**UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ**

**DEPARTAMENTO ACADÊMICO DE INFORMÁTICA**

**TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS**

**YASMIN VOLMER TULLIO**

# COMPARAÇÃO ENTRE AS PRINCIPAIS LINGUAGENS DE TRANSFORMAÇÃO DE MODELOS

**TRABALHO DE CONCLUSÃO DE CURSO**

**PONTA GROSSA**

**2018**

**YASMIN VOLMER TULLIO**

# COMPARAÇÃO ENTRE AS PRINCIPAIS LINGUAGENS DE TRANSFORMAÇÃO DE MODELOS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. MSc. Vinícius Camargo Andrade

**PONTA GROSSA**

**2018**

Ministério da Educação
**Universidade Tecnológica Federal do Paraná**
Campus Ponta Grossa

Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Tecnologia em Análise e Desenvolvimento de Sistema

# TERMO DE APROVAÇÃO

COMPARAÇÃO ENTRE AS PRINCIPAIS LINGUAGENS DE TRANSFORMAÇÃO DE MODELOS

por

YASMIN VOLMER TULLIO

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 11 de junho de 2018 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas. A candidata foi arguida pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. MSc. Vinícius Camargo Andrade
Prof. Orientador

_____
Prof. Dr. Richard Duarte Ribeiro
Membro titular

_____
Prof. MSc. Rafael dos Passos Canteri
Membro titular

_____
Profª Drª Helyane Bronoski Borges
Responsável pelos Trabalhos
de Conclusão de Curso

_____
Prof. Dr. André Pinz Borges
Coordenador do Curso
UTFPR - Campus Ponta Grossa

## AGRADECIMENTOS

Agradeço aos meus pais, Roselaine e Moacir, e meus irmãos, Yahra e Yahgo, por todo o apoio que me deram e por não medirem esforços para que eu pudesse concluir esse trabalho, sem vocês eu jamais conseguiria. Ao Igor, por ter entendido os momentos de ausência, pelo incentivo e companheirismo em todo o tempo. Ao meu orientador, professor Vinicius, por ter aceitado me orientar neste trabalho e pela total disponibilidade para me ajudar. A todos os meus amigos que torceram por mim e me ajudaram a passar por essa fase de forma mais leve.

Por fim, agradeço a Deus por ter me dado saúde, força, sabedoria e vontade para que este trabalho fosse realizado.

## RESUMO

TULLIO, Yasmin. **Comparação entre as Principais Linguagens de Transformação de Modelos.** 2018. 53 f. Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

O Desenvolvimento Orientado a Modelos (MDD - do inglês, *Model Driven Development*) é um método de desenvolvimento de *software* que tem a modelagem como o principal artefato para a criação do sistema. Além disso, é possível, por meio de linguagem de transformação, realizar transformações entre modelos. No entanto, cada linguagem possui características particulares, tendo vantagens e desvantagens uma em relação as outras. O objetivo deste trabalho é comparar as principais linguagens de transformação de modelos, apontando as principais características de cada uma. Para isso, definiram-se as linguagens *Triple Graph Grammars* (TGG), *Query/View/Transformation* (QVT) e *Atlas Transformation Language* (ATL), para a comparação, e os modelos de diagrama de atividades da linguagem unificada de modelagem (UML, do inglês, *Unified Modeling Language*) e rede de Petri, para a transformação. Como resultado, foram apontadas as vantagens e desvantagens de cada linguagem considerando características, além de apresentar trabalhos relacionados envolvendo transformação de modelos de diagrama de atividades – UML para rede de Petri.



**Palavras-chave:** Desenvolvimento Orientado a Modelos. Transformação de Modelos. Comparação de Modelos. Diagrama de Atividades. Rede de Petri.

## ABSTRACT

TULLIO, Yasmin. **Comparison of the Main Languages of Model Transformation**. 2018. 53 p. Work of Conclusion Course (Graduation in Technology in Systems Analysis and Development) - Federal Technology University - Paraná. Ponta Grossa, 2018.

Model Driven Development (MDD) is a software development method that has the modeling as the main artifact for the creation of the system. In addition, it is possible, by means of transformation language, to make transformations between models. However, each language has particular characteristics, having advantages and disadvantages in relation to others. The objective of this work is to compare the main languages of transformation of models, pointing the main characteristics of each one. For this, the languages Triple Graph Grammars (TGG), Query/View/Transformation (QVT) and Atlas Transformation Language (ATL), were defined for a comparison, and the activity diagram model of the Unified Modeling Language (UML) and Petri net, to transformation. As a result, the advantages and disadvantages of each language were pointed out, besides the related works involving transformation of activity diagram models - UML to Petri net.



**Keywords:** Model-Driven Development. Model Transformation. Model Comparison. Activity Diagram. Petri Net.

## LISTA DE FIGURAS

Figura 1 – A Taxonomia Da Estrutura E O Diagrama De Comportamento..... ....14
Figura 2 – Diagrama De Atividades.....16
Figura 3 – Grafo E Seus Elementos Básicos. .....17
Figura 4 – Rede De Petri Marcada.....18
Figura 5 – Rede De Petri Com Conflito .....18
Figura 6 – Rede De Petri Com Paralelismo.....19
Figura 7 – Arquitetura De Modelagem .....21
Figura 8 – Plano De Construção Para O Trem De Brinquedo.....23
Figura 9 – Metamodelo Do Projeto.....23
Figura 10 – Mapeamento Informal Dos Elementos Do Projeto Para Redes De.....24
Figura 11 – Diagrama Objeto Do Componente *Track* E Seu Correspondente.....24
Figura 12 – Regra TGG Para O Componente *Track*.....25
Figura 13 – Regra TGG Para A Divisão De Um *Switch* .....26
Figura 14 – Regra TGG Para A Fusão De Um *Switch* .....26
Figura 15 – Regra TGG Para Um *Connection*.....27
Figura 16 – Regra TGG Para Um *Train* .....27
Figura 17 – Metamodelo Para Os Objetos De Correspondência .....28
Figura 18 – Aplicação Das Regras TGG .....28
Figura 19 – Modelos Correspondentes Na Sintaxe Gráfica .....29
Figura 20 – Forward Transformation .....30
Figura 21 – *Forward Transformation* Depois De Duas Aplicações Da Regra Para...30
Figura 22 – *Forward Transformation* Depois Da Aplicação Da Regra Para O Com .31
Figura 23 – *Model Integration*.....32
Figura 24 – Relacionamentos Entre Metamodelos QVT .....33
Figura 25 – Visão Global Da Transformação De Modelo .....36

## LISTA DE QUADROS

Quadro 1 – Definição Da Sintaxe Da *Header Section* ..... 36
Quadro 2 – Exemplo De *Header Section* ..... 37
Quadro 3 – Definição Da Sintaxe Da *Import Section* ..... 37
Quadro 4 – Exemplo De *Import Section* ..... 38
Quadro 5 – Definição Da Sintaxe De Um *Helper* ..... 38
Quadro 6 – Exemplo De Um *Helper* ..... 39
Quadro 7 – Definição Da Sintaxe De Uma *Matched Rule* ..... 40
Quadro 8 – Exemplo De Uma *Matched Rule* ..... 41
Quadro 9 – Definição Da Sintaxe De Uma *Called Rule* ..... 42
Quadro 10 – Exemplo De Uma *Called Rule* ..... 42
Quadro 11 – Diferenças Filosóficas ..... 43
Quadro 12 – Diferença Semântica ..... 44
Quadro 13 – Diferenças Conceituais ..... 44
Quadro 14 – Cenários Suportados ..... 45
Quadro 15 – Características ..... 45

## LISTA DE ACRÔNIMOS

| | |
|---|---|
| ATL | *Atlas Transformation Language* |
| EMF | *Eclipse Modeling Framework* |
| MDA | *Model-Driven Architecture* |
| MDD | *Model-Driven Development* |
| MOF | *Meta-Object Facility* |
| OCL | *Object Constraint Language* |
| OMG | *Object Management Group* |
| QVT | *Query, View, Transformation* |
| RdP | Rede de Petri |
| TGG | *Triple Graph Grammar* |
| UML | *Unified Modeling Language* |
| XSLT | *Extensible Stylesheet Language Transformations* |

**SUMÁRIO**

**1 INTRODUÇÃO ..... 11**
1.1 PROBLEMA ..... 12
1.2 OBJETIVOS ..... 12
1.2.1 Objetivo Geral ..... 12
1.2.2 Objetivos Específicos ..... 12
1.3 ORGANIZAÇÃO DO TRABALHO ..... 13
**2. LINGUAGEM UNIFICADA DE MODELAGEM ..... 14**
2.1 DIAGRAMA DE ATIVIDADES ..... 15
**3 REDE DE PETRI ..... 17**
**4 DESENVOLVIMENTO ORIENTADO A MODELOS ..... 20**
**5 LINGUAGENS DE TRANSFORMAÇÃO ..... 22**
5.1 TRIPLE GRAPH GRAMMARS (TGG) ..... 22
5.1.1 Cenários de Aplicação ..... 29
5.1.1.1 *Model Transformation* ..... 29
5.1.1.2 *Model Integration* ..... 31
5.1.1.3 *Model Synchronization* ..... 32
5.2 QUERY, VIEW, TRANSFORMATION (QVT) ..... 33
5.3 ATLAS LANGUAGE TRANSFORMATION (ATL) ..... 35
5.3.1 Módulos ATL ..... 36
5.3.1.1 *Header Section* ..... 36
5.3.1.2 *Import Section* ..... 37
5.3.1.3 *Helpers* ..... 38
5.3.1.4 *Rules* ..... 39
**6 CONTRIBUIÇÃO ..... 43**
6.1 COMPARAÇÃO ENTRE AS LINGUAGENS ..... 43
6.2 TRABALHOS RELACIONADOS ..... 46
**7 CONCLUSÃO ..... 47**
**REFERÊNCIAS ..... 49**

# 1 INTRODUÇÃO

O desenvolvimento de um *software* complexo exige uma representação abstrata do produto em questão, tanto para que o levantamento de requisitos seja feito de forma correta, quanto para que a equipe de desenvolvimento utilize esse modelo como base para realizar o trabalho. A modelagem é vista como uma ferramenta secundária, apesar da sua relevância no processo de criação de um sistema (SELIC. 2003; ATKINSON, KÜHNE. 2003).

Como alternativa de abordagens de desenvolvimento, tem-se o Desenvolvimento Orientado a Modelos (MDD, do inglês *Model Driven Development*), que tem a modelagem como principal artefato para a construção de um projeto de *software*. Nesse cenário, a transformação entre modelos se torna um instrumento eficaz, já que permite um refinamento dos modelos já compreendidos (SENDALL, KOZACZYNSKI. 2003).

Há na literatura várias linguagens de modelagem de sistemas, como por exemplo, a Linguagem Unificada de Modelagem (UML, do inglês *Unified Modeling Language*), considerada padrão para modelagem orientada a objetos e utilizada pela indústria de *software* (BOOCH, et al. 2000). Por outro lado, não é considerada uma linguagem formal de modelagem, ou seja, não possibilita realizar análises e verificações formais nos modelos propostos.

Há outras linguagens formais de modelagens, destas pode-se citar a rede de Petri (MURATA, 1989), que possibilita representações mais simplificadas comparadas aos diagramas UML, além de análises e verificações formais no modelo gerado, por possuir embasamento matemático.

Para realizar a transformação entre os modelos, utilizam-se linguagens de transformação, que seguindo regras previamente estabelecidas, são capazes de traduzir um modelo a outro. As principais linguagens encontradas na literatura são: *Atlas Transformation Language* (ATL) (ATLAS group, LINA & INRIA, 2005a), *Query/View/Trasnformation* (QVT) (OMG, 2011) e *Triple Graph Grammar* (TGG) (KINDLER E WAGNER, 2007).

Este trabalho apresenta um estudo sobre linguagens de transformações de modelos, abordando as principais características, vantagens e desvantagens de cada uma. Para a realização, limitou-se na transformação de diagrama de atividades UML para rede de Petri. Esta definição deu-se pela proximidade das características entre

os dois modelos e também pelo fato da pertinência da transformação conforme citado anteriormente.

## 1.1 PROBLEMA

A transformação de modelos se faz necessária quando, dado um modelo, se quer outro modelo equivalente. Para que estas transformações sejam realizadas de forma padronizada e corretamente, deve-se definir regras e as seguir rigorosamente. Neste contexto estão as linguagens de transformações de modelos.

No entanto, há inúmeras linguagens de transformações de modelos na literatura, como por exemplo, *Triple Graph Grammars* (TGG), *Query/View/Transformation* (QVT) e *Atlas Transformation Language* (ATL). Sendo possível aplicá-las na resolução da mesma transformação.

Uma vez que o engenheiro de *software* tem a possibilidade de optar entre diferentes linguagens de transformações na resolução de uma conversão de modelos, tem-se a dúvida de qual linguagem utilizar na resolução de um determinado problema, considerando que cada linguagem tem vantagens e desvantagens, dependendo do contexto.

## 1.2 OBJETIVOS

Os objetivos gerais e específicos serão descritos a seguir.

### 1.2.1 Objetivo Geral

Comparar as principais Linguagens de Transformação de Modelos da atualidade.

### 1.2.2 Objetivos Específicos

- Estudar as principais Linguagens de Transformação de Modelos;
- Apontar as vantagens e desvantagens de cada Linguagem;
- Analisar trabalhos relacionados e aplicações das linguagens citadas.

## 1.3 ORGANIZAÇÃO DO TRABALHO

Esse trabalho está dividido em sete capítulos. O Capítulo 2 apresenta uma revisão a respeito da Linguagem Unificada de Modelagem. O Capítulo 3 apresenta um estudo sobre Rede de Petri. O Capítulo 4 descreve o Desenvolvimento Orientado a Modelos. O Capítulo 5 apresenta as Linguagens de Transformação de Modelos. O Capítulo 6 apresenta a comparação entre as linguagens e os trabalhos relacionados. E, por fim, o Capítulo 7 discorre sobre a conclusão do trabalho e faz propostas para trabalhos futuros.

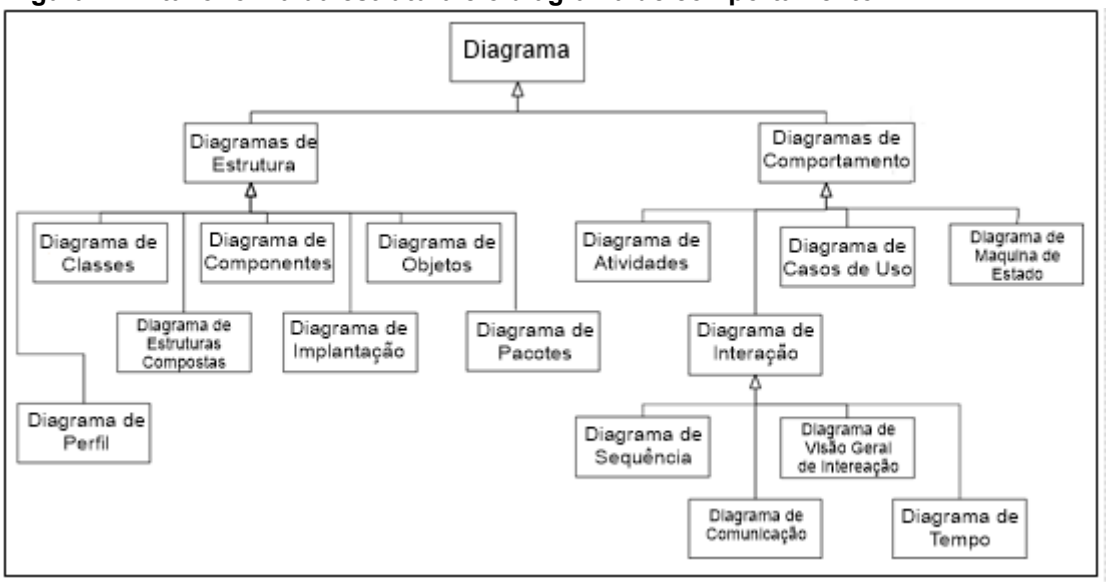
## 2 LINGUAGEM UNIFICADA DE MODELAGEM - UML

A modelagem é um processo que ocorre durante o período do projeto de desenvolvimento de um sistema e permite a representação do produto a ser desenvolvido com o intuito de facilitar o entendimento e a descrição do mesmo, especialmente nos casos em que o sistema exige maior complexidade (BOOCH et al, 2000)

Muitos métodos para a criação de modelos foram utilizados, tendo soluções e aspectos diferentes, até que, em 1994, James Rumbaugh e Grady Booch resolveram unificar seus métodos e criar uma linguagem padrão de modelagem. Assim iniciou-se o processo de criação da UML (em inglês, *Unified Modeling Language*), que permite que os desenvolvedores representem e descrevam seus sistemas em forma de diagramas em diferentes níveis de abstração (BOOCH et al., 2000).

Segundo *Object Management Group OMG* (2011), os diagramas apresentados pela UML são divididos em dois grupos: diagramas de estrutura e de comportamento, como mostra a Figura 1.

**Figura 1 - A taxonomia da estrutura e o diagrama de comportamento**




**Fonte: Adaptada de Object Management Group OMG (2011).**

Os diagramas de estrutura permitem a representação de aspectos que envolvem a estrutura do sistema, enquanto os diagramas de comportamento representam tipos gerais de comportamento e enquadram outros quatro tipos de

diagrama, os diagramas de interação.

São exemplos de diagramas de estrutura: Diagrama de classes, de Componentes, de Objetos, de Perfil, de Estruturas Compostas, de Implantação, e de Pacotes. São exemplos de diagramas de comportamento: Diagrama de Atividades, de Casos de Uso, de Interação, de Máquina de Estados, de Sequência, de Comunicação, de Visão Geral de Interação e de Tempo (OBJECT MANAGEMENT GROUP OMG, 2011).

Dentre os diagramas citados, será abordado em mais detalhes o Diagrama de Atividades, por ser encontrado em vários trabalhos envolvendo transformação de modelos, tais como o de Staines (2010), Lachtermacher et al. (2008), Jamal e Zafar (2016) e Staines (2011).

## 2.1 DIAGRAMA DE ATIVIDADES

O Diagrama de Atividades é classificado como um diagrama de comportamento e representa um fluxo existente entre atividades sequenciais ou concorrentes presentes no sistema. Esse tipo de representação dá ênfase especialmente a aspectos dinâmicos do programa e preocupa-se em descrever o passo a passo a ser percorrido para que um método seja concluído (BOOCH et al., 2000).

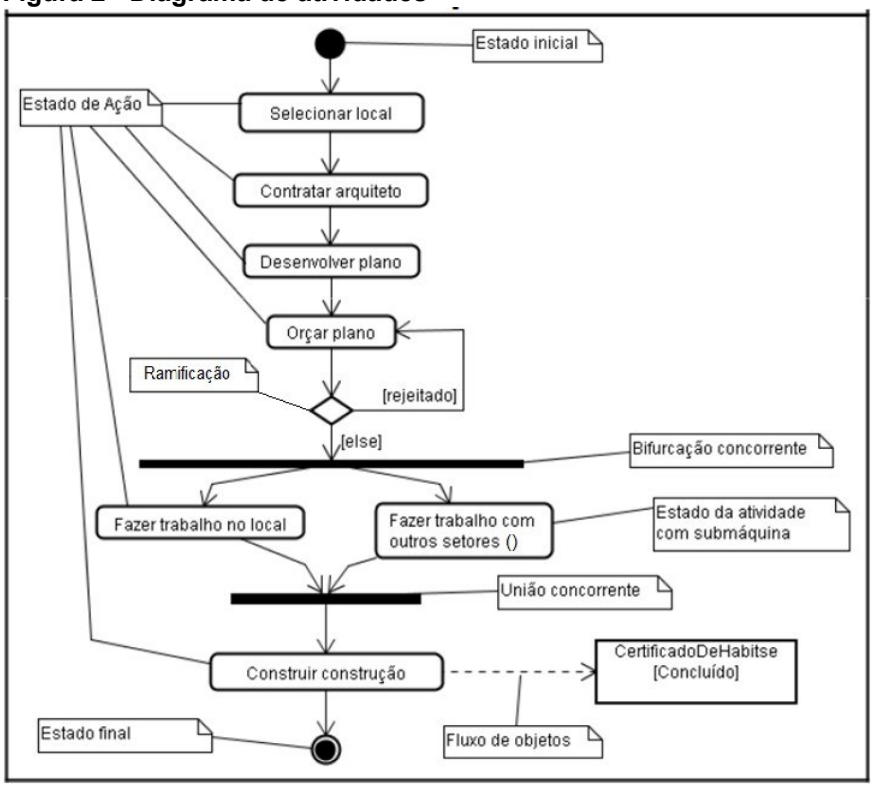
O diagrama de atividades é formado pelos seguintes elementos, segundo Booch et al. (2000):

- Estados de atividades – são execuções que não são atômicas, e ao observá-los, pode-se encontrar um novo diagrama de atividades.
- Estados de ação – são execuções atômicas, ou seja, não podem ser interrompidas.
- Estado Inicial – inicia o fluxo de atividades, dando abertura ao diagrama.
- Estado Final – encerra o fluxo.
- Transições – ao completar uma ação ou atividade, o fluxo de controle passa ao estado seguinte de ação ou atividade. Este fluxo é especificado utilizando transições.
- Ramificação – expressões booleanas permitem que caminhos alternativos existam, esses caminhos são apresentados por ramificações.

- Bifurcação e União – fluxos concorrentes são representados por uma barra de sincronização, especificando uma bifurcação ou união.
- Raias de Natação – separa o diagrama em grupos, que representam uma entidade do mundo real.
- Fluxo de objetos – um estado de ação pode resultar na criação, modificação ou destruição de um objeto, e a participação desse, resulta em um fluxo de objeto.

A Figura 2 ilustra um diagrama de atividades, neste exemplo podem ser observados alguns dos elementos citados por Booch et al. (2000), como o estado inicial e final, representados pelos círculos preenchidos; estados de ação e atividade, identificados pelos retângulos Selecionar local, Contratar arquiteto, Desenvolver plano, Orçar plano, Fazer trabalho no local, Fazer trabalho com outros setores e Construir construção; bifurcação e união, simbolizados por linhas horizontais; transições, ramificação e fluxo de objetos, representados pelas setas.

**Figura 2 - Diagrama de atividades**




**Fonte: Booch et al. (2000)**

# 3 REDE DE PETRI

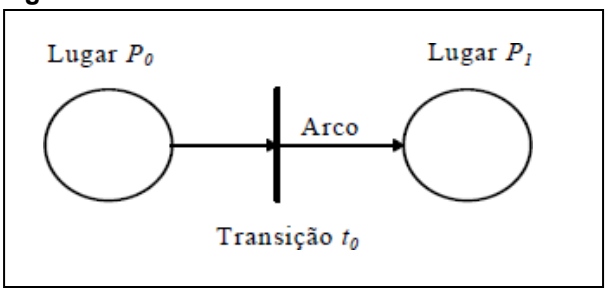
Segundo Murata (1989) Rede de Petri é uma ferramenta gráfica e matemática proposta por Carl Adam Petri (1962), utilizada na modelagem de sistemas, que permite obter informações sobre o comportamento do sistema em questão.

Uma Rede de Petri é uma quíntupla, *RdP=<P,T,Pre,Pos,M>*, onde P é um conjunto de posições, *T* é um conjunto de transições, *Pre* é a aplicação de entrada, *Pos* é a aplicação de saída e *M* é a marcação inicial. As notações matriciais das matrizes de incidência *Pre(P,T)* e *Pos(P,T)* são definidas como dimensão *n x m*, onde *n* é a quantidade de lugares e *m* é a quantidade de transições (CARDOSO E VALETTE, 1997).

Segundo Francês (2003), uma Rede de Petri pode ser representada graficamente, contendo assim dois tipos de nós: lugares e transições. Para unir um lugar a uma transição ou uma transição a um lugar, existem os arcos. Essas uniões, respectivamente, são previstas se, e somente se, $Pre(P,T) \neq 0$ e $Pos(P,T) \neq 0$.

A Figura 3 ilustra uma Rede de Petri com elementos básicos. Os lugares são identificados por: *Lugar* $P_0$ e *Lugar* $P_1$; a Transição é identificada por: *Transição* $t_0$; e os arcos são identificados pela união entre *Lugar* $P_0$ a *Transição* $t_0$ e *Transição* $t_0$ a *Lugar* $P_1$.

**Figura 3 - Grafo e seus elementos básicos**

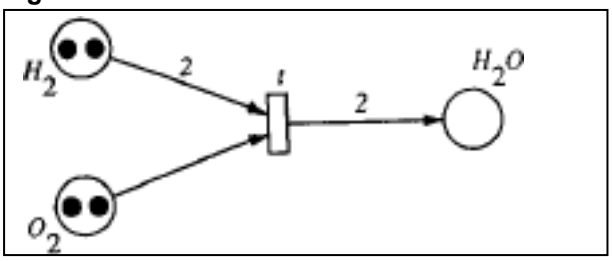



**Fonte: Francês (2003).**

Além destes elementos, tem-se também nas Redes de Petri as marcações. Estas podem ser atribuídas aos lugares para representar o estado da Rede em algum momento, sendo cada marca ilustrada por um círculo preto dentro de um nó lugar. Uma Rede de Petri marcada é representada por *M(p)* (FRANCÊS, 2003).

A Figura 4 ilustra uma Rede de Petri com 3 lugares, dois possuem duas marcações cada. São eles $H_2$ e $O_2$. O outro lugar, representado na figura por $H_2O$ não possui marcação.

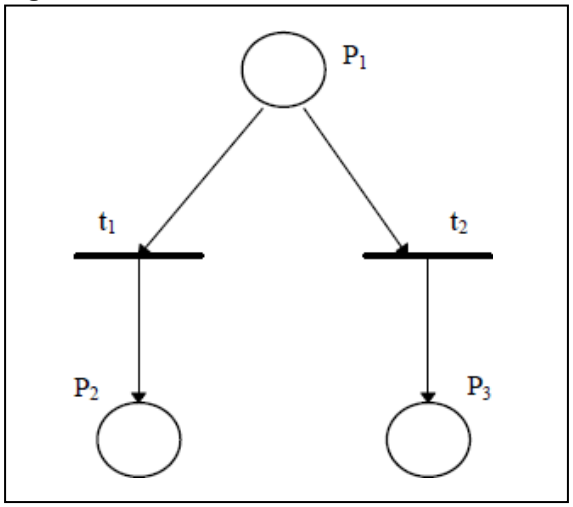
**Figura 4 - Rede de Petri Marcada**




**Fonte: Murata (1989)**

Quando dois ou mais eventos podem ocorrer, porém não ao mesmo instante, dá-se o conflito (FRANCÊS, 2003). A Figura 5 ilustra uma estrutura que um lugar $P_1$ possui duas transições de saída, $t_1$ e $t_2$. Neste caso caracteriza-se um conflito, também chamado de decisão ou escolha, pois a marcação, ao chegar no lugar $P_1$, seguirá o fluxo até a transição $t_1$ ou $t_2$, nunca a ambas.

**Figura 5 - Rede de Petri com Conflito**



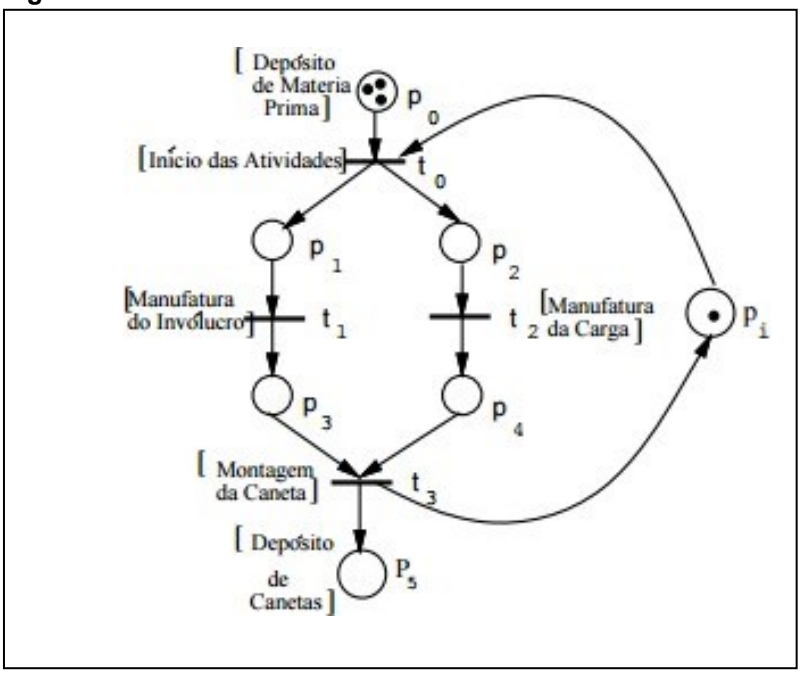

**Fonte: Francês (2003)**

No paralelismo, ambos os eventos podem ocorrer como concorrentes (MURATA, 1989). A Figura 6, na página 19, ilustra marcações inicialmente no lugar $P_0$. Ao seguir pelo arco tem-se a transição $t_0$ que possui dois fluxos de saída, o primeiro apontando para $P_1$ e o segundo para $P_2$. Neste caso, uma marcação irá à $P_1$ e outra à $P_2$, ou seja, os estados das atividades executarão paralelamente, sem que uma interfira na outra.

Conflito e paralelismo podem ocorrer na estrutura do sistema sem que sejam

efetivados. Para que um conflito ou paralelismo seja efetivo, é necessário que o número de marcas seja maior que o peso do arco que liga os lugares às respectivas transições. Ou seja, *M* ≥ *Pre(x, t1)* e *M* ≥ *Pre(x, t2)* (CARDOSO E VALETTE, 1997).

**Figura 6 - Rede de Petri com Paralelismo**




**Fonte: Maciel et at. (1996)**

# 4 DESENVOLVIMENTO ORIENTADO A MODELOS

Como as outras áreas da engenharia, a Engenharia de *Software* também necessita que seja realizada uma análise e abstração do sistema a ser desenvolvido antes da criação do produto. Essa abstração é dada a partir da modelagem, que permite uma melhor comunicação entre os desenvolvedores, clientes e todos os envolvidos na produção do *software*, por se tratar de uma representação de alto nível do produto. No decorrer do processo, são realizadas mudanças, e os modelos acabam tornando-se desatualizados (EISHIMA, 2014).

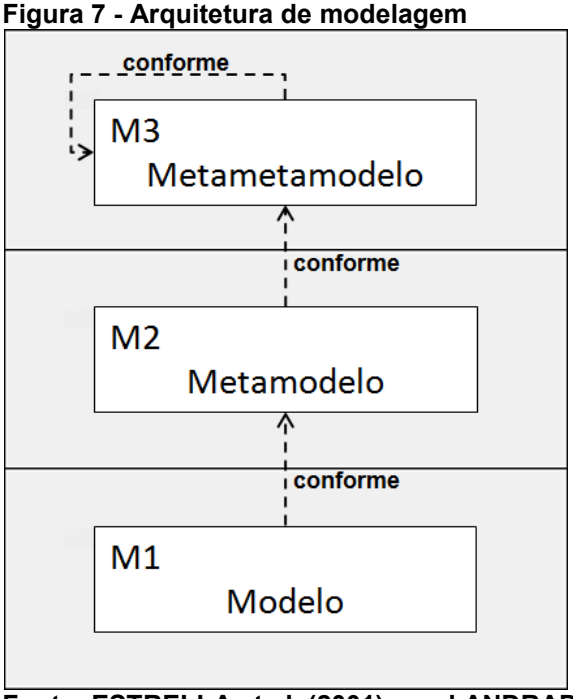
O Desenvolvimento Orientado a Modelo (MDD, em inglês *Model-Driven Development*) torna a modelagem não apenas um artefato secundário, mas o foco principal do desenvolvimento do *software*. Tendo a geração automática de códigos como o maior objetivo do MDD, os modelos deixam de ter o valor limitado que têm quando são utilizados como documentação e passam a ser parte do *software* (SELIC, 2003). Segundo Lucrédio (2009), as vantagens do MDD se resumem em evitar tarefas repetitivas que são necessárias para o desenvolvimento do *software*, automatizando as transformações dos modelos para códigos.

Para sintetizar o potencial da tarefa da modelagem, surge a metamodelagem, que descreve a estrutura de um modelo e define os construtores de uma linguagem de modelagem e seus relacionamentos (NETO, 2012).

Essa abordagem baseada em metamodelagem é especificada pelo *Object Management Group* (OMG), o qual define três elementos: modelos, metamodelos e metametamodelos. Modelo é uma abstração que demonstra aspectos do sistema, metamodelo é uma abstração do modelo, e o metametamodelo é uma abstração do metamodelo. A Figura 7, na página 21, ilustra a arquitetura de modelagem, apresentando o modelo, que deve estar em conformidade com seu metamodelo, que por sua vez, deve ser definido em conformidade com seu metametamodelo. Este último, deve estar em conformidade com ele mesmo (OMG, 2002).

O OMG define uma linguagem de metamodelagem denominada *Meta-Object Facility* (MOF), que permite a definição de classes com atributos e relacionamentos, além de uma interface para comunicação com dados e metadados (LUCRÉDIO, 2009).

**Figura 7 - Arquitetura de modelagem**




**Fonte: ESTRELLA et al. (2001) apud ANDRADE (2013)**

# 5 LINGUAGENS DE TRANSFORMAÇÕES DE MODELOS

A linguagem de transformação é responsável por traduzir, seguindo regras previamente estabelecidas, um modelo de origem em um modelo alvo. Para que isso seja possível, ambos os modelos devem estar em conformidade com seus respectivos metamodelos, que por sua vez, devem estar em conformidade com o meta-metamodelo considerado.

Para a realização deste estudo comparativo, selecionou-se na literatura as três principais linguagens de transformação de modelos. São elas: *Triple Graph Grammars* (TGG), *Query / View / Transformation (QVT)* e *Atlas Transformation Language* (ATL).

## 5.1 TRIPLE GRAPH GRAMMARS (TGG)

Segundo Kindler e Wagner (2007), *Triple Graph Grammars* (TGG) é uma técnica para definição de correspondência entre dois tipos de modelos diferentes em uma forma declarativa.

Devido a sua capacidade de definir a relação entre os dois modelos em questão, também é possível transformá-los de forma bidirecional, ou seja, pode-se transformar o modelo A em B e vice-versa.

Além disso, o mesmo pode ser utilizado para sincronizar e manter a correspondência entre os modelos, mesmo se ambos são alterados de forma independente, ou seja, TGG trabalha de forma incremental.

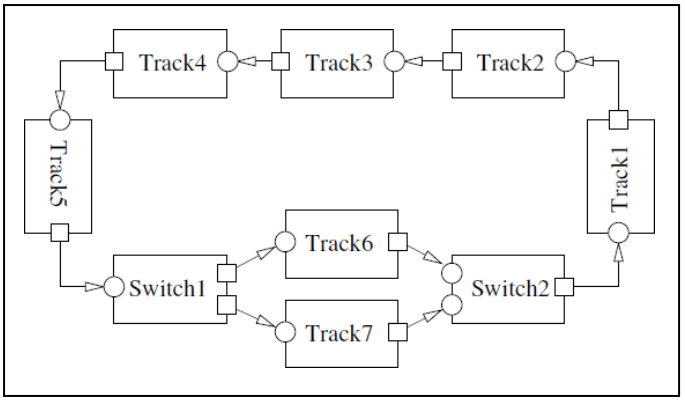
Para ilustrar o conceito apresentado, Kindler et. al. (2006) utiliza o exemplo de um trem de brinquedo em que são utilizados componentes padrão de Redes de Petri que definem o comportamento dinâmico do trem.

O plano de construção do trem de brinquedo consiste em dois componentes apenas: *Tracks* e *Switches*. Estes componentes possuem portas que podem ser conectadas mecanicamente. O componente *Track* possui apenas uma porta de entrada e uma de saída, já o *Switch* possui uma porta de entrada e duas portas de saída ou duas portas de entrada e uma de saída.

A Figura 8, na página 23, ilustra a representação gráfica do plano de construção, o qual foi chamado de Projeto. Os componentes são identificados por

caixas, as portas de entradas são representadas por círculos e as de saída são representadas por quadrados. A conexão entre uma porta de saída e uma porta de entrada é representada por uma seta, chamada de *Connection*.

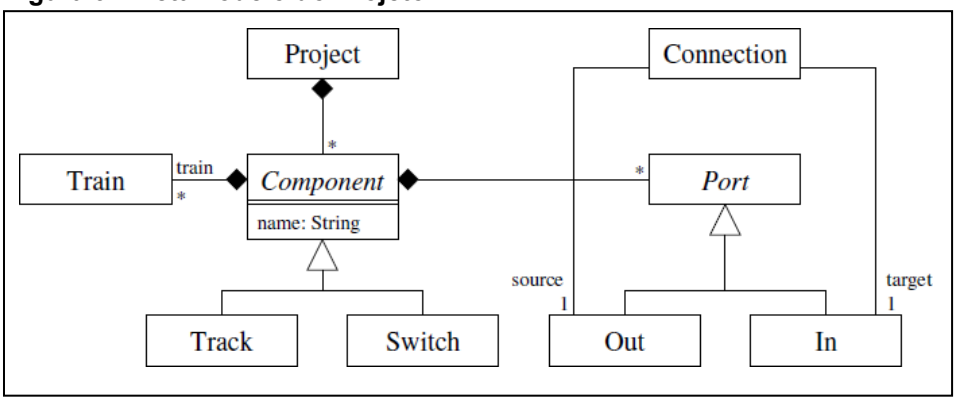
**Figura 8 - Plano de Construção para o trem de brinquedo**




**Fonte: Kindler e Wagner (2007)**

Para este projeto, o metamodelo é ilustrado na Figura 9. O Projeto é composto por vários *Components*, que por sua vez são compostos por um *Train* e n *Ports,* que podem ser de entrada ou saída – *Out* ou *In.*

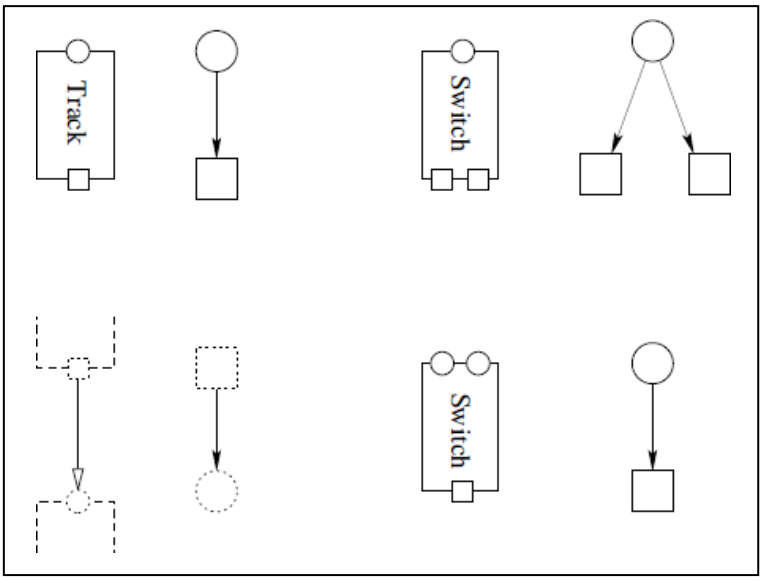
**Figura 9 - Metamodelo do Projeto**




**Fonte: Kindler e Wagner (2007)**

A seguir é definido um plano de construção do projeto e seus diferentes elementos são mapeados em outro modelo, neste caso, uma Rede de Petri. A Figura 10, na página 24, ilustra o mapeamento de forma informal em sintaxe gráfica, apenas como suporte para melhor entendimento do exemplo, já que a transformação em si ocorre em um nível acima, no metamodelo.

**Figura 10 - Mapeamento informal dos elementos do Projeto para Redes de Petri**

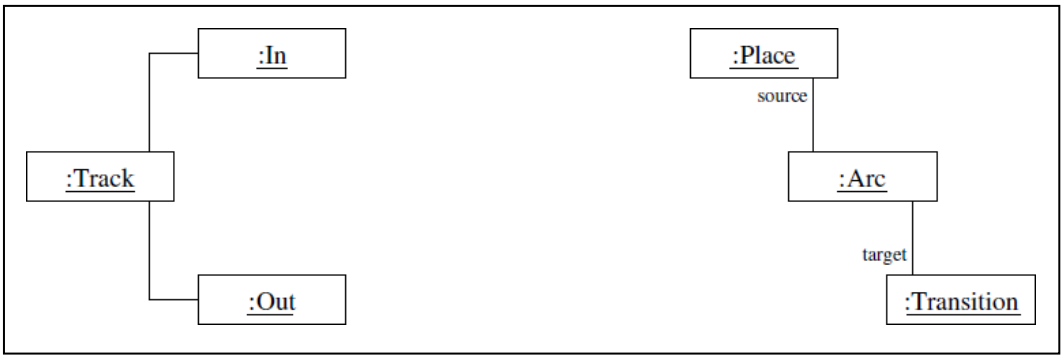



**Fonte: Kindler e Wagner (2007)**

Para realizar este mapeamento formalmente utilizam-se as regras TGG. Estas regras referem-se aos elementos de ambos os modelos na sintaxe abstrata, ou seja, no Diagrama Objeto.

Com base na Figura 10, em que são definidos os modelos gráficos de ambos os modelos, é realizada então a transformação de cada elemento para seu correspondente Diagrama Objeto. A Figura 11 ilustra o Diagrama Objeto especificamente do componente *Track* e seu correspondente na Rede de Petri.

**Figura 11 - Diagrama Objeto do componente *Track* e seu correspondente na Rede de Petri**

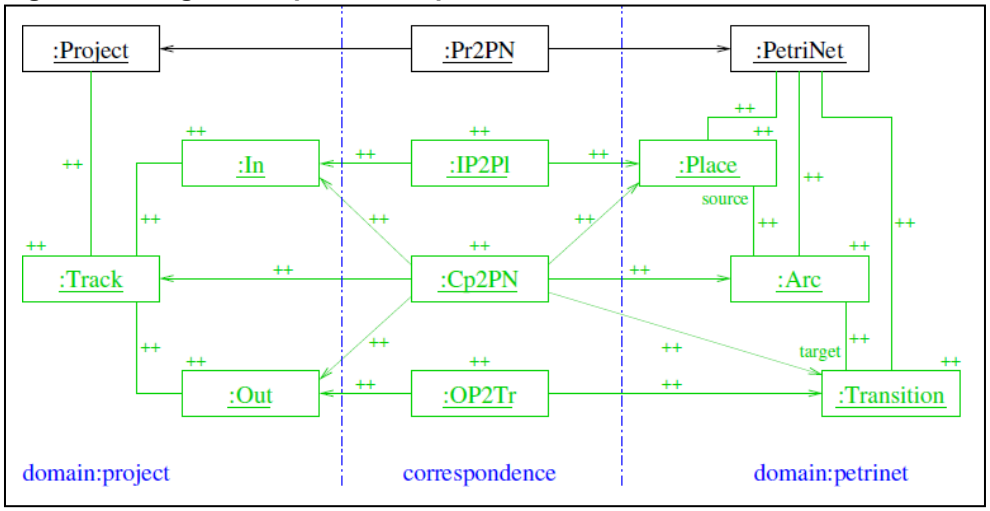



**Fonte: Kindler e Wagner (2007)**

Após a definição dos Diagramas Objetos, os mesmos são separados em 3 (três) colunas: a coluna da esquerda contém o domínio do projeto, a coluna da direita contém o domínio da Rede de Petri e, no centro, a região que define as correspondências dos diferentes modelos.

Esta regra do TGG é apresentada na Figura 12, em que as peças na cor preta representam um Projeto que corresponde a uma Rede Petri. As peças marcadas com cor verde representam os componentes correspondentes a cada diagrama e também as regras de correspondência entre os dois modelos.

**Figura 12 - Regra TGG para o componente *Track*.**



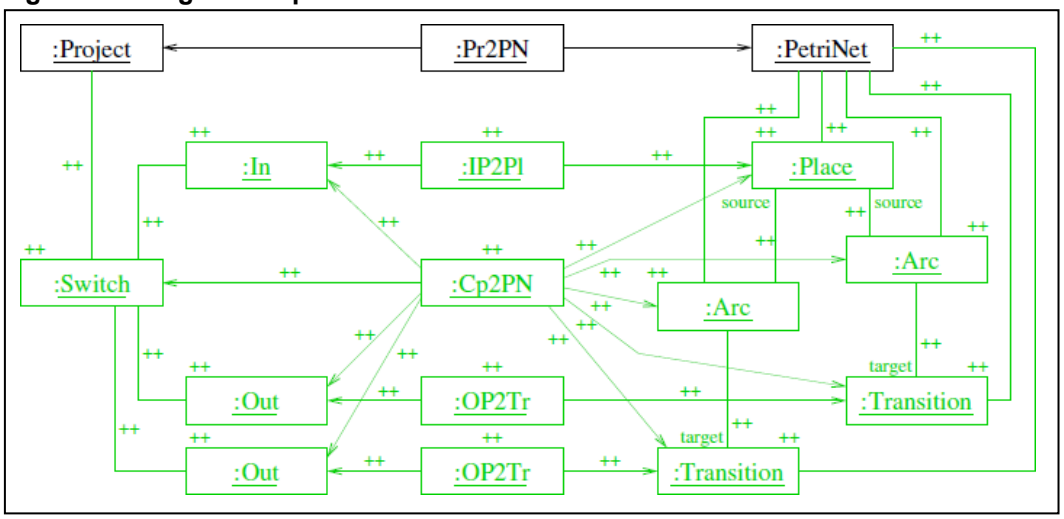

**Fonte: Kindler e Wagner (2007)**

Na coluna *Correspondence* da Figura 12, há 3 caixas, “IP2PI”, “Cp2PN” e “OP2Tr”, as quais correspondem respectivamente a transformação da caixa *“In”* para a caixa *“Place”,* da caixa *“Track”* para a caixa *“Arc”* e da caixa *“Out”* para a caixa *“Transition”*.

Após definidas as correspondências para o componente *Track*, é necessário também determinar para os demais componentes. A definição do componente *Switch* com uma entrada e duas saídas é ilustrada pela Figura 13, na página 26.

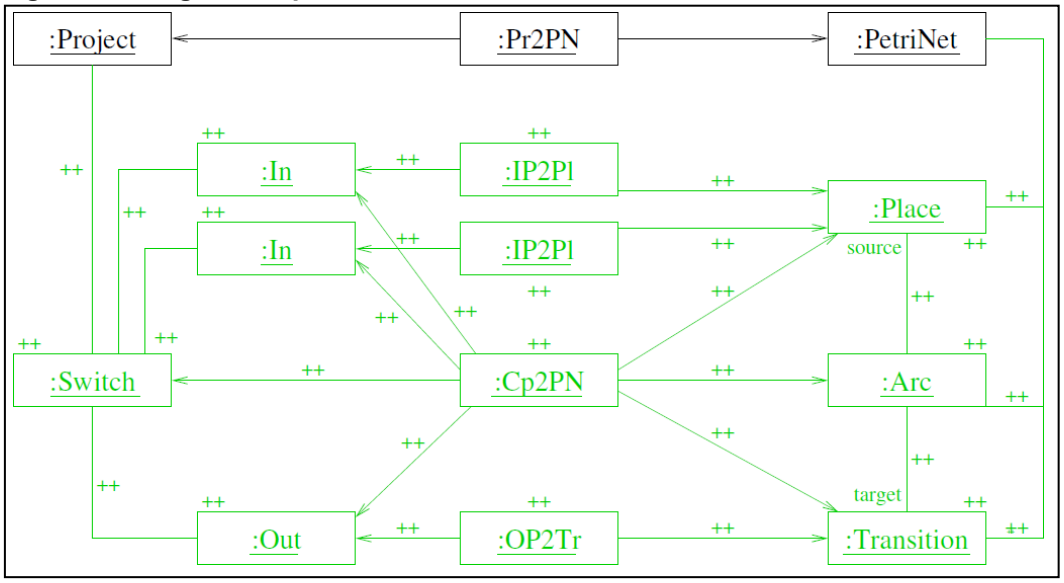
A Figura 14, na página 26, ilustra o mesmo processo, porém para o *Switch* de duas entradas e apenas uma saída.

**Figura 13 - Regra TGG para a divisão de um *Switch***




**Fonte: Kindler e Wagner (2007)**

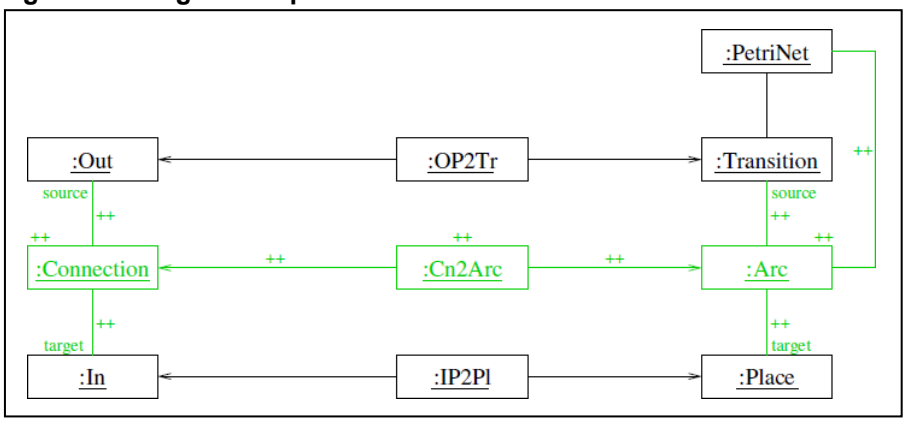
**Figura 14 - Regra TGG para a fusão de um *Switch***




**Fonte: Kindler e Wagner (2007)**

Para o componente *Connection*, o qual é responsável pela ligação de um componente a outro, equivale o componente *Arc* da Rede de Petri. Esta relação é apresentada pela Figura 15, na página 27.

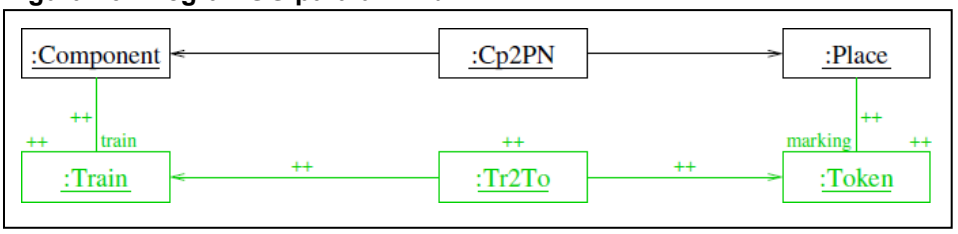
**Figura 15 - Regra TGG para um *Connection***




**Fonte: Kindler e Wagner (2007)**

O *Train* não deixa de ser um componente do Diagrama. Ele está representado no metamodelo do Projeto, ilustrado na Figura 9, e este corresponde a um *Token* na Rede de Petri. A regra de transformação está ilustrada na Figura 16.

**Figura 16 - Regra TGG para um *Train***



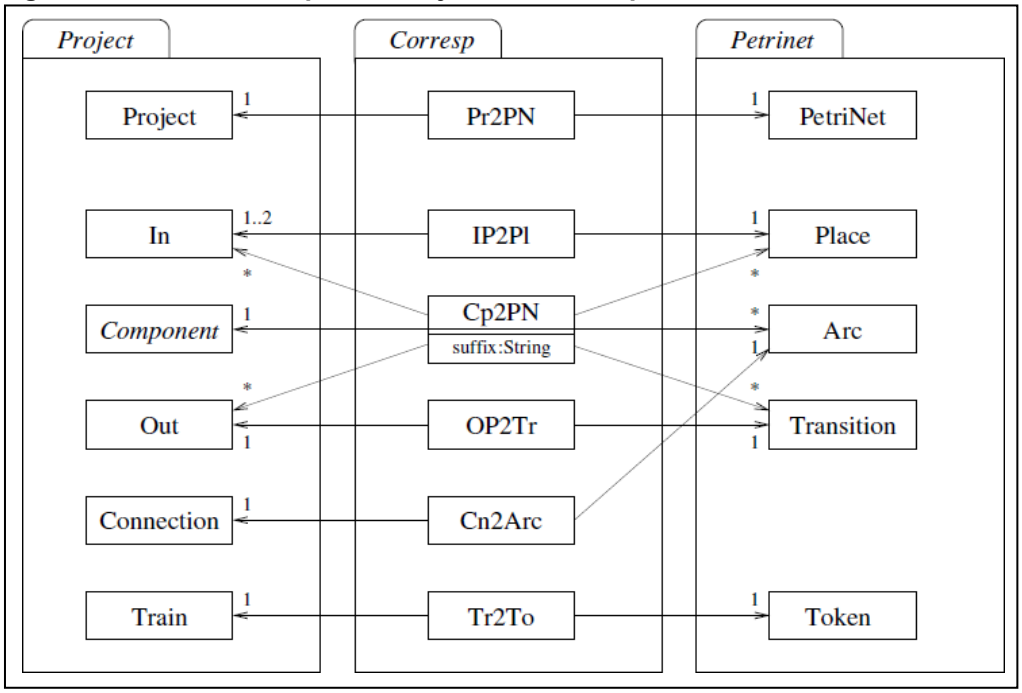

**Fonte: Kindler e Wagner (2007)**

Após o desenvolvimento de todos os componentes, obtém-se um metamodelo para os objetos de correspondência, ilustrado na Figura 17, na página 28.

A Figura 18, na página 28, ilustra um projeto que consiste em uma sequência de três *Tracks* junto com o seu modelo de Rede de Petri construído através da transformação utilizando as regras TGG.

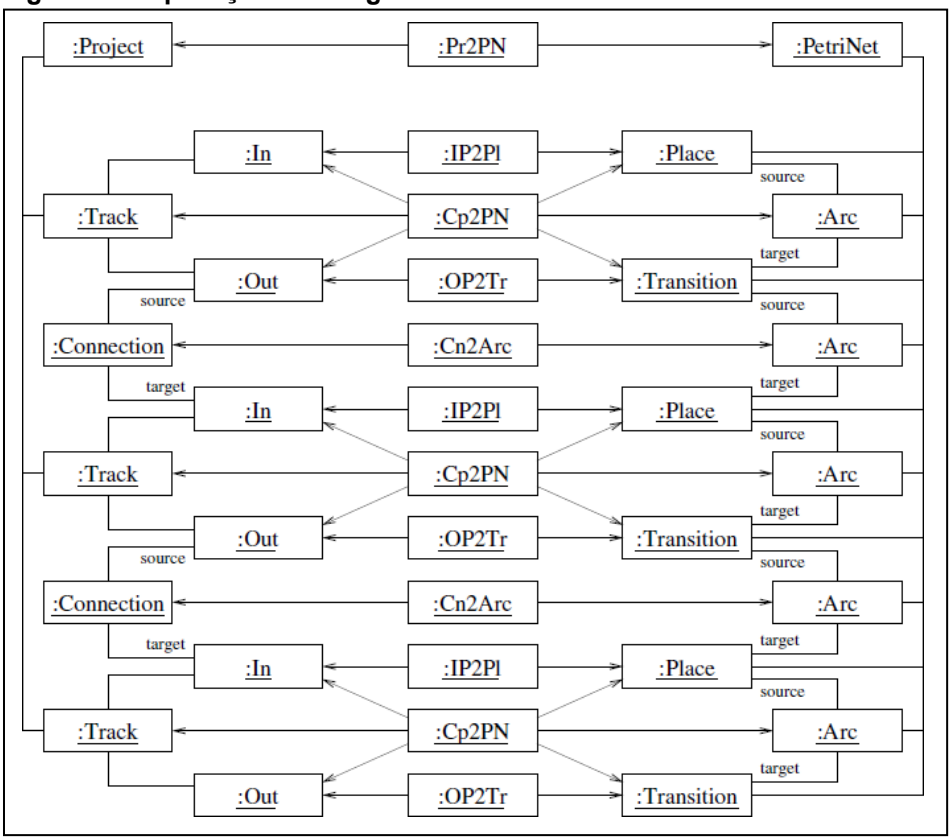
Seguindo o modelo da Figura 18, obtém-se o modelo correspondente na sintaxe gráfica, ilustrado na Figura 19, na página 29.

**Figura 17 - Metamodelo para os objetos de correspondência**



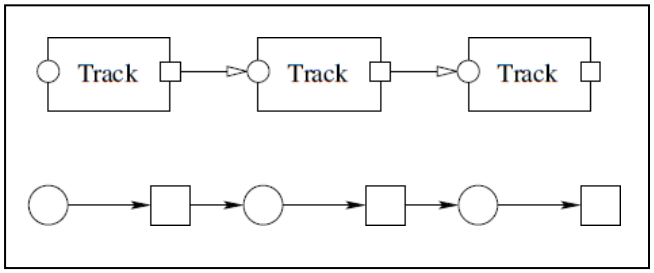

**Fonte: Kindler e Wagner (2007)**

**Figura 18 - Aplicação das Regras TGG**




**Fonte: Kindler e Wagner (2007)**

**Figura 19 - Modelos correspondentes na Sintaxe Gráfica**




**Fonte: Kindler e Wagner (2007)**

### 5.1.1 Cenários de aplicação

Conforme descrito anteriormente, a semântica do TGG é formada por três partes: modelo de origem, modelo de destino e as correspondências entre ambos os modelos. Porém na prática, os cenários mais comuns de utilização de TGG são: *Model Transformation, Model Integration* e *Model Synchronization* (KINDLER; WAGNER, 2007).

#### *5.1.1.1 Model Transformation*

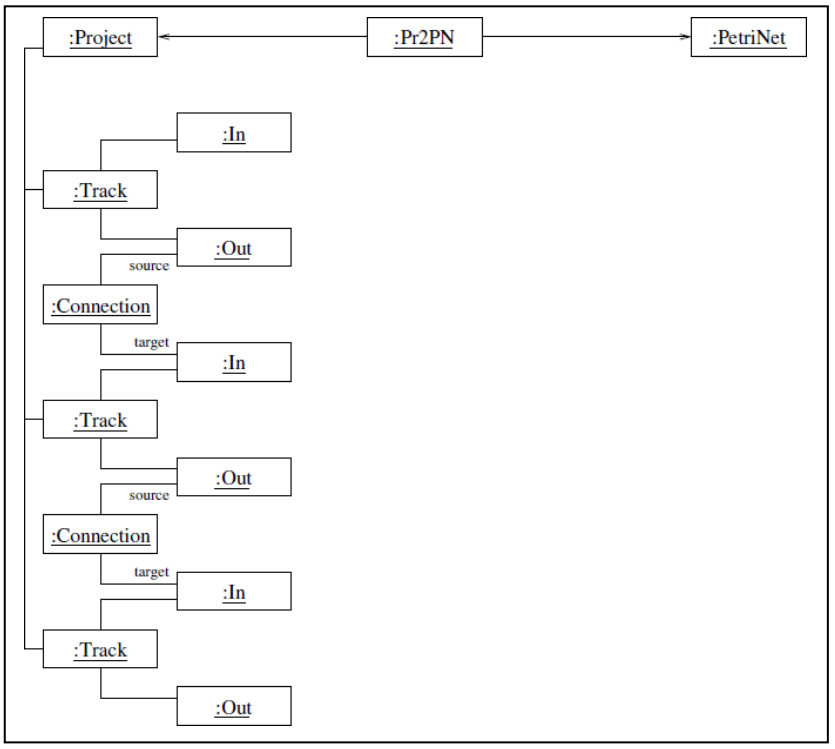
Esta transformação ocorre de um modelo A para um modelo B, em um cenário em que primeiro já existe, necessitando apenas gerar o segundo. Por exemplo, um "Projeto", o modelo A, e a partir deste, a necessidade de gerar uma Rede de Petri correspondente, modelo B.

Para a iniciar a transformação de modelos, coloca-se o Projeto ao lado esquerdo da TGG, tipicamente chamado de *source side,* ou modelo de origem. O lado direito da TGG, em que será gerado a Rede de Petri correspondente, é chamado de *target side,* ou alvo. Como esta transformação ocorre do lado esquerdo para o lado direito da TGG, é conhecida como *"forward transformation".*

A Figura 20, na página 30, ilustra o início da transformação em que se obtém apenas o diagrama de objetos de um projeto.

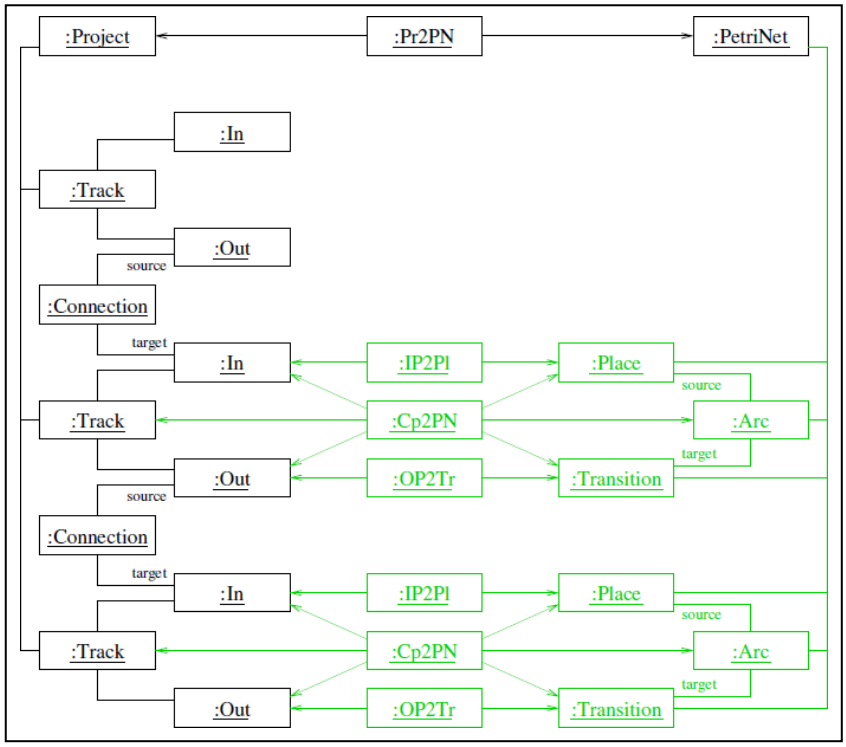
Com a o diagrama de objetos do lado esquerdo definido, são adicionados os nós de correspondência e nós do lado direito, neste caso, os nós da Rede de Petri. A Figura 21, na página 30, ilustra a transformação sendo realizada após duas aplicações da regra para o componente *Track*.

**Figura 20 - *Forward transformation***




**Fonte: Kindler e Wagner (2007)**

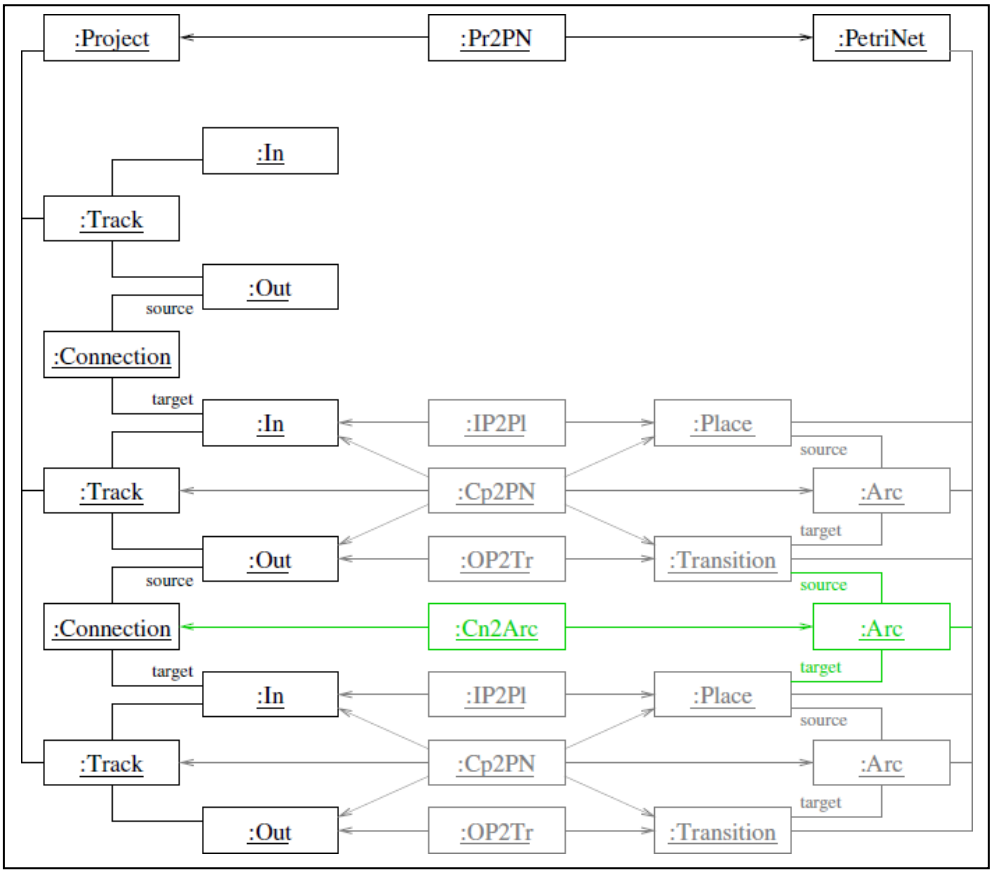
**Figura 21 - *Forward transformation* depois de duas aplicações da regra para o componente *Track***




**Fonte: Kindler e Wagner (2007)**

O próximo passo é aplicar a regra para o componente *Connection*. Esta fase é ilustrada na Figura 22.

**Figura 22 - *Forward transformation* depois da aplicação da regra para o componente *Connection***




**Fonte: Kindler e Wagner (2007)**

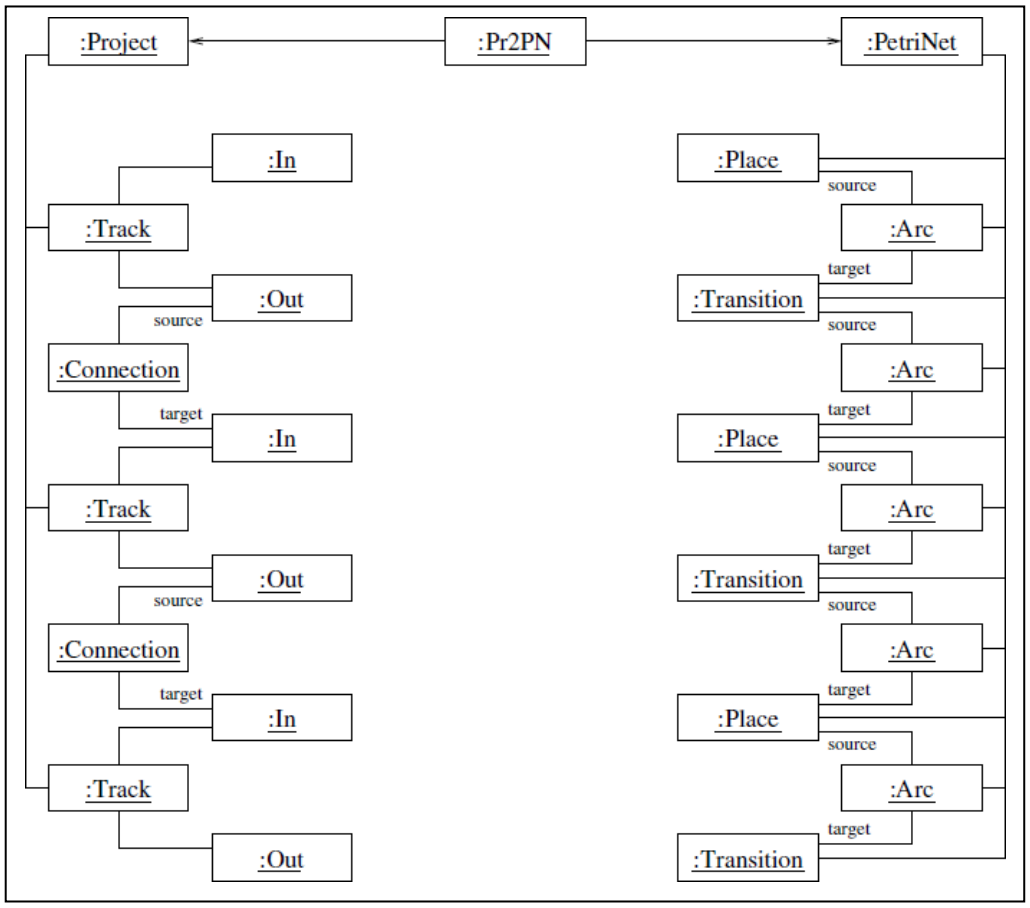
Uma vez que se tenha totalmente correspondido o modelo do projeto, é gerada a Rede de Petri correspondente. Para o exemplo da Figura 20, tem-se o modelo finalizado representado na Figura 18.

### *5.1.1.2 Model Integration*

Neste cenário se tem os dois modelos, e a necessidade é de desenvolver uma correspondência entre eles, conforme ilustrado na Figura 23, na página 32.

Quando ambos os modelos são totalmente compatíveis, é chamado de *model integration.* Neste exemplo da Figura 23, tem-se desde o início os modelos Projeto e Rede de Petri, que são compatíveis entre si, porém não há ainda as regras que definem a equivalência entre os componentes de ambos os modelos.

**Figura 23 - *Model Integration***




**Fonte: Kindler e Wagner (2007)**

Ao definir tais regras, o exemplo será finalizado na situação apresentada pela Figura 18. Porém podem haver casos em que os modelos não podem ser combinados totalmente, ou seja, um modelo não corresponde totalmente ao outro.

### *5.1.1.3 Model Synchronization*

O *Model Synchronization* é semelhante ao *Model Integration*. Além de obter os modelos de origem e destino, ainda possui a correspondência entre eles (KINDLER E WAGNER, 2007).

Os modelos não precisam necessariamente corresponder totalmente um ao outro, pois o *Model Synchronization* tem a função de modificar um ou os dois modelos, bem como a equivalência entre eles, a fim de obter uma correspondência completa e correta de acordo com as regras do TGG (KINDLER E WAGNER, 2007).

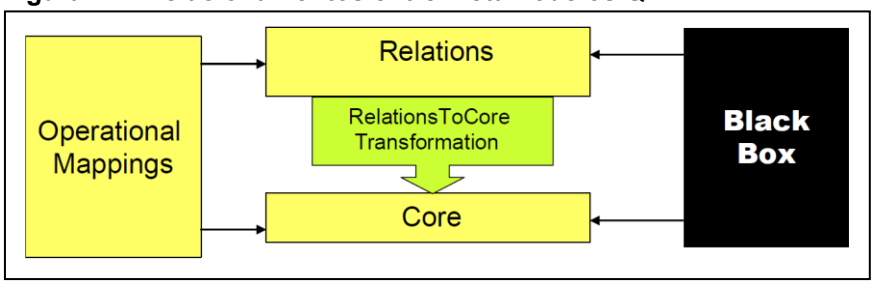
## 5.2 QUERY, VIEW, TRANSFORMATION (QVT)

*Query, View, Transformation,* ou simplesmente QVT, é uma linguagem de natureza híbrida declarativa e imperativa. Além disso, é dependente de três conceitos: consulta (*query*), visão (*view*) e transformação (*transformation*).

A parte declarativa da linguagem é dividida em dois níveis de arquitetura, a linguagem relacional (QVT-R) e a linguagem do núcleo (QVT-C, ou *Core Language*). Já a parte imperativa consiste na linguagem operacional (QVT-O) (OMG, 2016).

A especificação QVT é composta por uma arquitetura híbrida, sendo uma parte declarativa e outra imperativa (OMG, 2011). A Figura 24 ilustra as relações entre metamodelos QVT.

**Figura 24 - Relacionamentos entre metamodelos QVT**




**Fonte: OMG (2011)**

Segundo Gardner et al. (2002), o processo de transformação é realizado pela parte declarativa, ou seja, os relacionamentos entre as variáveis ocorrem utilizando as regras de inferência. Além disso, funções podem ser associadas à uma linguagem que está incorporada à parte declarativa.

Para que seja possível a aplicação de um algoritmo sobre as relações a fim se produzir um resultado, é necessário um compilador ou interpretador para a execução da linguagem. Esta camada pode ter informações suficientes para descrever uma transformação bidirecional ou unidirecional. Ao todo, existem duas camadas declarativas. São elas (OMG, 2011):

- Relação (*Relations*) – responsável por realizar uma especificação do relacionamento entre os modelos *Meta Object Facility* (MOF). A equivalência de objetos complexos e a criação de classes de rastreamento para verificar o que ocorreu durante a transformação, são possibilitadas pela linguagem utilizada nesta camada. Uma relação também pode ser utilizada para garantir que a outra relação está correta, além é claro, da

execução propriamente dita.

- *Core* – de acordo com um conjunto de condições e variáveis, tem como objetivo casar padrões. Este modelo pode ser diretamente implementado ou utilizado como referência da semântica da *Relations*. Utilizando a linguagem de transformação, uma *Relation*s pode ser transformada em Core.

A parte imperativa é composta por duas camadas: *Operational Mappings* (Mapeamento Operacional) e *Black Box* (Caixa Preta). A primeira é uma extensão das camadas declarativas e possui recursos encontrados em linguagens imperativas, como condições e *loops*. A segunda é responsável por mesclar facilidades expressas em outras linguagens como XSLT (*Extensible Stylesheet Language Transformations*) e integrar bibliotecas que não são da QVT (OMG, 2011).

A utilização de "*plug-ins*" de qualquer implementação de uma operação MOF com a mesma assinatura, possui vantagens, pois: (i) permite que algoritmos complexos possam ser codificados em qualquer linguagem de programação com uma ligação MOF; (ii) permite o uso de bibliotecas de domínios específicos para calcular os valores de propriedades do modelo. Por exemplo, matemática, engenharia, biociência, e em muitos outros domínios têm extensas bibliotecas que codificam algoritmos de domínios específicos que serão difíceis, senão impossíveis de expressar usando *Object Constraint Language* (OCL); (iii) permite que implementações de algumas partes de uma transformação sejam simplicadas.

No entanto, a implementação do "*plugin*" tem acesso a referências de objetos nos modelos, com isso podem ocorrer eventos arbitrários para esses objetos, como por exemplo, quebrar o encapsulamento. Implementações *black- box* não possuem nenhuma relação implícita com *Relations*, e cada *black-box* deve explicitamente implementar uma relação, a qual é responsável por manter as correspondências entre elementos de modelo relacionados com a implementação da *Operational*. Nestes casos, as partes relevantes dos modelos podem ser correspondidas por uma *Relations*, e passados para que a implementação seja realizada em uma linguagem mais relevante para processamento.

## *5.3* ATLAS TRANSFORMATION LANGUAGE (ATL)

ATL é uma linguagem de transformação de modelo especificada tanto em metamodelo, como em texto (ATLAS GROUP, LINA & INRIA, 2005a). Na ATL é possível por meio de um modelo fonte (*source*) obter um modelo alvo (*target*).

Segundo ATLAS group, LINA & INRIA (2005a), ATL é uma mistura de programação declarativa e imperativa, porém seu principal estilo de transformação é declarativo, pois permite expressar de forma simples as correspondências entre os elementos dos modelos fonte e alvo. No entanto, pode-se realizar uma construção imperativa, já que em alguns casos torna-se complexo tratá-lo de forma declarativa.

Um programa de transformação ATL é composto por regras que definem como os elementos de origem são combinados para criar e inicializar os elementos dos modelos alvo.

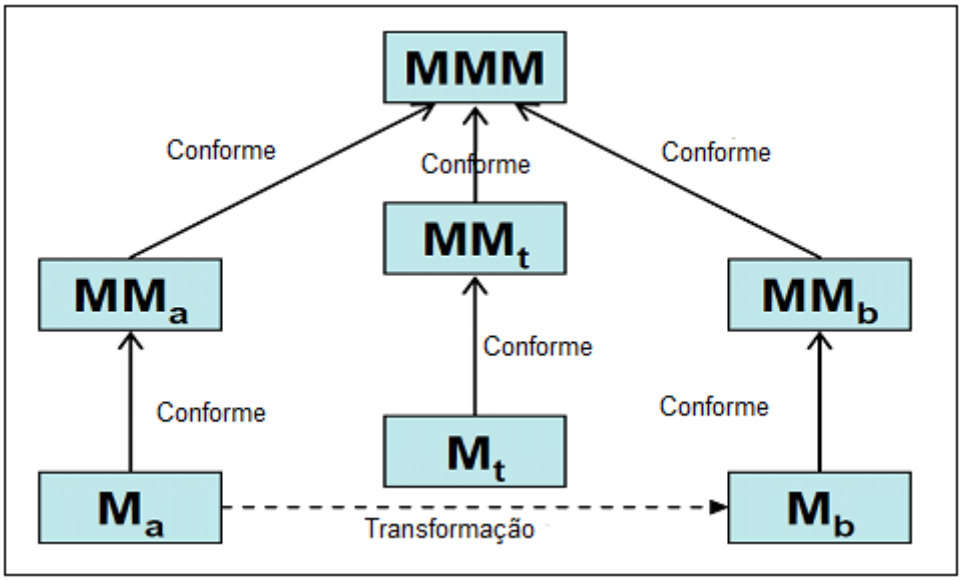
No campo da engenharia de modelo, os modelos são considerados entidades de primeira classe. Um modelo tem de ser definido de acordo com a semântica, fornecidos pelo seu metamodelo: um modelo deve estar em conformidade com o seu metamodelo, que, por sua vez, deve estar em conformidade com um metametamodelo (ATLAS GROUP, LINA & INRIA, 2005b).

Nesta arquitetura de três camadas (modelos, metamodelo e metametamodelo), o metametamodelo geralmente está de acordo com sua própria semântica, ou seja, pode ser definido de acordo com seus próprios conceitos. Como exemplos de metametamodelos, tem-se o MOF, o qual foi definido pela OMG (2002), e o *Ecore* (BUDINSKY et al., 2004a), que foi introduzido com o *Eclipse Modelling Framework* (BUDINSKY et al., 2004b).

A transformação do modelo, visa proporcionar facilidades para a geração de um modelo *Mb*, conforme um metamodelo *MMb*, de um modelo de *Ma* conforme um metamodelo *MMa* (ATLAS GROUP, LINA & INRIA, 2005B).

A Figura 25, na página 36, ilustra uma visão global do funcionamento de transformação de modelo em que um modelo *Ma*, conforme um metamodelo *MMa*, está sendo transformado em um modelo *Mb* que corresponda ao metamodelo *MMb*. Esta transformação é definida pelo modelo *Mt*, o qual satisfaz o metamodelo de transformação *MMt*. Este último metamodelo, juntamente com os metamodelos *MMa* e *MMb*, tem de estar em conformidade com um metametamodelo (tais como MOF ou Ecore), representado por *MMM*.

**Figura 25 - Visão global da transformação de modelo**




**Fonte: Adaptado de ATLAS group, LINA & INRIA (2005b)**

### 5.3.1 Módulos ATL

A estrutura de módulos ATL é composta por uma *header section,* uma *import section* opcional e um conjunto de *helpers* e *rules* que não pertencem a nenhuma seção específica, podendo assim, ser declaradas em qualquer ordem. Por meio desta estrutura é possível especificar como produzir um conjunto de modelos alvo através de um conjunto de modelos fonte (DIAS, 2009).

#### *5.3.1.1 Header Section*

Define o nome do módulo de transformação, o nome das variáveis correspondentes aos módulos fonte e alvo e codifica o modo de execução do módulo. A sintaxe da *header section* é representada no Quadro 1.

**Quadro 1 - Definição da sintaxe da *Header Section***

```
module nome_do_módulo;
create modelos_de_saída [from|refining] modelos_de_entrada;
```

**Fonte: Dias (2009)**

O *nome_do_módulo* indica a denominação do módulo e deve ser igual ao nome do arquivo ATL em que se insere. O modelo alvo é inserido após a palavra *create*, já o modelo fonte após a palavra *from* (no modo normal) ou *refining* (no caso de uma transformação "refinada").

Existem regras para declarar os modelos. Tanto para o modelo de entrada quanto para o modelo de saída, a regra a ser seguida é dada por: "*nome_do_modelo : nome_do_metamodelo".* Para declarar mais que um modelo de entrada ou saída é necessário separá-los com vírgulas, porém não podem ocorrer repetições nos nomes, pois os mesmos são utilizados para identificá-los.

O Quadro 2 ilustra um exemplo da utilização de uma *header section* do arquivo *Pai2Pessoal.atl*, utilizada para transformar modelos em que o seu metamodelo é o *Pai,* em modelos cujo metamodelo é *Pessoa*.

**Quadro 2 - Exemplo de *Header Section***

```
module Pai2Pessoa;

create OUT : Pessoa from IN : Pai;
```

**Fonte: Dias (2009)**

### *5.3.1.2 Import Section*

Neste local são declaradas as bibliotecas ATL que serão importadas. É possível declarar diversas bibliotecas, utilizando o nome da mesma seguida de sua respectiva extensão. O Quadro 3 ilustra a definição da sintaxe de uma *Import Section.*

**Quadro 3 - Definição da sintaxe da *Import Section***

```
uses nome_da_biblioteca_sem_extensão;
```

**Fonte: Dias (2009)**

O Quadro 4 apresenta um exemplo de utilização de uma *import section*. Neste caso, ocorre a importação de uma biblioteca de *Strings*.

**Quadro 4 - Exemplo de *Import Section***

```
uses strings;
```

**Fonte: Dias (2009)**

### *5.3.1.3 Helpers*

Possibilitam a definição de código fatorizado, combatendo assim a redundância. Além disso, podem ser chamados em diferentes pontos da transformação ATL. Por estes motivos, *helpers* são considerados equivalentes, no ATL, aos métodos em Java. O Quadro 5 ilustra a definição da sintaxe do *helper*.

**Quadro 5 - Definição da Sintaxe de um *Helper***

```
helper [context tipo_do_contexto]? def : nome [(parâmetros?)]? :
tipo_do_retorno = expressão;
```

**Fonte: Dias (2009)**

Um *helper* é composto pelo seu *tipo_do_contexto*, o seu *nome* e respectivo conjunto de *parâmetros*, um *tipo_do_retorno* e a expressão do *helper.*

A palavra *context* define o contexto o qual o *helper* se aplica, ou seja, o tipo de elementos a partir do qual será possível invocá-lo. Pode-se também deixar o *helper* em um contexto global do módulo ATL. Para fazer isto, omite-se o contexto da definição do *helper*.

A palavra *def* é seguida do nome do *helper* e seus parâmetros, se existentes, dentro de parênteses. Para a definição destes parâmetros, deve-se seguir a regra: *"nome_do_parâmetro : tipo_do_parâmetro".*

No caso de existirem vários parâmetros, estes deverão ser declarados separados por vírgulas, não podendo haver dois parâmetros com nomes repetidos no mesmo *helper.* Além disso, as expressões utilizadas no corpo do *helper* são especificadas pela *Object Constraint Language* (OCL) e para a definição dos tipos de contexto, de parâmetro e de retorno, devem ser dos tipos de dados suportados pela ATL.

O Quadro 6 ilustra um exemplo de *helper*, o qual é chamado de *mediaInferior* e é definido no contexto de um módulo ATL (visto que não há contexto especificado). Como resultado deste *helper,* será retornado um valor booleano, o qual é definido pelo valor da média dos valores contidos no parâmetro *seg*, sequência

(*Sequence*) de inteiros. Se esta média for menor comparativamente a um valor real dado (*valor*) o resultado será *true*, caso contrário, será *false*. A expressão *let*, é responsável por definir e inicializar a variável *media*, que por sua vez, é comparada com o valor de referência.

**Quadro 6 - Exemplo de um *Helper***

```
helper def : mediaInferior(seq : Sequence(Integer), valor : Real) :
Boolean =
let media : Real = seq->sum()/seq->size() in média<valor;
```

**Fonte: Dias (2009)**

#### *5.3.1.4 Rules*

Ao todo existem 3 tipos diferentes de regras (*rules*), são elas: *matched rules, lazy rules* e *called rules*. A primeira corresponde à programação declarativa, já as outras, à programação imperativa.

As *matches rules* permitem especificar para que tipos de elementos fonte se deve gerar elementos alvo e a maneira como os mesmo devem ser inicializados. A definição de sua sintaxe é ilustrada no Quadro 7, na página 40.

Em primeira instância, há a palavra *rule*, a qual representa que uma nova regra está sento especificada, esta palavra chave é seguida pelo *nome_da_regra*, que a identifica. Seguem os padrões das fontes e dos alvos, e pelas seções opcionais compostas por variáveis locais e imperativas. Após a palavra-chave *from* é definido o padrão fonte que permite especificar o elemento do modelo fonte, correspondente a determinado metamodelo que será transformado. O padrão alvo é introduzido pela palavra-chave *to,* e permite especificar os elementos (atributos e associações) que serão gerados quando o padrão fonte da regra tem correspondência e como estes elementos gerados são inicializados.

De acordo com as relações determinadas, a *matched rule* gera elementos alvo para cada elemento fonte que não deve ser correspondente a mais do que uma *matched rule,* nem pode gerar valores primitivos.

**Quadro 7 - Definição da Sintaxe de uma *Matched Rule***

```
rule nome_da_regra {
    from
        var_fonte : modelo_fonte[(condição)]?
    [using {
        variável_entrada1:tipo_variável_entrada1=expressão1;
        ...
        variável_entradan:tipo_variável_entradan=expressãon;
    }]?
    to
        variável_saída1 : tipo_variável_saída1(associações1),
        variável_saída2 : distinct tipo_variável_saída2 foreach(e in
collection)(associações2),
        ...
        variável_saídan : tipo_variável_saídan (associaçõesn)
    [do {
       declarações_imperativas
    }]?
}
```

**Fonte: Dias (2009)**

A palavra-chave *using,* é utilizada para declarações de variáveis locais. Estas variáveis são então declaradas e inicializadas localmente, sendo apenas visíveis no domínio da regra em que estão inseridas.

Para a execução do código imperativo após a inicialização dos elementos alvo gerados pela regra, utilizam-se as seções opcionais imperativas, as quais são introduzidas pela palavra-chave *do*.

O Quadro 8, página 41, ilustra um exemplo de uma *matched rule.* Nesta regra, a qual recebeu o nome de *InformacaoPai2InformacaoPessoa*, o principal objetivo é a transformação do modelo fonte *InformacaoPai* no modelo alvo *InformacaoPessoa.* Nota-se que esta regra possui apenas padrões obrigatórios e que o modelo alvo não define filtro algum, isto é, todas as classes *InformacaoPai* do metamodelo *MMPai* serão correspondidas pela regra. Para finalizar, o elemento padrão *t* alocará as classes *InformacaoPessoa* e as mesmas serão inicializadas com os atributos correspondentes da classe *InformacaoPai*.

Semelhantes as *matched rules*, as *lazy rules* possuem apenas uma diferença,

elas apenas podem ser aplicadas quando chamadas por outras regras.

Para a programação imperativa, as *called rules* são mais indicadas, pois para serem executadas dependem explicitamente de chamadas, podendo aceitar parâmetros. Outra característica importante desta regra, é que a mesma pode gerar elementos de modelo alvo, como as *matched rules*.

**Quadro 8 - Exemplo de uma *Matched Rule***

```
rule InformacaoPai2InformacaoPessoa {
    from
        s : MMPai!InformacaoPai
    to
        t : MMPessoa!InformacaoPessoa (
        name <- a.name,
        surname <- a.surname
    )
}
```

**Fonte: Dias (2009)**

Há apenas duas maneiras de uma *called rule* ser invocada. A primeira delas é por meio de uma *matched rule*, a segunda, por meio de outra *called rule*. O Quadro 9, página 42, ilustra a definição da sintaxe de uma *called rule.*

Tal como mencionado anteriormente, *called rule* não faz a correspondência de nenhum elemento de um modelo fonte, por este motivo, a inicialização de elementos de um modelo alvo tem de ser baseada na combinação de variáveis locais, parâmetros e atributos do módulo. A palavra-chave *to* como em uma *matched rule*, define o padrão alvo.

Uma *called rule* tem de ser única numa transformação ATL e não pode ter o mesmo nome que um *helper,* nem chamar-se *main*. O Quadro 10, página 43, ilustra um exemplo de uma *called rule* chamada *NovaPessoa*. Esta *called rule* tem como objetivo gerar elementos alvo do tipo *Pessoa*. Dois parâmetros são aceitos, o primeiro é referente ao *nome* e o segundo ao *sobrenome* do modelo *Pessoa* que será criado pela execução da regra. O padrão alvo, chamado de *t*, aloca a classe *Pessoa* sempre que a regra é invocada e inicializa o nome do atributo. O código imperativo atribui no *sobrenomep* o valor do parâmetro *sobrenome*.

**Quadro 9 - Definição da sintaxe de um *Called Rule***

```
[entrypoint]? rule nome_da_regra''(''parâmetros''){
    [using {
        variável1:tipo_variável1 = expressão1;
        ...
        variáveln:tipo_variáveln = expressãon;
    }]?
    [to
        variável_saída1 : tipo_variável_saída1(associações1),
        variável_saída2 : distinct tipo_variável_saída2 foreach(e in
collection)(associações2),
        ...
        variável_saídan : tipo_variável_saídan (associaçõesn)
    [do {
        declarações_imperativas
    }]?
}
```

**Fonte: Dias (2009)**

**Quadro 10 - Exemplo de uma *called rule***

```
rule NovaPessoa (nome: String, sobrenome: String) {
    to
        t : MMPessoa!Pessoa (
        nomep <- nome
    )
    do {
        t.sobrenomep <- sobrenome
    }
}
```

**Fonte: Dias (2009)**

# 6 CONTRIBUIÇÃO

Esse Capítulo apresenta a contribuição do trabalho e está dividido em 2 seções: (i) comparação entre as linguagens e (ii) trabalhos relacionados.

## 6.1 COMPARAÇÃO ENTRE AS LINGUAGENS

Stephan Seifermann e Jörg Henß (2017) realizaram uma comparação entre as linguagens TGG e QVT-O, de forma que na transformação em QVT, duas transformações unidirecionais foram utilizadas para complementar a transformação bidirecional permitida na linguagem TGG. Suas conclusões foram que a linguagem QVT requer um esforço maior para realizar e manter as transformações, já a TGG divide o esforço para criar, mas requer uma força considerável para manter as transformações sincronizadas após mudanças.

Amstel et al. (2011) compararam as linguagens QVT-O, QVT-R e ATL. Após realizadas as transformações e obtidas as métricas das mesmas, percebeu-se que a transformação seguindo a linguagem ATL é até cinco vezes mais rápida do que pela linguagem QVT-R ou QVT-O.

Joel Greenyer e Ekkart Kindler (2007) apresentaram três tipos de diferenças entre as linguagens TGG e QVT. São elas: filosóficas, semânticas e conceituais, além de citar características avançadas. Uma das diferenças filosóficas é a forma como o mapeamento QVT e as regras de TGG são lidas, utilizando QVT o mapeamento é lido de baixo pra cima, enquanto as regras TGG são lidas de cima pra baixo, o que implica na forma como as transformações são implementadas. Outra diferença filosófica está no fato de QVT ser unidirecional, enquanto TGG é bidirecional, mantendo os modelos sincronizados independente de mudança em algum deles. O Quadro 11 apresenta as diferenças filosóficas.

**Quadro 11 - Diferenças Filosóficas**

| | TGG | QVT |
|---|---|---|
| LEITURA | De cima para baixo | De baixo para cima |
| DIREÇÃO | Bidirecional | Unidirecional |

**Fonte: Autoria Própria**

A diferença semântica encontrada é que utilizando TGG, cada objeto tem apenas um nó de criação correspondente na regra de TGG aplicada, enquanto o padrão QVT não especifica se há apenas um correspondente para cada objeto, conforme Quadro 12.

**Quadro 12 – Diferença Semântica**

| TGG | QVT |
|---|---|
| Apenas um nó de criação correspondente | Não especifica |

**Fonte: Autoria Própria**

Como diferença conceitual é citado o fato de que o padrão QVT está mais relacionado à definição de variáveis e relação entre elas, enquanto o padrão TGG apresenta modelos gráficos, com nós e arestas. Além disso, é apresentado o fato de que a linguagem QVT foi proposta no contexto de Arquitetura Orientada a Modelos (MDA, do inglês *Model-Driven Architecture*), baseado no MOF, enquanto a linguagem TGG foi criada antes da existência do MOF, porém há implementações baseadas no *Eclipse Modeling Framework* (EMF), que é uma implementação do MOF. Essas características são apresentadas no Quadro 13.

**Quadro 13 - Diferenças Conceituais**

| | TGG | QVT |
|---|---|---|
| | Modelos gráficos | Relacionado a definição de variáveis e relação entre elas |
| CONTEXTO | EMF | MDA |

**Fonte: Autoria Própria**

Frédéric Jouault e Ivan Kurtev (2006) apresentam uma comparação entre o padrão QVT e a linguagem ATL. A primeira diferença encontrada é que essas linguagens apresentam níveis de abstração diferentes, mostrando que ATL, QVT-O e QVT-C são menos abstratos do que QVT-R.

Além disso, foram apresentados três cenários de transformação, como apresentado no Quadro 14: sincronização de modelos (suportados por QVT-C e QVT-R), verificação de conformidade (suportados por QVT-C e QVT-R) e

transformação de modelos (suportados por todos os componentes do QVT e ATL). O paradigma das linguagens também foi uma diferença encontrada, enquanto ATL tem o paradigma híbrido, QVT se divide entre declarativo e imperativo, QVT-O é imperativo e QVT-R é declarativo.

**Quadro 14 – Cenários Suportados**

| | ATL | QVT-C | QVT-R | QVT-O |
|---|---|---|---|---|
| Sincronização de modelos | | X | X | |
| Verificação de conformidade | | X | X | |
| Transformação de modelos | X | X | X | X |

**Fonte: Autoria Própria**

Da mesma forma, a direção em que as transformações são executadas, temos ATL e QVT-O como unidirecional, e QVT-R e QVT-C como bidirecional. A cardinalidade, que define a quantidade de modelos de entrada e saída, das linguagens ATL, QVT-O e QVT-R é de muitos para muitos, enquanto de QVT-C é de muitos para 1. A rastreabilidade, que é a categoria que indica se a linguagem suporta registros de correspondência entre os elementos de entrada e saída dos modelos, aparece nas linguagens ATL, QVT-R e QVT-O de forma automática, enquanto na linguagem QVT-C apenas se especificada pelo usuário. Essas características são apresentadas no Quadro 15.

**Quadro 15 – Características**

| | ATL | QVT-C | QVT-R | QVT-O |
|---|---|---|---|---|
| Direção | Unidirecional | Bidirecional | Bidirecional | Unidirecional |
| Cardinalidade | Muitos pra muitos | Muitos pra 1 | Muitos pra muitos | Muitos pra muitos |
| Rastreabilidade | SIM | Apenas se especificada pelo usuário | SIM | SIM |

**Fonte: Autoria Própria**

## 6.2 TRABALHOS RELACIONADOS

Com o objetivo de entender e conhecer as formas de transformação de modelos, foram levantados artigos que apresentam transformações utilizando os modelos UML para rede de Petri.

Um dos trabalhos que se relacionam com as transformações entre Diagrama de Atividades e Rede de Petri é o de Staines (2010). Este trabalho apresenta um mapeamento para que posteriormente possa ser realizada a transformação entre os modelos. É utilizada como base a linguagem TGG, e são criadas seis regras para a implementação da transformação, que é citada como trabalhos futuros.

Lachtermacher et al. (2008), apresenta uma transformação de Diagrama de Atividades para Rede de Petri através da linguagem QVT. Nesse trabalho é apresentado desde o mapeamento até a transformação em si, descrevendo cada decisão tomada.

O trabalho de Jamal e Zafar (2016) apresenta uma transformação tendo como modelo de entrada o Diagrama de Atividades e modelo de saída Rede de Petri Colorida. A transformação é realizada utilizando um grafo, denominado *Weighted Directed Graph,* Este grafo possui um conjunto de vértices conectados por arestas, associadas a estas estão as possíveis direções. Para a realização da transformação, primeiramente o modelo de entrada é convertido em um *Weighted Directed Graph*, que, por sua vez, é transformado no modelo de saída, preservando a lógica do Diagrama de Atividades.

Staines (2011) realiza uma transformação bidirecional do Diagrama de Atividades para Rede de Petri, utilizando a linguagem TGG. Este trabalho apresenta a transformação utilizando regras simples que são decompostas considerando a complexidade do modelo.

# 7 CONCLUSÃO

Neste trabalho foi apresentado o conceito de desenvolvimento orientado a modelos, bem como a possibilidade de realizar transformações de modelos, que consiste na definição de um conjunto de regras sobre um modelo a fim de convertê-lo em outro equivalente. Este processo visa diminuir a taxa de erros e tornar ainda mais eficiente o desenvolvimento, manutenção e atualização de sistemas, mantendo a consistência entre os vários artefatos relacionados.

Para a realização automática da transformação de modelos, há na literatura algumas linguagens próprias para esta finalidade. Neste contexto, o objetivo principal deste trabalho era: (i) a identificação das principais linguagens de transformação de modelos, e (ii) a comparação destas linguagens, apontando características, vantagens e desvantagens de cada uma.

Deste modo, foram analisados trabalhos que abordam o paradigma de transformação de modelos a fim de verificar qual o método utilizado em cada trabalho. Após a realização da análise, optou-se por estudar as linguagens *Triple Graph Grammars* (TGG), *Query/View/Transformation* (QVT) e *Atlas Transformation Language* (ATL).

Apesar de serem linguagens que possuem a mesma finalidade, as mesmas possuem características particulares, realizando a transformação de modelos de modos diferentes, como por exemplo, sincronia, cardinalidade, rastreabilidade, transformações unidirecionais, bidirecionais, entre outros.

Dentre as três linguagens analisadas, não há como determinar qual a melhor, pois há fatores específicos que determinam o engenheiro de *software* optar pela utilização de uma ou outra. O primeiro são as características de cada linguagem que podem variar de acordo com a complexidade do modelo de origem, ou seja, a mesma linguagem pode ter vantagem a outras quando utilizada em modelos simples, porém levar desvantagem quando considerada em modelos mais complexos. O segundo fator é a experiência do engenheiro na linguagem. Como se trata de uma atividade não trivial, o conhecimento avançado em uma determinada linguagem pode significar o sucesso ou fracasso na execução da tarefa de transformação de modelos.

Para uma comparação aprofundada, sugere-se como trabalhos futuros a implementação das transformações do modelo de diagrama de atividades – UML – em rede de Petri nas três linguagens citadas.

# REFERÊNCIAS

AMSTEL, Marcel V.; BOSEMS, Steven; KURTEV, Ivan; PIRES, Luís F. **Performance in Model Transformations: Experiments with ATL and QVT**. ICMT 2011, LNCS 6707, p. 198–212, 2011.

ANDRADE, Vinícius C. **Transformação de Modelos de Diagrama de Sequência UML Contemplando Restrições de Tempo e Energia para Rede de Petri Temporal.** Curitiba. 2013. 70 f. Dissertação (Mestrado) – Programa de Pós- Graduação em Informática, Universidade Federal do Paraná. Curitiba, 2013.

ATLAS group, LINA & INRIA. **ATL: ATLAS TRANSFORMATION LANGUAGE.** v.0,2. 2005a. 10 p.

ATLAS group, LINA & INRIA. **ATL: ATLAS TRANSFORMATION LANGUAGE.** v.0,1. 2005b. 20 p.

ATKISON, Colin; KÜHNE, Thomas. **Model-Driven Development: A Metamodeling Foundation.** Published by the IEEE Computer Society. 2003

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML: Guia do Usuário.** Elsevier Brasil, 2000.

BUDINSKY, F., STEINBERG, D., ELLERSICK, R., GROSE, T. **Eclipse Modeling Framework, Chapter 5 ”Ecore Modeling Concepts”.** Addison Wesley Professional. ISBN: 0131425420, 2004.

BUDINSKY, F., STEINBERG, D., ELLERSICK, R., GROSE, T. **Eclipse Modeling Framework.** Addison Wesley Professional. ISBN: 0131425420, 2004.

CARDOSO, Janette; VALETTE, Robert. **Redes de Petri.** Florianópolis: 1997.

DIAS, P. **Atlas Transformation Language: Transformação de Modelos Java em Modelos de Classes UML.** Porto, Portugal. 2009. 75 p. Relatório – DEI-ISEP. Licenciatura em Engenharia Informática. Porto. 2009.

EISHIMA, Tânia E. **Proposta de Metamodelo para Desenvolvimento Orientado a Modelo para Empresas do APL de Londrina.** Londrina – PR, 2014.

FRANCÊS, Carlos R. L. **Introdução às Redes de Petri.** 2003.

GARDNER, T., GRIFFIN, C., KOEHLER, J., et al., **A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard,** OMG Document: ad/03-08-02. 2002.

GREENYER, Joel; KINDLER, Ekkart. **Reconciling TGGs with QVT**. MoDELS 2007, LNCS 4735, p. 16–30, 2007.

JAMAL, Maryam; ZAFAR, Nazir A. **Transformation of Activity Diagram into Coloured Petri Nets Using Weighted Directed Graph**. International Conference on Frontiers of Information Technology. 2016.

KINDLER, E., WAGNER, R. **Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios.** Technical Report, University of Paderborn, Department of Computer Science. June 2007.

KINDLER, E.; RUBIN, V; WAGNER, R. **Component Tools: Integrating Petri nets with other formal methods.** In S. Donatelli and P. S. Thiagarajan, editors, Application and Theory of Petri Nets 2006, 27th International Conference, volume 4024 of LNCS, pages 37{56. Springer, June 2006.

LACHTERMACHER, Luana; SILVEIRA, Denis S.; PAES, Rodrigo de B; LUCENA, **Carlos J. P.** Transformando o Diagrama de Atividade em uma Rede de Petri. Pontificia Universidade Católica do Rio de Janeiro. Abril, 2008.

LUCRÉDIO, Daniel. **Uma Abordagem Orientada a Modelos para Reutilização de Software.** USP – São Carlos – SP. Junho de 2009.

MACIEL, Paulo R. M., et al. **Introdução às Redes de Petri e Aplicações.** X Escola de Computação de Campinas – SP. 1996.

MURATA, Tadao. **Petri nets: Properties, analysis and applications.** Proceedings of the IEEE, v. 77, n. 4, p. 541-580, 1989.

NETO, David F. **CoMDD: uma abordagem colaborativa para auxiliar o desenvolvimento orientado a modelos.** 2012.

OMG, Object Management Group - **Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification**, Version 1.1, January 2011. Disponível em: <http://www.omg.org/spec/QVT/1.1>. Acesso em: 01 mar. 2012.

OMG, Object Management Group. **OMG Unified Modeling Language$^{TM}$ (OMG UML)**, Superstructure, v. 2.4.1. 2011.

OMG. Object Management Group. **Meta Object Facility (MOF) Specification.** v. 1.4. 2002.

OMG. Object Management Group. **Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification** v. 1.3. 2016.

OMG/MOF Meta Object Facility (MOF) 1.4. **Final Adopted Specification Document.** formal/02-04-03, 2002.

PETRI, Carl Adam. **Kommunikation mit automaten.** 1962.

SEIFERMANN, Stephan; HENβ, Jörg. **Comparison of QVT-O and Henshin-TGG for Synchronization of Concrete Syntax Models.** Proceedings of the Sixth International Workshop on Bidirectional Transformations (Bx 2017), Uppsala, Suécia, Abri, 2017.

SELIC, Bran. **The Pragmatics of Model-Driven Development.** IEEE Software. set./out. 2003.

SENDALL, Shane; KOZACZYNSKI, Wojtek. **Model Transformation – the Heart and Soul of Model-Driven Software Development**. 2003.

STAINES, Anthony S. **A Triple Graph Grammar (TGG) Approach for Mapping UML 2 Activities into Petri Nets.** 2010.

STAINES, Anthony S. **Simplified Bi-Directional Transformation of UML Activities into Petri Nets.** 2011.

STAINES, Tony S. **Intuitive Mapping of UML 2 Activity Diagrams into Fundamental Modeling Concept Petri Net Diagrams and Colored Petri Nets.** 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems. p. 191-200. 2008.