, 59(7):682 – 693. Enterprise Integration and Interoperability in Manufacturing Systems.

[Selim et al. 2012] Selim, G. M. K., Wang, S., Cordy, J. R., e Dingel, J. (2012). Model Transformations for Migrating Legacy Models: An Industrial Case Study. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications*, ECMFA'12, pp. 90–101, Berlin, Heidelberg. Springer-Verlag.

[Sendall e Kozaczynski 2003] Sendall, S. e Kozaczynski, W. (2003). Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45.

[Shah et al. 2010] Shah, S., Anastasakis, K., e Bordbar, B. (2010). From UML to Alloy and Back Again. In Ghosh, S. (editor), *Models in Software Engineering*, volume 6002 de *Lecture Notes in Computer Science*, pp. 158–171. Springer Berlin Heidelberg.

[Shaw 2003] Shaw, M. (2003). Writing Good Software Engineering Research Papers: Minitutorial. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pp. 726–736, Portland, Oregon, Washington, DC, USA. IEEE Computer Society.

[Soares et al. 2011] Soares, M. d. S., Vrancken, J., e Verbraeck, A. (2011). User Requirements Modeling and Analysis of Software-Intensive Systems. *Journal of Systems and Software*, 84(2):328–339.

[Soares e Vrancken 2007] Soares, M. d. S. e Vrancken, J. L. M. (2007). Requirements specification and modeling through SysML. In *IEEE International Conference on Systems, Man and Cybernetics, 2007. ISIC. on*, pp. 1735–1740.

[Soares e Vrancken 2009] Soares, M. d. S. e Vrancken, J. L. M. (2009). Evaluation of UML in Practice - Experiences in a Traffic Management Systems Company. In *International Conference on Enterprise Information Systems*, pp. 313–319.

[Staines 2008] Staines, T. (2008). Intuitive Mapping of UML 2 Activity Diagrams into Fundamental Modeling Concept Petri Net Diagrams and Colored Petri Nets. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pp. 191–200.

[Tiako 2009] Tiako, P. (2009). *Designing Software-intensive Systems: Methods and Principles*. IT Pro. Information Science Reference.

[Vanderperren e Dehaene 2005] Vanderperren, Y. e Dehaene, W. (2005). SysML and Systems Engineering Applied to UML-Based SoC Design.

[Viehl et al. 2006] Viehl, A., Schönwald, T., Bringmann, O., e Rosenstiel, W. (2006). Formal Performance Analysis and Simulation of UML/SysML Models for ESL Design. In

Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings, DATE '06, pp. 242–247. European Design and Automation Association.

[Willard 2007] Willard, B. (2007). {UML} for Systems Engineering. *Computer Standards & Interfaces*, 29(1):69 – 81. {ADC} Modelling and Testing.

[Zhao e Zhang 2007] Zhao, C. e Zhang, K. (2007). Transformational Approaches to Model Driven Architecture - A Review. In *Proceedings of the 31st IEEE Software Engineering Workshop*, SEW '07, pp. 67–74. IEEE Computer Society.