DAVI AZEVEDO DE QUEIROZ SANTOS

TRANSFORMAÇÃO DE MODELOS INDEPENDENTE DE METAMODELOS USANDO REFLEXÃO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Marcos Didonet del Fabro

CURITIBA

2014

S237t Santos, Davi Azevedo de Queiroz
Transformação de modelos independente de metamodelos usando reflexão / Davi Azevedo de Queiroz Santos. – Curitiba, 2014.
74f. : il. color. ; 30 cm.

Dissertação (mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2014.

Orientador: Marcos Didonet del Fabro.
Bibliografia: p. 60-65.

1. Engenharia de software. 2. Projeto de sistemas. 3. Simulação (Computadores). I. Universidade Federal do Paraná. II. Fabro, Marcos Didonet del. III. Título.

CDD: 005.12

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Davi Azevedo de Queiroz Santos, avaliamos o trabalho intitulado, *"Transformação de modelos independente de metamodelos usando reflexão"*, cuja defesa foi realizada no dia 03 de abril de 2014, às 10:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela:

(X) **aprovação** do candidato. ( ) **reprovação** do candidato.

Curitiba, 03 de abril de 2014.

Prof. Dr. Marcos Didonet Del Fabro
DINF/UFPR – Orientador

Prof. Dr. Marco Aurélio Wehrmeister
UTFPR – Membro Externo

Prof. Dr. Fabiano Silva
DINF/UFPR – Membro Interno

DAVI AZEVEDO DE QUEIROZ SANTOS

TRANSFORMAÇÃO DE MODELOS INDEPENDENTE DE METAMODELOS USANDO REFLEXÃO

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comisso formada pelos professores:

Orientador: Prof. Dr. Marcos Didonet del Fabro
DINF/UFPR

Prof. Dr. Marco Aurélio Wehrmeister
UTFPR - membro externo

Prof. Dr. Fabiano Silva
DINF/UFPR - membro interno

Curitiba, 03 de abril de 2014

AGRADECIMENTOS

Ao professor e orientador Marcos Didonet del Fabro, pela orientação e suporte.

Aos meus professores de graduação, Gustavo Augusto Lima de Campos e Jerffeson Texeira de Souza.

À CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pela concessão da bolsa.

Finalmente agradeço à minha família por tudo.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A integração de técnicas de programação por restrições em plataformas de engenharia dirigida por modelos (MDE) tem como objetivo principal usar resolvedores de programação por restrições para execução de operações sobre modelos. As operações são chamadas operações de busca, pois buscam instâncias válidas de modelos baseada em um conjunto de restrições.

Entretanto, as plataformas MDE existentes possuem formatos incompatíveis com os formatos das soluções dos resolvedores atuais. Por isso é necessário desenvolver mecanismos de interoperabilidade. Existem diferentes abordagens para automatizar esta integração, normalmente baseada na execução de uma cadeia de transformações, sendo uma das mais conhecidas chamada "modelagem como busca". Esta abordagem especifica quais operações são necessárias para, a partir de um modelo e metamodelo de entrada, produzir uma especificação no formato do resolvedor, e como transformar o resultado produzido por um resolvedor em um modelo e metamodelo.

Entretanto, esta e as demais abordagens são baseadas em transformações de modelo que são implementadas sobre os elementos de metamodelos. Com isso, para cada metamodelo de saída, é necessário implementar uma transformação do formato do resolvedor para um modelo conforme a este metamodelo, tornando a abordagem inviável.

Este trabalho propõe um conjunto de transformações para automatizar a cadeia de operações da abordagem de modelagem como busca. Dentre as diferentes transformações, damos destaque a transformação do modelo do formato do resolvedor para o metamodelo de saída. Esta transformação será desenvolvida sem conhecimento prévio dos elementos do metamodelo, estes serão interpretados durante a execução, sendo assim chamada de "transformação independente de metamodelos". Apresentaremos a formalização da cadeia de operações, e esta será implementada em dois casos de uso, mostrando sua factibilidade.

Palavras-chave: engenharia dirigida por modelos, transformação de modelos

ABSTRACT

The integration of constraint programming techniques with model-driven engineering (MDE) platforms has as main objective to use constraint programming solvers to perform operations on models. These operations are called search operations, because they search valid model instances based on a set of constraints.

However, the existing MDE platforms have incompatible formats with the formats of the solutions produced by the current solvers. Therefore it is necessary to develop mechanisms for interoperability. There are different approaches to automate that integration, usually based on the execution of a chain of transformations, one of the most known approach is called “model search”. This approach specifies what operations are needed to produce, from an input model and metamodel, an specification in the solver’s format, and how to turn the result produced by a solver into a model and metamodel.

Yet, this and the other approaches are based on model transformations that are implemented over the elements of metamodels. Thus, for each output metamodel, it is necessary to implement a transformation of the solver’s format to a model conforming to this metamodel, making the approach difficult to implement.

This work proposes a set of transformations to automate the chain of operations of the model search approach, focusing on the interoperability part. Among the different transformations, we highlight the model transformation from the solver’s format to the output metamodel. This transformation will be developed without prior knowledge of the metamodel elements, these are interpreted at runtime, thus being called ”metamodel independent transformation.”We will present the formalization of the chain of operations, and this chain will be implemented in two use cases, showing its feasibility.

Keywords: model-driven engineering, model transformation

CAPÍTULO 1

INTRODUÇÃO

A Engenharia Dirigida por Modelos (Model-Driven Engineering, MDE) [12] é uma abordagem de desenvolvimento de software onde modelos são artefatos de primeira ordem [37], e não são usados apenas como entrada para atividades de geração de código ou documentação. Isto diferencia esta abordagem de métodos como o RUP, onde modelos são usados essencialmente para documentação [40].

Um modelo é uma representação de um sistema, utilizado para um determinado objetivo, que tenta extrair algumas de suas características de forma simplificada [12]. Temos como exemplo um modelo representando um sistema de gestão de controle acadêmico, onde um aluno é um elemento deste modelo que possui atributos como nome, idade, e relaciona-se com outros elementos como disciplina e histórico. Estes relacionamentos podem indicar que um aluno está matriculado em uma disciplina e possui um histórico com as notas das disciplinas cursadas.

Os tipos dos elementos pertencentes a um modelo e seus relacionamentos são definidos por um modelo chamado metamodelo [19]. A relação de tipagem entre um modelo e seu metamodelo é chamada de conformidade [19]. Entretanto, metamodelos não possuem sempre capacidade de expressão adequada, pois definem informações essencialmente estruturais e é necessário incluir restrições mais específicas [37]. Por exemplo, a idade de um aluno não pode ser negativa.

Searchbased MDE (SBMDE) é uma área recente que estuda a integração de técnicas de MDE com programação por restrições [32]. As aplicações mais comuns de SBMDE são a verificação automática e manual de modelos UML [13, 27], Linhas de Produto de Software [15], e mais recentemente abordagens como JTL [14], Modelagem como Busca [37] e Transformação como Busca [38]. As abordagens de programação por restrições possuem resolvedores que, a partir de um conjunto de variáveis, restrições e valores de

entrada, tentam encontrar uma solução (conjunto de valores atribuídos às variáveis) que satisfaça todas as restrições especificadas [37].

As soluções de programação por restrições atuais não possuem integração completa com o MDE [14, 27], pois as soluções de cada resolvedor possuem formatos de representação e ferramentas específicos e incompatíveis. Isto dificulta a integração de ambas abordagens. Dizemos que a abordagem MDE e programação por restrições são definidas em espaços tecnológicos diferentes. Espaços tecnológicos representam o contexto para a especificação e representação dos modelos [12], como por exemplo: XML [58], Ecore [56], SQL [26], MOF [49] e soluções de programação por restrições. Cada espaço tecnológico possui sua própria terminologia, formalismo, capacidades e conjunto de ferramentas de suporte [42]. Por exemplo, o que é chamado de metamodelo no espaço tecnológico MDE corresponde a um esquema no espaço tecnológico do XML. É importante possuir técnicas para converter modelos de um espaço tecnológico para outro, para ser possível aproveitar as capacidades de diferentes espaços tecnológicos. Nesta dissertação, usaremos o MDE como espaço tecnológico base.

Para realizar operações de conversão entre modelos, usamos transformações de modelos [12]. Transformações de modelos recebem um ou mais modelos de entrada e produzem um ou mais modelos de saída [11]. Quando transformações de modelo são desenvolvidas, é necessário conhecer o metamodelo de entrada e o metamodelo de saída. Por exemplo, para transformar diagramas de classes em tabelas [22], devemos conhecer os elementos do metamodelo de classes e os elementos do metamodelo de tabelas. Transformações que convertem modelos entre diferentes espaços tecnológicos são chamadas de injetores ou extratores.

Se considerarmos o MDE como espaço tecnológico de referência, um injetor é uma transformação que converte uma representação de um espaço tecnológico X para o espaço tecnológico MDE. Um extrator é a transformação inversa, do espaço tecnológico MDE para o espaço tecnológico X [12].

1.1 Motivação

Existem diferentes abordagens que permitem fazer a conexão entre o espaço tecnológico MDE e o espaço tecnológico de programação por restrições de forma automatizada, como por exemplo, a Modelagem como Busca [37].

A Modelagem como Busca (*Model Search*) é uma abordagem que usa programação por restrições para aplicar operações de busca de modelos válidos, isto é, dado um conjunto de elementos de um modelo, não conectados, como entrada, um resolvedor é utilizado para encontrar um ou vários modelos de saída válidos. Uma aplicação de exemplo são as linhas de produto de software. Uma extensão da modelagem como busca chamada Transformação como Busca (Transformation as Search, TAS) [38] é aplicada especificamente para a operação de transformação de modelos.

O funcionamento da modelagem como busca consiste em encontrar um modelo de saída a partir do metamodelo alvo, suas restrições e um modelo parcial. Este modelo parcial é composto por atribuições iniciais de elementos e relacionamentos do modelo, que não precisam, necessariamente, satisfazer todas as restrições do metamodelo. Em outras palavras, um conjunto de elementos não conectados e não conforme ao metamodelo. O resolvedor, quando executado, pode produzir uma ou várias soluções, dependendo do modelo parcial e das restrições do metamodelo. Essa capacidade de produzir várias soluções para um mesmo conjunto de entrada é útil em diversos cenários, como em aplicações de linha de produto de software.

A abordagem da modelagem como busca é formada por uma cadeia com cinco etapas [37]:

1. Transformação do metamodelo do problema em um modelo de programa de busca.

2. Transformação do modelo parcial em dados do programa de busca. Ao término destas duas etapas, temos o modelo do programa de busca, que contém os elementos do metamodelo, suas restrições e o modelo com as atribuições iniciais.

3. Extração do modelo do programa de busca do espaço tecnológico MDE para o espaço tecnológico do resolvedor. Por exemplo, um arquivo texto com o programa a ser

executado.

4. Execução do programa pelo resolvedor com a obtenção da sua solução. Esta fase é a execução da operação de busca.

5. Injeção desta solução no espaço tecnológico MDE, de forma a gerar um modelo em conformidade com o metamodelo de saída. Este metamodelo não precisa ser conhecido durante a especificação desta cadeia.

Entretanto, não há abordagens atualmente que implementem toda esta cadeia de forma genérica, em especial a última etapa. Isto porque as ferramentas existentes de injeção e extração precisam conhecer o metamodelo de saída para que a transformação seja codificada. Isto é, é preciso escrever uma transformação do formato de saída do resolvedor para cada metamodelo de saída diferente. Porém, na modelagem como busca e na transformação como busca, isto deveria ser evitado, escrevendo apenas uma operação de extração de um espaço tecnológico para outro, sem ser dependente do metamodelo de saída para a sua especificação. Para evitar essa limitação, nesta dissertação apresentaremos uma solução de transformação de modelos independente de metamodelo para injeção da solução de um resolvedor no espaço tecnológico MDE.

1.2 Objetivos

O objetivo deste trabalho é propor e implementar uma abordagem de injeção do espaço tecnológico do resolvedor para o espaço tecnológico MDE sem conhecer o metamodelo de saída durante sua implementação.

O metamodelo de saída será um parâmetro de entrada que será interpretado durante a execução usando técnicas de reflexão. Esta transformação será escrita usando os elementos do metametamodelo. Por isso, sua implementação será independente do metamodelo de saída. A transformação será chamada de transformação independente de metamodelo.

Implementaremos também transformações de suporte, no total de 4, para levar em conta a diferença de formatos entre o espaço tecnológico dos resolvedores e o espaço tecnológico MDE.

Para cumprir tal objetivo, foi proposta uma cadeia composta com as quatro transformações de modelos descritas abaixo:

1. Injeção do programa do resolvedor para um modelo de programa do resolvedor no espaço tecnológico MDE. O modelo do programa do resolvedor contém informações do metamodelo de saída e suas restrições;

2. Injeção da solução do programa do resolvedor no espaço tecnológico MDE. Esta solução contém os elementos do modelo de saída;

3. Transformação do modelo produzido pela primeira transformação no metamodelo de saída. Vale salientar que o metamodelo e o modelo do resolvedor são sempre os mesmos em uma execução. Os elementos variáveis são os metamodelos e modelos da solução (saída);

4. Transformação do modelo da solução do resolvedor em um modelo em conformidade com o metamodelo de saída. A implementação desta transformação será independente do metamodelo, isto é, não será necessário conhecer o metamodelo de saída (produzido por 3) durante sua implementação. Este será apenas interpretado durante a execução.

Assim, quando a cadeia de modelagem como busca for executada, as soluções do resolvedor poderão ser injetadas no espaço tecnológico MDE de forma automática e com a especificação de uma única transformação na etapa 4, para qualquer metamodelo ou modelo de saída.

Após esta implementação, o processo de validação e análise dos resultados foi realizado utilizando dois estudos de caso: linhas de produto de software [15] para a modelagem como busca e a transformação Class2Relational [22] para a transformação como busca. A validação foi feita comparando os elementos gerados com os esperados e com verificação do formato correto do modelo através da ferramenta do Eclipse. O resolvedor utilizado foi o Alloy Analyser [30]. Algumas ferramentas já existentes foram utilizadas como o injetor do programa Alloy de [37]. Para as outras transformações utilizou-se o ATL [2] e Java com o *framework* EMF [56] para suas implementações.

1.3 Organização do texto

Este trabalho está organizado da seguinte forma:

No próximo capítulo é realizada uma síntese dos principais conceitos utilizados neste trabalho, como a engenharia dirigida por modelos, as ferramentas de transformação de modelos e programação por restrições.

No capítulo três é apresentado a modelagem como busca e a transformação como busca, abordagens às quais este trabalho visa contribuir.

No capítulo quatro é apresentado a solução proposta através de uma cadeia de transformações, onde a última é denominada transformação independente de metamodelos usado reflexão.

No capítulo cinco são apresentados os resultados obtidos. Finalmente, no capítulo seis temos as conclusões e sugestões de trabalhos futuros.

CAPÍTULO 2

REVISÃO BIBLIOGRÁFICA

Neste capítulo será realizada uma revisão bibliográfica dos conceitos usados nesta dissertação. Esta dissertação abrange conceitos de duas áreas, engenharia dirigida por modelos e programação por restrições, além da integração de ambas. Estes são os conceitos e ferramentas em que este trabalho se baseia.

2.1 Engenharia Dirigida Por Modelos

A Engenharia Dirigida por Modelos (Model-Driven Engineering, MDE) [35] é uma abordagem que usa modelos como artefatos de primeira classe em todas as fases do ciclo de vida do processo de desenvolvimento de software. Recentemente em [12], tem-se utilizado o termo engenharia de software dirigida por modelos (Model-Driven Software Engineering - MDSE) para os usos do MDE dentro da Engenharia de Software. Há diferentes abordagens baseadas no MDE como Model-Driven Development (MDD) [12]; Model-Driven Architecture (MDA) [48], que é uma visão particular da MDD proposta pela Object Management Group (OMG); e Model-Based Engineering (MBE) [12]. A figura 2.1 ilustra a hierarquia dessas siglas. Como exemplos de MDE temos o Eclipse Modeling Framework [56], o Meta-Object Facility [49].

Um modelo pode ser definido como a representação de um sistema que possui duas

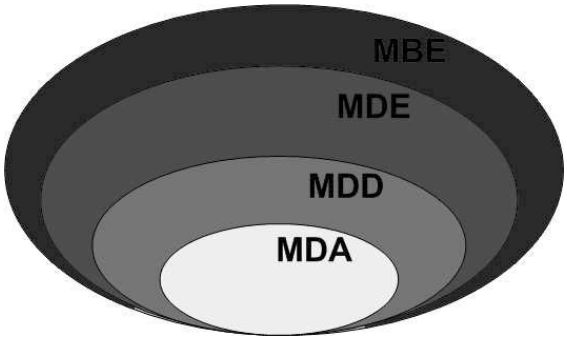


Figura 2.1: Hierarquia das siglas MBE, MDE, MDA e MDD [12]

utilidades [12]: refletir uma parte das propriedades e entidades relevantes do sistema e representar o sistema de forma abstrata e generalizada.

Um sistema é um grupo de elementos que interagem, relacionam-se, ou são independentes e que formam um todo [19]. Podemos usar um modelo para representar um sistema para um determinado objetivo. Os modelos podem ter diferentes propósitos [12]: podem ser descritivos (descrevem a realidade de um contexto ou sistema, como um esquema de dados), prescritivos (determinam o escopo e detalhes, como os diagramas UML) ou definem como um sistema será implementado. Os modelos podem ser classificados como estruturais (ou estáticos), que focam no aspecto da representação da estrutura e dados do sistema com o diagrama de classe da UML [53], ou comportamentais (ou dinâmicos), que enfatizam a sequência de ações e algoritmos, além de mostrar a colaboração entre componentes e mudanças de seus estados internos [12], como por exemplo os diagramas de estados e sequência da UML [53].

Um passo natural para a definição de modelos é especificar estes modelos como instâncias de outros modelos mais abstratos [12]. Estes modelos, chamados de metamodelos, definem as informações de tipagem e os relacionamentos possíveis entre os elementos de um modelo [19], ou de acordo com [42], um metamodelo é um modelo de uma linguagem de modelagem. A relação entre o modelo e seu metamodelo é chamada de conformidade, isto é, um modelo está em conformidade com seu metamodelo. Esta relação é frequentemente indicada pela sigla c2 (conformsTo). Tomando como exemplo um documento XML [58] (modelo) e seu respectivo esquema XSD [59] (metamodelo), o esquema XSD especifica quais elementos e atributos o documento XML deve possuir. Da mesma forma que existe a relação entre um modelo e seu metamodelo, podemos ter a relação entre o metamodelo e outro modelo que especifica quais são os tipos e relacionamentos do metamodelo. Este modelo é chamado de metametamodelo. Um metametamodelo é um modelo que especifica a representação base para todos os modelos e metamodelos de um dado contexto. Um metamodelo está em conformidade consigo mesmo [19].

A seguir temos as definições destes termos de acordo com [19] e [38]:

Definição 2.1 *Um sistema é um grupo de elementos que se interagem, relacionam-se ou*

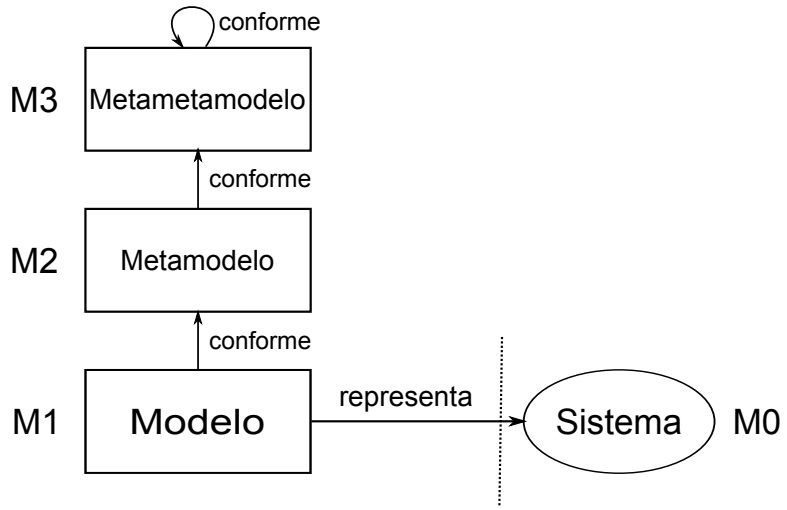


Figura 2.2: Arquitetura de três níveis [19]

são independentes, e que formam um todo.

Definição 2.2 *Um modelo é um artefato que representa um sistema. Um modelo M é uma tripla* $< G, \omega, \mu >$ *onde G é um multigrafo rotulado acíclico;* ω *(modelo de referência de M) é um outro modelo ou o mesmo modelo (uma auto-referência);* μ *é uma função que associa vértices e arestas do grafo G para os nós do grafo* G_ω *(o grafo associado a seu modelo de referência* ω*).*

Definição 2.3 *Um metamodelo é um modelo que define o tipo dos elementos e relacionamentos dos elementos de um modelo.*

Definição 2.4 *Um metametamodelo é um modelo que especifica a representação base para todos os modelos e metamodelos de um dado contexto. Um metamodelo está em conformidade consigo mesmo.*

Definição 2.5 *A relação entre um modelo e seu metamodelo é chamada de conformidade e é notada como conformsTo (ou abreviado como c2).*

A figura 2.2 mostra as relações entre sistema, modelo, metamodelo e metametamodelo. M3 é o metametamodelo. M2 é o metamodelo. M1 é o modelo que representa o sistema M0. Como o sistema está fora da fronteira dos modelos, temos apenas três níveis.

Esta arquitetura de três níveis é suficiente para ser aplicada em diferentes contextos. Na figura 2.3 temos a ilustração de três exemplos onde esta arquitetura pode ser aplicada.

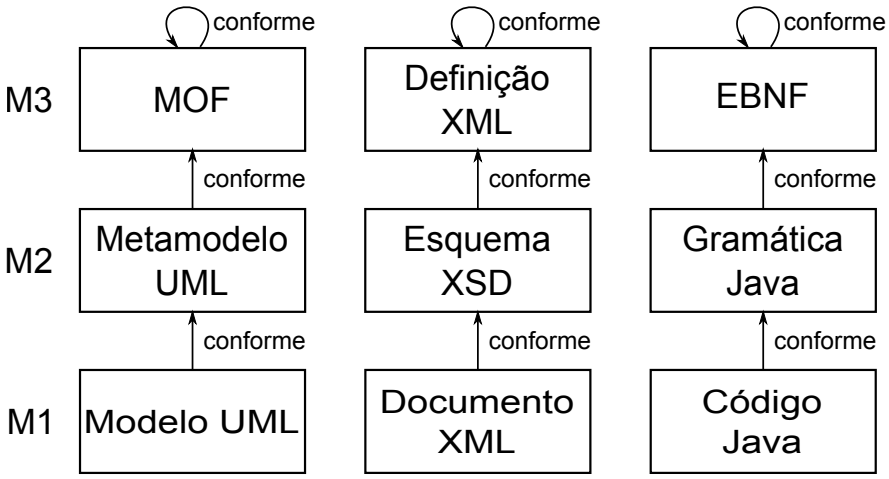


Figura 2.3: Exemplos da arquitetura de três níveis [19]

O MOF (Meta-Object Facility) da OMG [49] é o metametamodelo que fornece uma base independente de plataforma para a MDA. Podemos definir um metamodelo de classe que está em conformidade com o MOF. Este metamodelo define os tipos e relacionamentos dos elementos de um diagrama de classe, como associações e herança. Em seguida temos o exemplo do XML [58], que também pode ser especificado em três níveis. O documento XML (modelo) está em conformidade com um esquema XSD [59] (metamodelo). Este esquema define a validade da forma de um documento XML. Por sua vez, o esquema XSD está em conformidade com a definição de um arquivo XML (metametamodelo). Também podemos representar um código de uma linguagem de programação qualquer. O código de um programa (modelo) está em conformidade com a gramática de uma linguagem de programação (metamodelo), e esta linguagem é definida pela notação EBNF [55] (metametamodelo). Todos os modelos e metamodelos do MDE são projetados seguindo essa arquitetura de três níveis [19].

Como temos diferentes formatos de representação, dizemos que cada forma de representação delimita um espaço tecnológico [12]. Um espaço tecnológico representa o contexto para a especificação e implementação de aplicações [12]. Eles possuem uma linguagem para especificar os metamodelos e formatos para armazenar estes modelos, além de um conjunto de ferramentas de suporte. Por exemplo, o espaço tecnológico do EMF [56] possui como metametamodelo o Ecore e usa XMI [50] como formato para armazenar os modelos. O MOF [49] possui um metametamodelo diferente, apesar de usar também

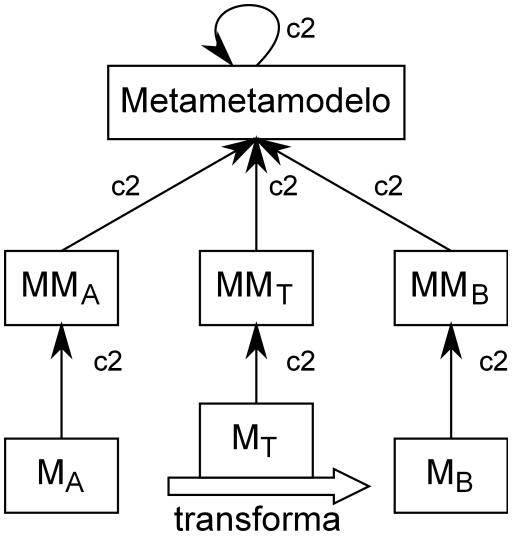


Figura 2.4: Esquema base da transformação de modelos

XMI como formato de armazenamento de seus modelos. Ligar estes diferentes espaços tecnológicos de forma uniforme é necessário [42], porque os espaços tecnológicos podem possuir capacidades e ferramentas diferentes. É preciso de uma operação que realize a integração e manipule estes modelos em espaços tecnológicos diferentes ou até mesmo dentro do mesmo espaço tecnológico. Estas operações são chamadas de transformações de modelos.

2.1.1 Transformação de Modelos

Transformação de modelos é a principal operação do MDE e é definida usando modelos de transformação. Isto é, a própria transformação é definida por um modelo. Estes modelos definem o mapeamento entre os metamodelos de entrada e saída. Elas possuem diversas aplicações [18]: gerar modelos de baixo nível (código), mapear, sincronizar e evoluir modelos, engenharia reversa, entre outras.

A figura 2.4 mostra o esquema básico de uma operação de transformação de modelos. Neste exemplo, M_a é o modelo de entrada que é transformado para o modelo de saída M_b. M_a está em conformidade com MM_a (metamodelo de M_a) e M_b com MM_b (metamodelo de M_b), isso é indicado pelas setas c2. O modelo da transformação M_t está em conformidade com MM_t (metamodelo de M_t) e define as regras de transformação entre os metamodelos de entrada e saída. MM_t é o modelo da linguagem de transformação. M_t contém as

operações que são executadas para transformar M_a em M_b. Todos os metamodelos estão em conformidade com o mesmo metametamodelo. A seguir temos uma definição para transformação de modelos conforme [19]:

Definição 2.6 *Transformação de modelo é uma operação que recebe um conjunto de modelos como entrada, visita os elementos destes modelos e produz um conjunto de modelos de saída.*

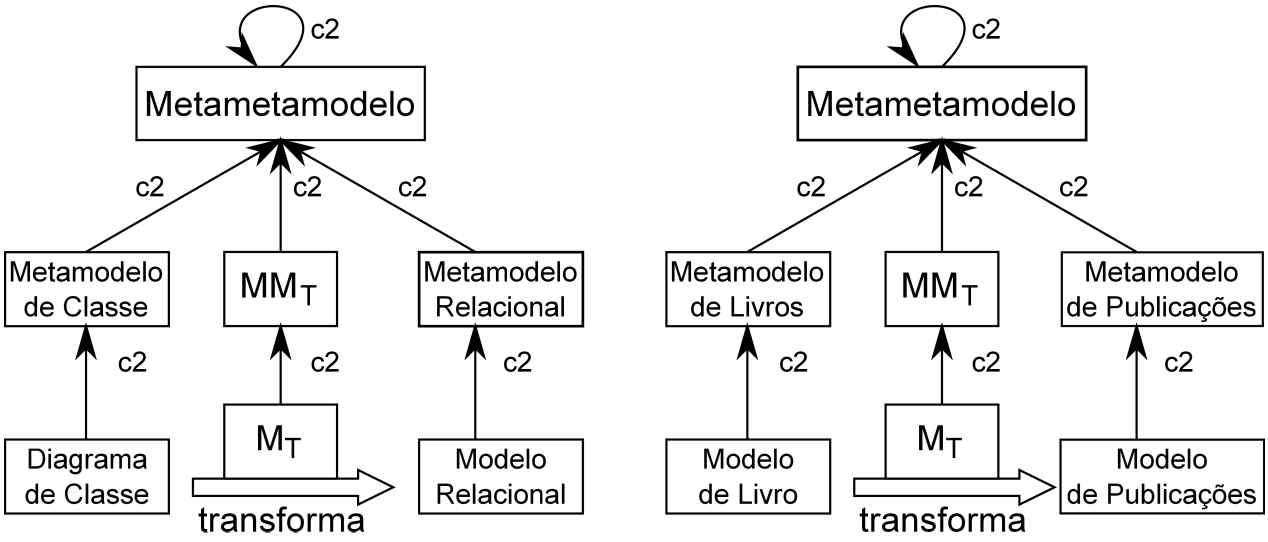
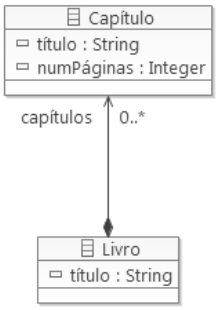
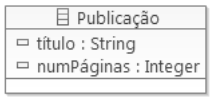


Figura 2.5: Exemplos de transformações de modelos [22]

Na figura 2.5, temos dois exemplos de transformações. A primeira consiste em transformar um diagrama de classe em um modelo relacional [22]. A segunda consiste na transformação de um modelo de livros para um modelo de publicações [21]. Temos na figura 2.6(a) o metamodelo de livros e na figura 2.6(b) o de publicações. O metamodelo de livros determina que a classe Livro possui os atributos título e um conjunto de capítulos, e a classe Capítulo possui os atributos título e número de páginas deste capítulo. O metamodelo de publicações contém uma única classe Publicação, que possui os atributos título da publicação e número de páginas totais. A transformação consiste em transformar cada instância da classe Livro em uma instância da classe Publicação, sendo que o título da publicação é o título do livro e o número de páginas da publicação é a soma do número de páginas de todos os capítulos deste livro. Na figura 2.7 temos um exemplo de

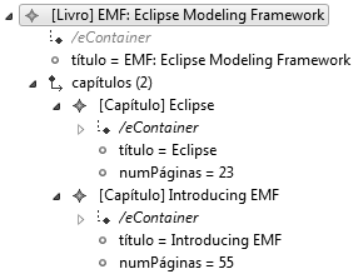


(a) Metamodelo de Livros



(b) Metamodelo de Publicações

Figura 2.6: *Metamodelos de entrada e saída do exemplo Livro para Publicações [21]*



(a) Modelo de Livros (entrada)

(b) Modelo de Publicações (saída)

Figura 2.7: *Exemplo de modelo de entrada da transformação e sua respectiva saída*

modelo a ser transformado (figura 2.7(a)) e o resultado da transformação em um modelo de Publicações (figura 2.7(b)).

Quando os modelos de entrada e saída estão em espaços tecnológicos diferentes, é possível transformar estes modelos de um espaço tecnológico para outro [12]. Esta operação é denominada injeção ou extração [37].

- A injeção é uma transformação de um espaço tecnológico para o espaço tecnológico base (MDE).
- A extração é uma transformação do espaço tecnológico base (MDE) para outro espaço tecnológico.

Injetores e extratores podem ser implementados de várias formas, código nativo [2], Textual Concrete Syntax [34], ou usando ferramentas de transformação que realizem transformação modelo-para-texto [12].

A seguir, temos um resumo da classificação dos problemas que envolvem as transformações e das ferramentas utilizadas para realizá-las feito por [11]. Czarnecki el al. [18], Mens et al. [43] e Huber [28] realizaram a classificação das principais linguagens e ferramentas utilizadas para a transformação de modelos. Biehl [11] sintetiza estas abordagens em dois tipos: classificação para problemas de transformação de modelos e classificação para Linguagens de Transformação de Modelos.

2.1.2 Classificação para Problemas de Transformação de Modelos

Um problema de transformação de modelos é definido como um problema que deve ser resolvido usando transformação de modelos [11]. Logo, é preciso uma classificação da relação entre os modelos de entrada e saída. Esta relação possui as seguintes propriedades segundo [11]:

- **Mudança de Abstração**:
  - O nível de abstração é a medida da quantidade de detalhes de um modelo[11]. A transformação de modelos pode mudar o nível de abstração inserindo ou removendo detalhes, ou mesmo não realizar nenhuma alteração. Esta propriedade é independente da mudança do metamodelo, isto é, os metamodelos de entrada e saída podem ser o mesmo ou diferentes. Há dois tipos: transformação vertical e transformação horizontal.
  - Uma transformação vertical muda o nível de abstração. Se houver a inserção de detalhes no modelo, a transformação é chamada de transformação de refinamento [11].
  - Uma transformação horizontal muda a representação do modelo, mas não muda seu nível de abstração. Este tipo de transformação produz o modelo de saída em um metamodelo diferente e mantendo o mesmo nível de abstração [43].
- **Mudança de Metamodelos**:

  - Se os metamodelos de entrada e saída são os mesmos, temos uma transformação endógena. Se são diferentes, a transformação é chamada exógena [18].

- **Espaços Técnológicos Suportados**:

  - Como foi visto, modelos podem estar em espaços tecnológicos diferentes. Os espaços tecnológicos limitam quais ferramentas de transformação podem ser usadas.

- **Número de Modelos Suportados**:

  - O número de modelos de entrada e saída também é uma característica importante de uma transformação. A maioria das transformações envolve dois modelos diferentes, um de entrada e outro de saída. Também é possível ter vários modelos de entrada e saída.
  - Quando o modelo de entrada e saída é o mesmo, a transformação é chamada transformação *in-place*. Neste caso, o modelo de saída é criado modificando algumas partes do modelo de entrada.

- **Tipo do Modelo de Saída Suportado**:

  - Podemos diferenciar a transformação de modelos de acordo com o tipo do modelo de saída. Uma transformação modelo-para-modelo (*model-to-model*, M2M) cria um novo modelo a partir do modelo de entrada. Elementos do modelo de entrada são mapeados em elementos do modelo de saída. Uma transformação modelo-para-texto (*model-to-text*, M2T) cria texto ou código. Logo, os elementos do modelo de entrada são mapeados em fragmentos de texto ou código.

- **Preservação de Propriedades**:

  - A transformação pode ou não preservar algumas propriedades. Há três tipos:
    * Preservação Semântica: os modelos de entrada e saída são os idênticos, mesmo estando em espaços tecnológicos diferentes. As computações podem

ser diferentes, mas os resultados produzidos devem ser iguais [11]. Por exemplo, a refatoração de um modelo.

* Preservação Comportamental: As restrições (implícitas e explícitas) do modelo de entrada são mantidas no modelo de saída [11].
* Preservação de Sintaxe: Os modelos de entrada e saída possuem a mesma sintaxe [11]. Este tipo é comum em uma transformação endógena horizontal.

2.1.3 Classificação para Linguagens de Transformação de Modelos

Biehl [11], baseado nos trabalhos [18] e [28], propôs as seguintes características para as linguagens de transformação de modelos:

- **Paradigma**:
  - Há diferentes paradigmas para as linguagens de transformação de modelos [11] e [18]:
    * Imperativo/Operacional: Especificam o fluxo sequencial e descrevem como a transformação deve ser executada, isto é, a transformação é descrita como uma sequência de ações em uma linguagem imperativa [18], como Java ou C#.
    * Declarativo/Relacional: Elas não explicitam o controle de fluxo. Apenas indicam como a transformação deve ser executada, focando em o quê deve ser mapeado pela transformação. Como descrevem o relacionamento entre os metamodelos de entrada e saída, este relacionamento pode ser bidirecional [11]. Como exemplo, temos a ATL [33].
    * Híbrido: Oferece as abordagens imperativa e declarativa, deixando a escolha para o usuário [11].
    * Transformação de Grafos: *Triple Graph Grammars* (TGG) [54] são uma forma de descrever transformações de grafos. É um subconjunto do tipo

declarativo, onde modelos de entrada e saída são interpretados como grafos e há um terceiro grafo que descreve o mapeamento entre os elementos da entrada e saída.

  - Baseado em Templates: É usada em transformações modelo-para-texto. Para cada elemento do metamodelo de saída, há um fragmento de texto correspondente [18].
  - Manipulação Direta: É a aplicação de linguagem de programação de propósito geral para implementar a transformação [18]. Por exemplo, usar Java para produzir um documento XML.

- **Agendamento das Regras**:
  - O agendamento das regras determina a ordem em que cada regra é a aplicada [18]. Uma linguagem de transformação pode fornecer mecanismos para determinar de forma explícita quando e onde uma determinada regra deve ser aplicada [11].

- **Organização das Regras**:
  - Em uma transformação com muitas regras, pode ser interessante a sua organização, isto é, agrupar estas regras de forma que possam ser compostas e/ou reutilizáveis (modularizadas). Assim, com o reuso é possível combinar regras simples para construir regras mais complexas [11].

- **Rastreabilidade**:
  - A execução de uma transformação pode gerar um registro do mapeamento realizado entre os elementos. Esta funcionalidade pode estar embutida na ferramenta ou implementada como parte da descrição da transformação [11].

- **Direção**:
  - O mapeamento entre os modelos de entrada e saída por ser unidirecional, isto é, as regras de transformação só podem ser aplicadas no modelo de entrada;

ou multidirecional, quando as regras podem ser aplicadas tanto do modelo de entrada para o modelo de saída, como também do modelo de saída para o modelo de entrada. Neste caso, se há apenas um modelo de entrada e outro de saída, a transformação é bidirecional [18].

- **Transformação Incremental**:
  - Uma transformação é incremental quando as mudanças no modelo de entrada se propagam para o modelo de saída. Por outro lado, uma transformação não-incremental recria o modelo de saída [11].
- **Representação da Transformação**:
  - As transformações podem ser representadas como texto ou como um modelo. Se são modelos, elas podem ser manipuladas por transformações de modelos [11].

2.2 Linguagens e Ferramentas para a Transformação de Modelos

A seguir apresentamos a listagem das principais ferramentas e linguagens usadas para a transformação de modelos.

EMF Henshin [9] é uma linguagem que suporta transformações modelo-para-modelo endógenas e exógenas. É baseada no Eclipse Modeling Framework (EMF) [56] e a transformação é representada por Triple Graph Grammars (TGG). Contudo, não há suporte para rastreabilidade, multidirecionalidade ou transformação incremental.

Atlas Transformation Language (ATL) [33] é uma linguagem híbrida (imperativa e declarativa, esta última a preferível) para a transformações modelo-para-modelo. Suporta apenas transformações unidirecionais e não incrementais. É integrada com a IDE Eclipse, oferece suporte dedicado a rastreamento, a ordem de execução das regras é determinada automaticamente na maioria dos casos.

OpenArchitectureWare (OAW) [24] é um conjunto de boas práticas e processos de MDE integrados no Eclipse. Possui uma linguagem imperativa e baseada em templates

chamada Xpand [20] para transformações modelo-para-texto.

Query/View/Transform (QVT) [51, 41] é uma especificação da OMG (Object Managment Group) para uma linguagem capaz de expressar consultas, visões e transformações sobre modelos. É divida em três sub-linguagens que são imperativas e declarativas para transformações modelo-para-modelo:

- QVT Relational é uma linguagem declarativa de alto nível. Suporta a especificação de transformações bidirecionais; modelos de rastreabilidade são criados implicitamente.
- QVT Core é uma linguagem declarativa de baixo nível e serve como base para a QVT Relational.
- QVT Operational é uma linguagem imperativa que estende a QVT Relational. As transformações são unidirecionais e usam modelos de rastreamento implícitos.

SmartQVT [5] é uma implementação de um motor de transformação para a linguagem QVT Operational. É uma linguagem imperativa para transformações modelo-para-modelo para modelos do EMF. Não suporta transformações incrementais nem multidirecionalidade.

Kermeta [45, 52] é uma linguagem de modelagem imperativa e propósito geral capaz de realizar transformações modelo-para-modelo. Não há suporte para multidirecionalidade, rastreabilidade, nem para transformações incrementais.

Epsilon Transformation Language (ETL) [39] é uma linguagem híbrida de transformação modelo-para-modelo, e pode tratar vários modelos de entrada e saída. Oferece a funcionalidade do agendamento da aplicação das regras e as regras podem ser reutilizadas e herdadas.

VIATRA2 [17] é um *framework* com uma linguagem híbrida baseada em transformação de grafos que suporta transformações modelo-para-modelo e modelo-para-texto. Também suporta transformações incrementais.

Janus Transformation Language (JTL) [14] é uma linguagem de transformação declarativa modelo-para-modelo que realiza transformações bidirecionais. Possui suporte a

transformações incrementais e rastreabilidade.

A Transformação como Busca (Transformation as Search, TAS) [38] é uma extensão da Busca de modelos [37] que realiza transformações exógenas modelo-para-modelo usando uma linguagem declarativa. Possui suporte a rastreabilidade, incrementabilidade e transformações bidirecionais.

Nas tabelas 2.1 e 2.2 temos o resumo das características mais importantes de cada linguagem.

Ferramenta	**Tipo do Modelo de Saída**	**Paradigma**
EMF Henshin	Modelo-para-modelo	Transformação de Grafo
OpenArchitectureWare	Modelo-para-texto	Imperativa e Baseada em Templates
ATL	Modelo-para-modelo	Híbrida
QVT	Modelo-para-modelo	Híbrida
SmartQVT	Modelo-para-modelo	Imperativa
Kermeta	Modelo-para-modelo	Imperativa
ETL	Modelo-para-modelo	Híbrida
VIATRA2	Modelo-para-modelo Modelo-para-texto	Híbrida
JTL	Modelo-para-modelo	Declarativa
TAS	Modelo-para-modelo	Declarativa

Tabela 2.1: Comparativo

Ferramenta	**Direcionalidade**	**Rastreabilidade**	**Incremental**
EMF Henshin	Unidirecional	Não	Não
OpenArchitectureWare	Unidirecional	Sim	Não
ATL	Unidirecional	Sim	Não
QVT	Unidirecional e Bidirecional	Sim	Sim
SmartQVT	Unidirecional	Sim	Não
Kermeta	Unidirecional	Não	Não
ETL	Multidirecional	Sim	Não
VIATRA2	Unidirecional	Sim	Sim
JTL	Bidirecional	Sim	Sim
TAS	Bidirecional	Sim	Sim

Tabela 2.2: Comparativo

Podemos ver que a maioria das abordagens são modelo-para-modelo, usam linguagens imperativas e são unidirecionais. Poucas possuem suporte a incrementabilidade. Ainda é preciso evoluir bastante nessa área, já que há a necessidade de transformações bidirecionais e incrementais, porque durante o desenvolvimento os modelos de entrada e saída

são modificados continuadamente [14]. Logo, nenhumas das abordagens atuais sozinhas podem ser utilizadas para todos os tipos de problemas, elas sempre são combinadas entre si para obter o resultado desejado.

Neste trabalho, onde é proposto uma cadeia de transformações para realizar última etapa da cadeia da busca de modelos (e consequentemente da transformação como busca), a ATL foi escolhida como ferramenta auxiliar a definição e execução de transformações entre modelos, devido a sua fácil integração com a IDE do Eclipse e com *framework* de modelagem do Eclipse (Eclipse Modeling Framework, EMF), além de ser possível executá-la diretamente dentro de um código java.

2.3 Eclipse Modeling Framework

O Eclipse Modeling Framework (EMF) [56] é um framework de modelagem e geração de código para construir ferramentas e outros aplicativos baseados em modelo. Este framework permite manipular o metametamodelo Ecore usado nesta dissertação. O EMF consiste de dois frameworks básicos: *Core* e *EMF.Edit*. O Core fornece uma API para a geração e manipulação de modelos, além de fornecer ferramentas para produzir código Java a partir de um modelo. O EMF.Edit estende o Core, adicionando suporte para gerar classes adaptadoras, visualizadores e editores de modelos.

O EMF foi inicialmente uma implementação do Meta-Object Facility (MOF) [49], que é o padrão da Object Management Group (OMG) para o MDE, mas acabou ganhando importância própria, e pode ser visto uma implementação Java de um subconjunto da API MOF [56].

O EMF é fornecido como um plugin para a IDE Eclipse e usa XMI (XML Metadata Interchange) para armazenar (serializar) os modelos.

Ecore é o metametamodelo fornecido pelo EMF. Ele tem quatro tipos de classes como principais elementos [56] (figura 2.8):

- EClass: é usado para modelar as classes. São identificadas por nome e possuem atributos, referências. Também há suporte para herança.

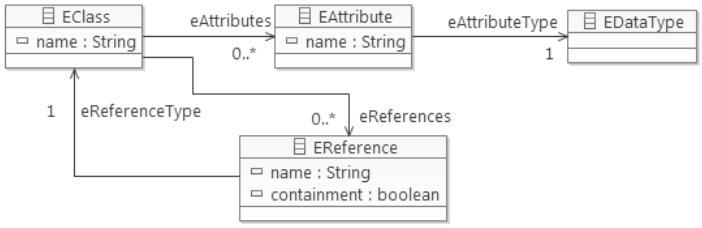


Figura 2.8: Diagrama das principais classes do Ecore [56]

- EAttribute: modela atributos. São identificados por nome e possuem um tipo (um EDataType).

- EDataType: é usado para representar tipos simples, cujos detalhes não são modelados em classes. Também identificados por um nome e são associados a tipos primitivos das linguagens de programação.

- EReference: é usado para modelar associações entre classes. Também são identificadas por um nome e também possuem um tipo (uma EClass). Uma referência também possui multiplicidade.

Estes tipos são usados para modelar as características estruturais dos modelos. Para modelar as características comportamentais, a classe EOperation é usada. No entanto, o Ecore apenas modela a interface destas operações comportamentais, porém é possível especificar implementações em outras linguagens usando anotações [56].

Além de fornecer ferramentas para criar e manipular os modelos Ecore dentro da IDE do Eclipse, é possível fazer a manipulação direta destes modelos utilizando a linguagem Java. Entre as vantagens de realizar esta manipulação em nível de metametamodelo estão: maior flexibilidade, capacidade de manipulação dos modelos e metamodelos em tempo execução e utilização da API reflexiva [56]. Esta API reflexiva permite que programas examinem as informações sobre a estrutura dos modelos e metamodelos que estão sendo manipulados em tempo de execução, com é isso possível criar modelos dinamicamente e criar novas ferramentas.

2.4 Programação por Restrições

Nesta seção apresentaremos os conceitos de programação por restrições usados nesta dissertação.

Programação por restrições (Constraint Programming , CP) [32] é um paradigma de programação declarativo usado para resolver problemas combinatórios [37]. A programação por restrições consiste na especificação de um conjunto de restrições sobre um conjunto de variáveis com um tipo de dado. Em seguida, um programa chamado de resolvedor analisa esta especificação e tenta encontrar uma atribuição de valores válida (valoração) para estas variáveis, satisfazendo as restrições de entrada. O resolvedor pode encontrar zero ou mais valorações (soluções de um problema de programação por restrições).

Cada variável possui um tipo e um domínio, isto é, um conjunto de valores possíveis para uma variável [37]. As restrições sobre um conjunto de variáveis são um subconjunto do produto cartesiano dos domínios destas variáveis [7]. O conjunto de valores de todas as variáveis formam a solução, e este conjunto de valores é encontrado por resolvedores.

Um exemplo de problema resolvido por programação por restrições é determinar se uma fórmula booleana é satisfeita ou não, também conhecido como problema SAT [16].

Uma fórmula booleana é a ligação de um conjunto variáveis com domínio binário (verdadeiro ou falso) por operadores lógicos (conjunção, disjunção e negação) [8]. Por exemplo, tendo as variáveis A e B, uma fórmula para a conjunção da negação destas variáveis pode ser escrita como $\neg A \wedge \neg B$.

A programação por restrições não apenas verifica a satisfabilidade, como também encontra os valores possíveis que tornam a fórmula verdadeira.

O paradigma SAT possui algumas limitações: o domínio das variáveis possui apenas dois valores e fornece uma linguagem de baixo nível para predicados (apenas negação, conjunção e disjunção). Há uma grande variedade de resolvedores SAT eficientes: MiniSat [23], SAT4J [10], Chaff [46] e outros. Estendendo o problema SAT para variáveis com domínios não apenas binários, chegamos aos problemas de satisfação de restrições (Constraint Satisfaction Problems, CSP) [37].

Há diversos resolvedores para CSP, a maioria são baseados em Prolog [57], uma linguagem de programação lógica ou realizam redução ao SAT. A seguir temos uma lista de alguns resolvedores utilizados no contexto do MDE:

- ECLiPSe [8] é uma ferramenta de código aberto baseada em Prolog, logo as restrições são expressas como predicados lógicos. Possui uma API para ser integrada em aplicações java.
- IBM ®ILOG CPLEX CP Optimizer ®[1] é uma ferramenta proprietária para programação por restrições. Possui uma linguagem de modelagem e resolvedores bastante robustos.
- Alloy [30] foi desenvolvido pelo MIT e, apesar de usar resolvedores SAT, possui uma linguagem relacional bastante expressiva. Esta linguagem permite a expressão de predicados usando conjuntos, relações e quantificadores. O Alloy pode utilizar diferentes resolvedores SAT, além de poder ser facilmente integrada em aplicações Java.

Entre estes resolvedores, o ILOG CPLEX CP Optimizer possui a desvantagem de ser uma ferramenta proprietária e paga. O ECLiPSe e o Alloy já foram utilizados no contexto do MDE em [13], [38]. Neste trabalho o Alloy foi escolhido, pois uma implementação atual da modelagem como busca já o utiliza, está disponível em código aberto, tem ferramentas e documentação de suporte, e sua especificação pode ser traduzida para diferentes resolvedores.

2.5 Framework Alloy

O framework Alloy [30] é uma ferramenta de código aberto desenvolvida pelo MIT, e seu propósito é analisar programas escritos pela linguagem de modelagem Alloy utilizando o Alloy Analyzer.

A linguagem Alloy é baseada na lógica de predicados, e pode ser usada para representar modelos e suas restrições [44]. Os programas escritos na linguagem Alloy são modelos for-

mados por estruturas chamadas assinaturas (*signatures*), que são declarações de conjunto de átomos (*atoms*), e fatos (*facts*).

Átomos são entidades primitivas, indivisíveis, imutáveis e não possuem propriedades [31]. A linguagem permite a especificar propriedades através da definição de relações entre átomos [31]. Estas relações são especificadas na linguagem por meio de campos (fields) dentro das assinaturas. Fatos são declarações de restrições sobre átomos ou assinaturas. Esta restrições podem ser predicados, limite de cardinalidade, ou operações sobre conjuntos [37].

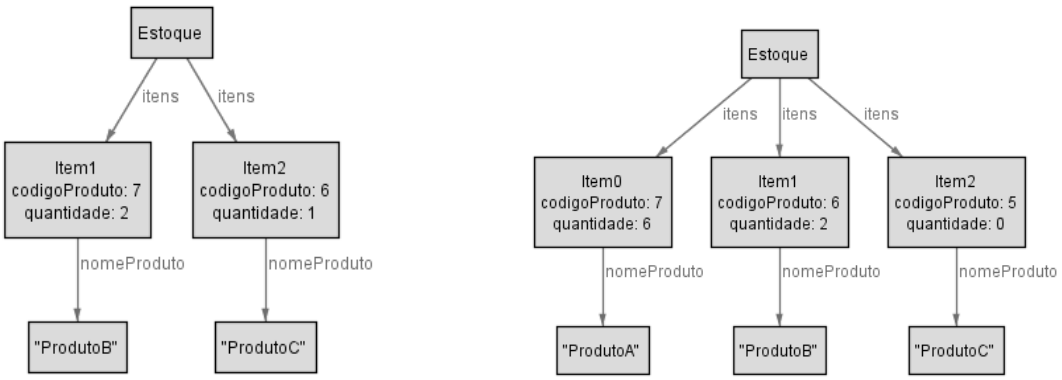
Código 2.1: Exemplo de código Alloy

```
1 sig Estoque {
2         itens: set Item
3 }
4 sig Item {
5         codigoProduto: one Int,
6         nomeProduto: one String,
7         quantidade: one Int
8 }
9 fact { all y: Item | y.quantidade >= 0 }
```

Estes conceitos são exemplificados na listagem de código 2.1 com um sistema de controle de estoque. Um estoque é formado por um conjunto de itens. As entidades são descritas por assinaturas. Cada item é descrito por uma assinatura que contém a relação de três átomos. O primeiro átomo é o código do produto, o segundo é nome do produto e o terceiro é quantidade deste produto no estoque. Na linha 9 é definida uma restrição que exclui os valores negativos do domínio do campo quantidade.

O Alloy Analyzer funciona como um resolvedor de programação por restrições [44]. Ele recebe como entrada um programa Alloy, e produz uma instância. Esta instância é uma valoração das variáveis do programa que satisfazem as restrições, ou seja, é a solução encontrada pelo resolvedor SAT [29]. Se o programa permite mais de uma solução, o Alloy Analyzer é capaz de encontrá-la.

A figura 2.9 ilustra duas instâncias para o exemplo do sistema de controle de estoque. Na figura 2.9(a), temos o átomo 'Estoque' relacionado com dois átomos do tipo Item ('Item1' e 'Item2') através do campo itens. Cada uma destas relações forma uma tupla.



(a) Instância com dois itens (b) Instância com três itens

Figura 2.9: *Exemplos de instâncias Alloy*

Cada átomo do tipo Item possui três campos: código, nome do produto e quantidade. Na figura 2.9(b), temos uma instância similar com três átomos do tipo Item.

Internamente, o Alloy Analyzer traduz o programa Alloy em uma fórmula booleana, e um resolvedor SAT é utilizado para encontrar uma ou mais valorações desta fórmula. Cada valoração é uma solução, que é traduzida automaticamente para a linguagem do programa Alloy. Uma característica do Alloy Analyazer é restringir o espaço da busca através de um escopo, isto é, uma restrição no espaço de busca. Como esse escopo é limitado, o resolvedor nem sempre encontra uma instância, mas isso significa apenas que não foi possível encontrar a solução dentro daquele escopo. Porém, as instâncias encontradas são seguramente válidas [44]. Também é oferecido um suporte a rastreamento, assim é possível realizar análises e simulações podem ser aplicadas as instâncias encontradas [31].

Finalmente, as instâncias encontradas pelo Alloy Analyzer são armazenadas como documentos XML [29]. Para as instâncias serem utilizada no espaço tecnológico MDE, é preciso injetar este documento XML num modelo.

CAPÍTULO 3

MODELAGEM COMO BUSCA

Neste capítulo apresentaremos primeiramente a abordagem chamada "modelagem como busca", que integra técnicas de programação por restrições com execução de operações MDE. Em seguida, apresentaremos uma extensão do MAS (Model as Seach), específica para execução de transformação de modelos. Finalmente, apresentaremos uma implementação que realizamos para avaliar a cadeia de TAS.

3.1 Modelagem como Busca

A Modelagem como Busca (Model Search) [37] é uma abordagem que usa programação por restrições para aplicar operações sobre modelos. A natureza dos resolvedores de programação por restrições permite, dado um conjunto de elementos do modelo não conectados como entrada, encontrar um ou mais modelos de saída que satisfazem a estrutura do metamodelo e um conjunto de restrições. Esta possibilidade de encontrar múltiplas soluções é um diferencial em relação as abordagens de transformação apresentadas na seção anterior.

Os metamodelos podem conter restrições embutidas (metamodelos com restrições). Um metamodelo com restrições é definido por um metamodelo e um conjunto de restrições que um determinado modelo deve satisfazer. Quando este modelo satisfaz a todas as restrições dizemos que está em conformidade com as restrições.

A figura 3.1 ilustra uma operação de modelagem como busca. Esta operação consiste em encontrar um modelo (M) a partir de outro modelo entrada M_r que está parcialmente em conformidade com um metamodelo (CMM). Este modelo M_r é chamado de modelo parcial, e está em conformidade com o relaxamento das restrições do metamodelo CMM. Este relaxamento consiste em remover restrições e elementos (cardinalidade, opcionais, etc) do metamodelo original, criando assim o chamado metamodelo relaxado

CMM_r. Quando este modelo parcial $M_r$está em conformidade com o metamodelo relaxado CMM_r, dizemos que o modelo parcial está em conformidade parcial (p-conformsTo) com o metamodelo com restrições CMM. Finalmente, tendo o modelo parcial, o metamodelo relaxado e o metamodelo com restrições, a modelagem como busca é a operação que procura encontrar um modelo (finito) a partir do modelo parcial que esteja em conformidade com o metamodelo com restrições.

O conceito de metamodelo parcial foi introduzido nesta abordagem para poder definir o conjunto de elementos não conectados que uma operação de modelagem como busca recebe como entrada. Isto se deve ao fato de que esta abordagem precisa de um "pool" de elementos de entrada, isto é, não é uma abordagem generativa.

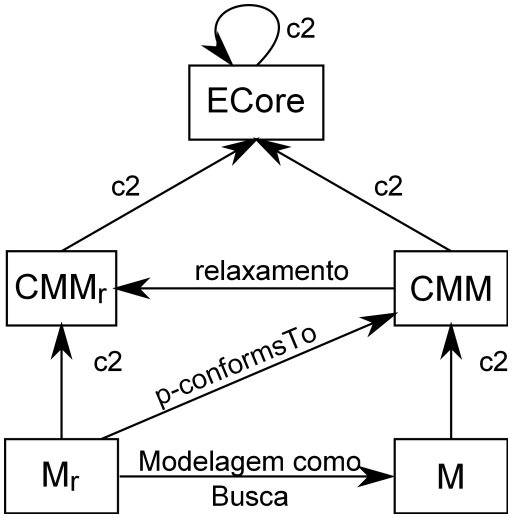


Figura 3.1: Modelagem como Busca [37]

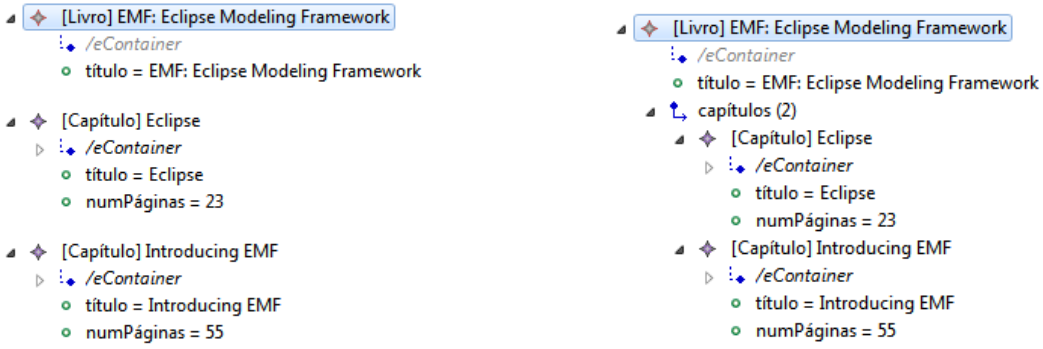
A modelagem como busca é considerada uma transformação de modelos onde o modelo de entrada (modelo parcial M_r e o metamodelo com restrições CMM) é uma instância de um problema combinatório, e o modelo de saída M é uma solução deste problema (se existir). Do ponto de vista da programação por restrições, o metamodelo CMM age como uma fórmula, enquanto que o modelo parcial é uma atribuição parcial que precisa ser completada para a fórmula ser verdadeira [37].

Para exemplificar a modelagem como busca, utilizaremos novamente o metamodelo de Livros [21] (figura 3.2(b)). Um modelo está em conformidade com este metamodelo com restrições quando um Livro possui zero ou mais capítulos. Ao relaxar este meta-

(a) Metamodelo de Livros relaxado

(b) Metamodelo de Livros com restrições

Figura 3.2: *Exemplo do metamodelo Livro*



(a) Modelo parcial

(b) Modelo encontrado

Figura 3.3: *Exemplo de instância de Livro*

modelo, podemos excluir a referência entre Livro e Capítulo, de forma que os elementos do modelo parcial sejam desconexos. Com isso, temos o metamodelo relaxado na figura 3.2(a). O modelo parcial pode ser qualquer modelo que esteja em conformidade com este metamodelo relaxado. Pode ser um modelo vazio ou um modelo que possui apenas uma instância da classe Livro e duas instâncias da classe Capítulo que não estão associadas a nenhuma instância da classe Livro (figura 3.3(a)). Quando um desses modelos parciais é usado como entrada pela modelagem como busca, o resolvedor realiza atribuições que tornem esse modelo parcial em conformidade com as restrições e com o metamodelo não-relaxado [37]. No segundo exemplo de modelo parcial, o resolvedor adiciona as instâncias das classes Capítulo à instância da classe Livro 3.3(b).

Em [37] é apresentado um cadeia para a automatização e integração da modelagem como busca ao espaço tecnológico MDE. A figura 3.4 ilustra essa cadeia e a divide em cinco tarefas. Os metamodelos CMM, CMM_r, *Search Engine*, *SE Solution* e XML estão no

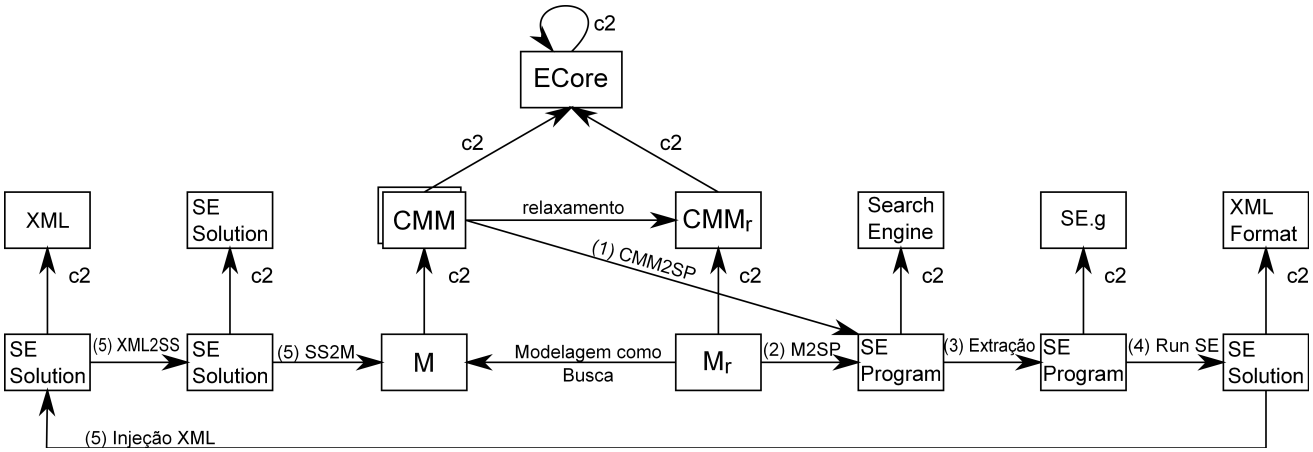


Figura 3.4: Integração da modelagem como busca no espaço tecnológico MDE [37]

espaço tecnológico MDE. Os metamodelos *Search Engine* e *SE Solution* são os metamodelo usados para representar o programa e a solução do resolvedor no espaço tecnológico MDE. O metamodelo XML é o metamodelo para documentos XML dentro do espaço tecnológico MDE. Os outros metamodelos estão fora do espaço tecnológico MDE. SE.g representa a gramática do programa do resolvedor e *XML Format* é o esquema do documento XML onde está a solução do resolvedor. A seguir temos uma descrição de cada etapa da cadeia:

1. Geração do Problema de Busca (CMM2SP): esta tarefa gera o modelo do problema de programação por restrições a partir do metamodelo de entrada e suas restrições (CMM).

2. Geração dos Dados de Busca (M2SP): gera os dados de entrada do programa de programação por restrições a partir do modelo parcial M_r. Esta transformação é específica para cada metamodelo de entrada.

3. Extração do Programa: converte o modelo do problema de programação por restrições para um formato executável pelo resolvedor usado.

4. Busca da Programação por Restrições: o programa gerado nos passos anteriores é executado, e uma ou mais soluções são encontradas no formato da saída do resolvedor.

5. Injeção da Solução (XML2SS, SS2M): Injeta a solução encontrada pelo resolvedor em um modelo em conformidade com o metamodelo CMM. Esta etapa consiste em escrever uma transformação para injetar a solução encontrada no formato da saída do resolvedor em um modelo do espaço tecnológico base.

A modelagem como busca também pode utilizar um metamodelo de saída diferente do de entrada, neste caso é usada para transformações exógenas, e esta abordagem recebe o nome de transformação como busca, que será apresentada a seguir.

3.2 Transformação como Busca

A Transformação como Busca (Transformation As Search, TAS) [38] é uma abordagem, baseada na modelagem como busca, para definir e executar transformações de modelo bidirecionais. A diferença principal em relação a modelagem como busca é a execução de transformações exógenas. Isto impacta na especificação das restrições sobre o metamodelo da solução, pois teremos um metamodelo fonte e um metamodelo destino, e as restrições deverão ser escritas entre os dois metamodelos. Estas restrições podem ser consideradas correspondências entre metamodelos. Como se baseia na modelagem como busca (é independente de resolvedores), não é preciso definir uma linguagem específica, assim qualquer ferramenta de programação por restrições pode ser usada. A abordagem fornece suporte para transformações modelo-para-modelo, verticais e horizontais, endógenas e exógenas, bidirecionais, incrementais, rastreáveis, mas sem suporte a agendamento e controle de aplicação das regras. Estas características são possíveis devido às expressividade das linguagens declarativas usadas pelos resolvedores. Porém algumas delas, como a bidirecionalidade, dependem de como é realizada a implementação da transformação.

A figura 3.5 ilustra o funcionamento da cadeia de transformação como busca, numa transformação exógena unidirecional. M^A é o modelo de entrada que está em conformidade com o metamodelo de entrada CMM^A, M^B é o modelo de saída em conformidade com o metamodelo CMM^B. M_r^T é o modelo parcial para a modelagem como busca e consiste numa cópia do modelo de entrada. M^T é a solução encontrada pela modelagem

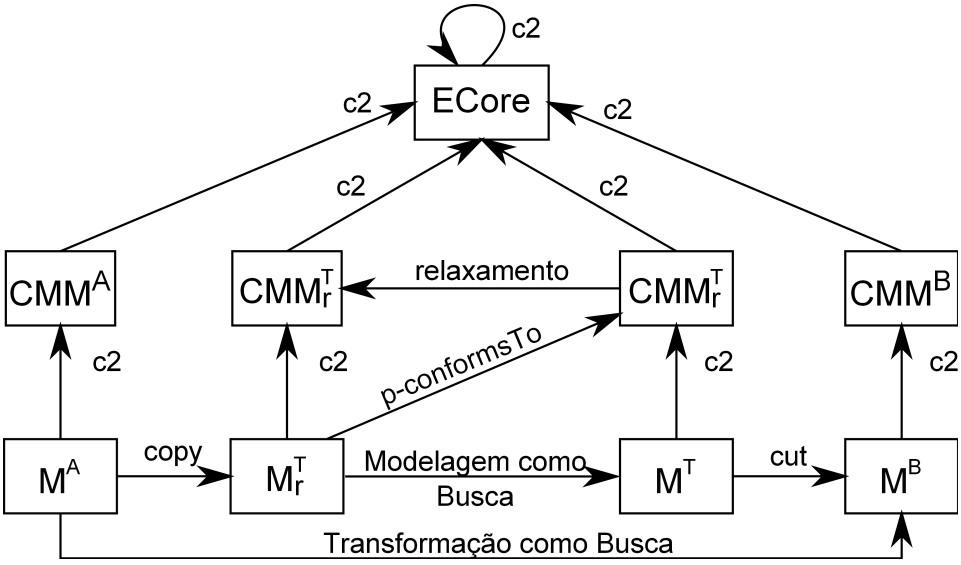


Figura 3.5: Transformação como busca

como busca, CMM^T é metamodelo de transformação (unificação dos metamodelos de origem e saída e seus elos) e CMM_r^T é o metamodelo relaxado de CMM^T. Depois de encontrada a solução pela modelagem como busca, há um último passo chamado 'cut' que extrai o modelo desejado. Este passo é necessário porque o resultado da modelagem como busca possui tanto o modelo de entrada como o de saída.

A seguir, apresentaremos um caso de uso de transformação como busca para aplicação e teste da cadeia completa [1]. Foi implementada a transformação de modelos ClassToRelational [22], que consiste em transformar um modelo de classes em um relacional. Esta transformação e usada em diversos trabalhos como ilustração de problemas de transformações. A figura 3.6 ilustra o metamodelo da transformação usado em [38]. Este metamodelo ilustra a correspondência (weaving links) entre os metamodelos de entrada e saída. Nesta implementação foi utilizado como resolvedor o framework Alloy/SAT.

A tabela 3.1 ilustra o mapeamento entre os elementos dos metamodelos. Não há um mapeamento 1:1 entre o elemento Table do metamodelo relacional e um elemento do metamodelo de classe e sim um mapeamento disjunto entre tabelas, atributos multivalorados e classes. Isto é, há uma tabela para uma classe ou um atributo multivalorado. Isso é realizado pela associação wnamed entre AttributeClassAndTable e NamedElt. Porém, existe uma restrição que especifica que os elementos dessa associação só podem ser classes

[1] Publicado em [38]

ou atributos multivalorados. Há uma equivalência entre atributos e colunas, eles possuem estruturas muito semelhantes, mas possuem algumas propriedades diferentes (como a indicação de chave de uma tabela para o modelo relacional). Um caso diferente são os atributos multivalorados de uma classe, pois eles estão associados a uma tabela e a uma coluna. Já os tipos de dados do metamodelo relacional e de classe possuem a mesma estrutura, então seu mapeamento é direto. Há também casos onde um elemento de um metamodelo não possui correspondência com algum elemento do outro metamodelo. Neste caso, o metamodelo relacional possui chaves, porém o metamodelo de classes não possui esta informação, e não há correspondência e essa informação fica contida apenas no modelo relacional.

Uma implementação da transformação como busca pode ser dividida em dois passos:

1. Criação do metamodelo de transformação: identificar as correspondências, com suas ligações e elementos;

2. Adição de restrições: adicionar restrições que tornem os modelos corretos.

No primeiro passo nem sempre este mapeamento é bijetor (1:1). É preciso identificar os casos onde há uma relação 1:N, N:1 e M:N entre os elementos dos metamodelos. No segundo passo é preciso adicionar as restrições que tornem corretos o mapeamento e os modelos de entrada e saída que serão produzidos. Estes dois passos correspondem a unificação dos metamodelos da transformação como busca. No exemplo ClassToRelational, os nomes de atributos, classes, tabelas, colunas, tipos devem se corresponder de forma adequada.

Elementos do Metamodelo de Classe	**Elementos do Metamodelo Relacional**
Class	Table
Attribute (monovalorado)	Column
Attribute (multivalorado)	Column e Table
DataType	Type

Tabela 3.1: Mapeamentos entre os elementos

A seguir temos a listagem de algumas restrições da transformação usando a OCL+ [3]

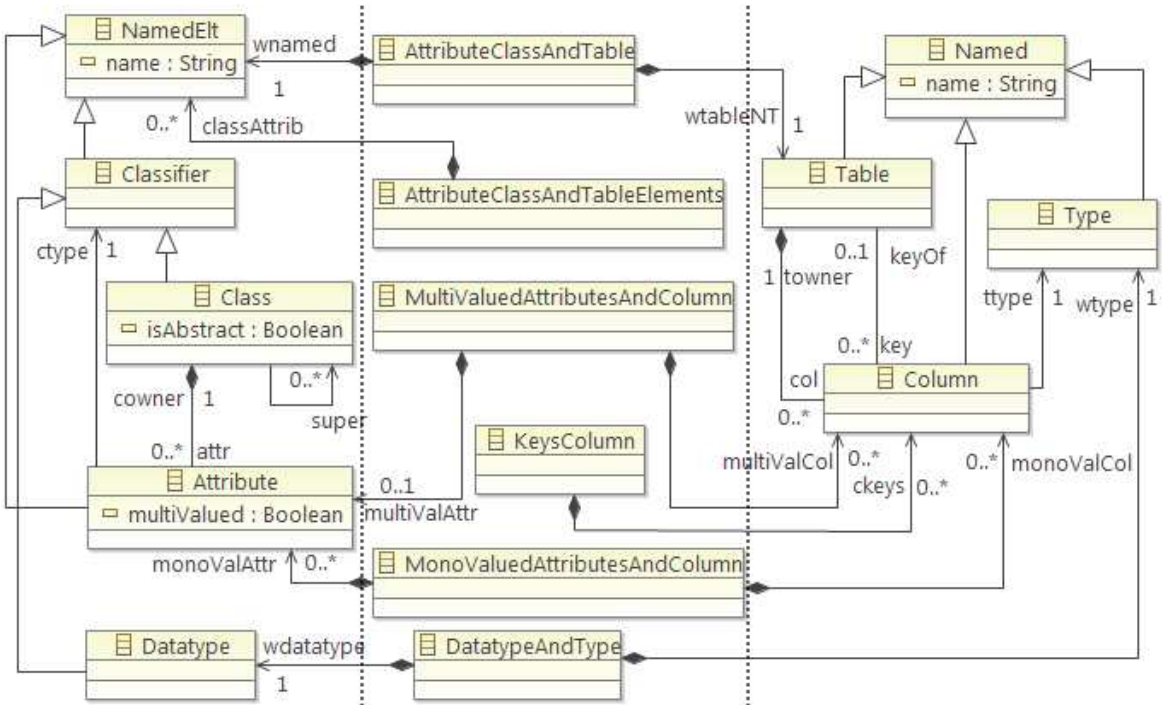


Figura 3.6: Metamodelo de transformacão para a transformacão como busca [38]

(uma extensão da linguagem OCL [47]). Estas restrições estabelecem a correspondência entre os nomes de tipos, classes, atributos multivalorados e tabelas.

A injeção destes metamodelos e suas restrições num programa de busca representam o final da tarefa Extração do Programa. Nesse ponto, o resolvedor é executado (isto é, é executada a busca da programação por restrições), e uma solução é encontrada.

A figura 3.7 apresenta a entrada e o resultado da transformação de um modelo de classe para o modelo relacional. O modelo utilizado consiste em duas classes (Family e Person) com atributos monovalorados e mutivalorados. Os atributos multivalorados e as classes foram mapeados corretamente para tabelas. Da mesma forma, os atributos monovalorados foram mapeados corretamente.

Como podemos ver na tabela 3.2, o um ponto negativo da implementação foi o tempo de execução. Os modelos utilizados possuem poucos elementos (apenas duas classes e cinco atributos). Foi realizado um experimento com uma entrada maior e extraída de um sistema real (Sistema de Apoio da Pós-graduacão - SAPos) [4] (figura 3.8). Esta entrada possui 18 classes. Os experimentos mostraram que para essa entrada o framework Alloy traduziu o problema para um problema booleano com uma explosão de variáveis. O

Restrições OCL+

```
1- context dtt: DataTypeAndType inv:
        dtt.wdatatype.name = dtt.wtype.name
2- context act: AttributeClassAndTable inv:
        act.wnamed.name = act.wtableNT.name
3- context act: AttributeClassAndTable inv:
     act.wnamed.includes(named: NamedElt |
     named.oclIsTypeOf(Class) or
     (named.oclIsTypeOf(Attribute) and named.multiValued)))
4- context mvac : MonoValuedAttributesAndColumn inv:
      mvac.monoValAttr.forAll (attr: Attribute |
        not attr.multiValued  iff
        mvac.monoValAttr.including(attr) and
        mvac.monoValCol.one( col | col.name = attr.name and
                col.keyOf.oclIsUndefined() and
                col.towner.name = attr.cowner.name and
                if DataType.one( dt | attr.ctype.name = dt.name) then
                        col.ttype.name = attr.ctype.name
                else
                        col.ttype.name = 'Integer')
```

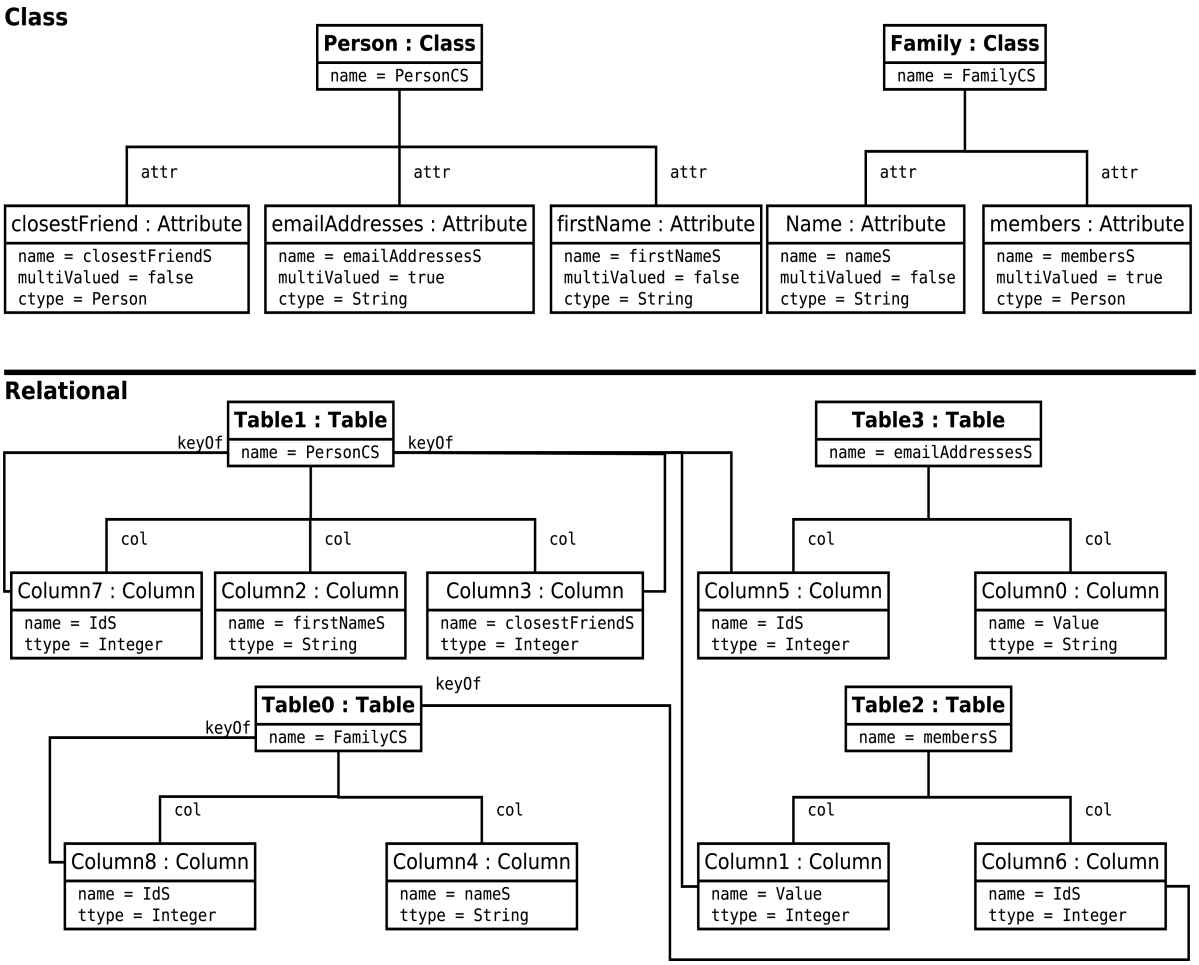


Figura 3.7: Resultado da transformação do diagrama de classe para relacional

Cenário-versão	Número de Variáveis Booleanas	Tempo de Execução (ms)
Classe para Relacional	7179	2483
Relacional para Classe	5725	1655
Incremental	5448	666

Tabela 3.2: Tempos de Execução

resolvedor minisat foi utilizado nestes experimentos.

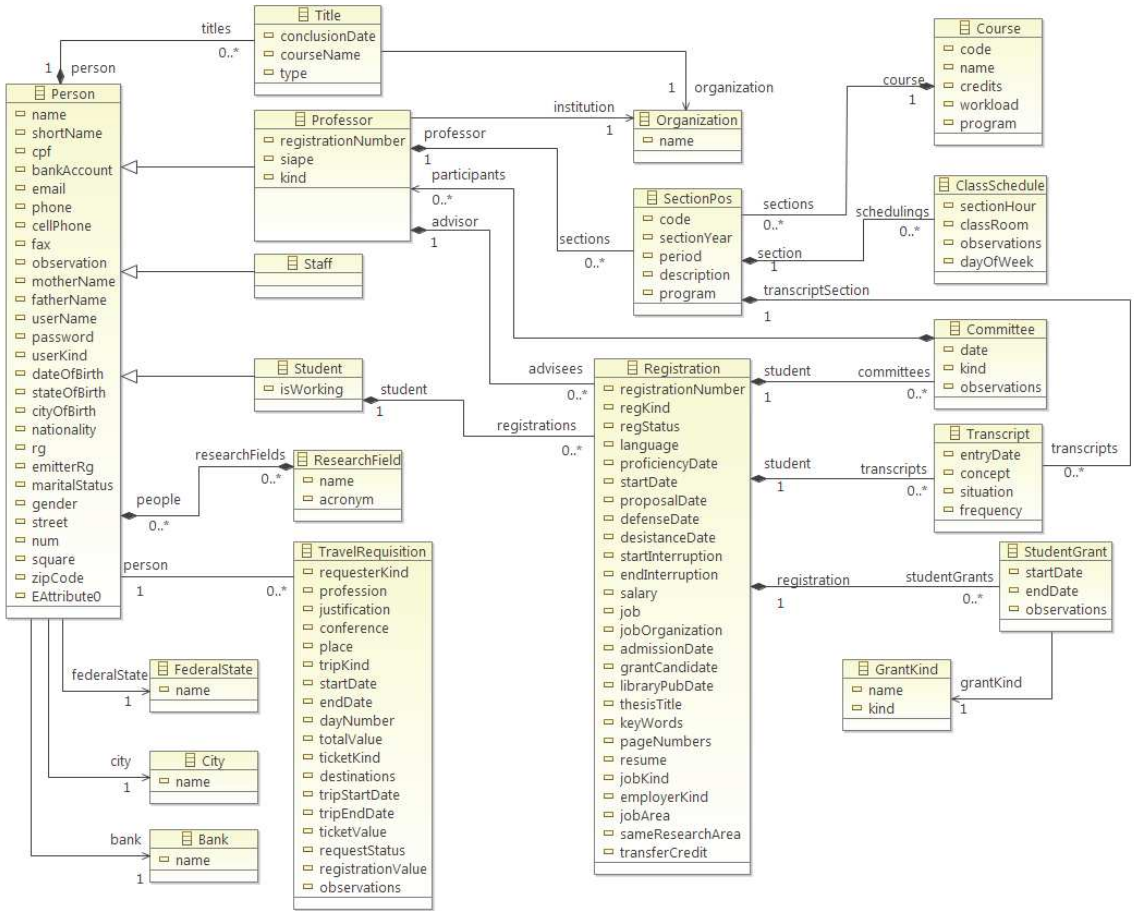


Figura 3.8: Diagrama com as classes do SAPos

Após esta etapa, a solução encontrada deve ser injetada no espaço tecnológico MDE. Numa abordagem clássica é preciso escrever uma transformação de modelos T_1 que injete a solução encontrada pelo resolvedor em um modelo em conformidade com o metamodelo relacional. Porém esta transformação T_1 não serve para injetar a solução num modelo em conformidade com o metamodelo de Classe ou com o metamodelo de Livros, por exemplo. Seria preciso escrever transformação T_2 para o metamodelo de Classe e uma transformação T_3 para o metamodelo de livros. Neste caso, não é desejável escrever uma injeção diferente para cada metamodelo de saída diferente. Automatizar esta injeção para

metamodelos de saída diferentes traria benefícios para o uso da modelagem como busca e transformação como busca.

É preciso de uma implementação genérica para a injeção da solução da cadeia da modelagem como busca. As soluções encontradas pelo resolvedor estão em formato XML. A injeção da solução é crucial para a praticidade da modelagem como busca, e consequentemente da transformação como busca.

3.3 Trabalhos Relacionados

Em [27] os autores apresentam uma ferramenta para verificação de modelos UML com restrições especificadas com a linguagem OCL [47]. Há suporte para os diagramas de classe, objeto e sequência da UML, mas é preciso escrever um código para realizar a validação. Este código consiste em consultas que são executadas por um resolvedor OCL. Este resolvedor é usado como um biblioteca que retorna o resultado da consulta em formato de texto, que é exibido na interface do programa. A injeção da solução do resolvedor desta abordagem é sempre para um formato já conhecido.

A abordagem [13] tem o mesmo objetivo, mas com um escopo mais reduzido (apenas diagramas de classe). Fornece uma verificação automática. Neste caso o ECLIPSe é utilizado como resolvedor. Esta abordagem transforma o modelo UML de entrada e suas restrições em um programa de busca, e o resolvedor tenta encontrar uma atribuição (ou mais de uma) que satisfaça todas as restrições. Se todas as restrições são satisfeitas, um modelo que representa uma instância válida daquele diagrama de classe é encontrado. Se este modelo não é encontrado, significa que a especificação do modelo de entrada e restrições está errada. A solução encontrada pelo resolvedor está em um grafo armazenado em formato textual. A ferramenta GraphViz [25] é utilizada para gerar apenas uma visualização desta solução.

Em nenhuma dessas duas abordagens, o objetivo é produzir um modelo usável a partir da solução do resolvedor. Os modelos encontrados estão em formato próprio, e não há uma transformação deles em modelo Ecore, por exemplo.

Em [14] é apresentado uma abordagem chamada Janus Transformation Languagem

(JTL), que possui funcionamento similar à transformação como busca. É uma linguagem de transformação de modelos baseado em resolvedores de programação por restrições. Porém, o modelo e metamodelos de entrada precisam ser codificados usando uma linguagem própria da ferramenta, porque não há uma extração automática dos metamodelos ECore para a linguagem usada pelo motor de transformação. Nesta abordagem a injeção da solução é realizada por uma transformação que recebe o metamodelo de saída e gera uma outra transformação que transforma a saída do resolvedor em um modelo em conformidade com o metamodelo de saída.

A modelagem como busca tem como objetivo realizar estas extrações e injeções de forma automática e transparente. Por exemplo no caso de linhas de produto de software, o metamodelo, restrições e modelo parcial estão no espaço tecnológico base. Estes modelos e restrições são extraídos para o espaço tecnológico do resolvedor de forma automática e a solução do resolvedor é injetada no espaço tecnológico base também de forma automática em conformidade com o metamodelo de entrada.

No próximo capítulo é apresentada a abordagem utilizada para extrair a solução encontrada pelo resolvedor e injetá-la como um modelo usável no espaço tecnológico MDE escrevendo apenas uma única transformação.

CAPÍTULO 4

TRANSFORMAÇÃO DE MODELOS INDEPENDENTE DE METAMODELOS USANDO REFLEXÃO

Neste capítulo, apresentaremos uma abordagem para a injeção da solução produzida pela operação de modelagemo como busca em um modelo no espaço tecnológico MDE, usando apenas uma transformação para todos os metamodelos de saída. Esta abordagem é composta por quatro transformações. As três primeiras operações são relativas a interoperabilidade entre espaços tecnológicos. A quarta transformação da abordagem é o ponto chave, denominada Transformação de Modelos Independente de Metamodelos usando Reflexão (TIMeR). Seu objetivo é implementar uma transformação que seja independente do metamodelo de saída, isto é, um modelo de entrada é transformado em um modelo de saída em conformidade com um metamodelo que é desconhecido na fase de implementação.

Primeiramente será apresentada uma abordagem independente de resolvedor em seguida sua instanciação com o framework Alloy.

4.1 Abordagem Independente de Resolvedor

Como foi apresentado, a primeira etapa da cadeia da modelagem como busca (CMM2SP) é transformar o metamodelo e suas restrições em um programa de busca, e a última etapa (SS2M) é transformar o modelo da solução encontrada pelo resolvedor em um modelo em conformidade com o metamodelo CMM. Portanto, para realizar a transformação do modelo da solução, precisamos de duas etapas (figura 4.1):

1. Consiste no passo reverso da CMM2SP, isto é, o metamodelo CMM é extraído do programa de busca em um metamodelo no espaço tecnológico MDE (transformação SP2MM).

2. Consiste na execução da TIMeR, que recebe como entrada o metamodelo CMM produzido em (1) e o modelo com a solução do resolvedor.

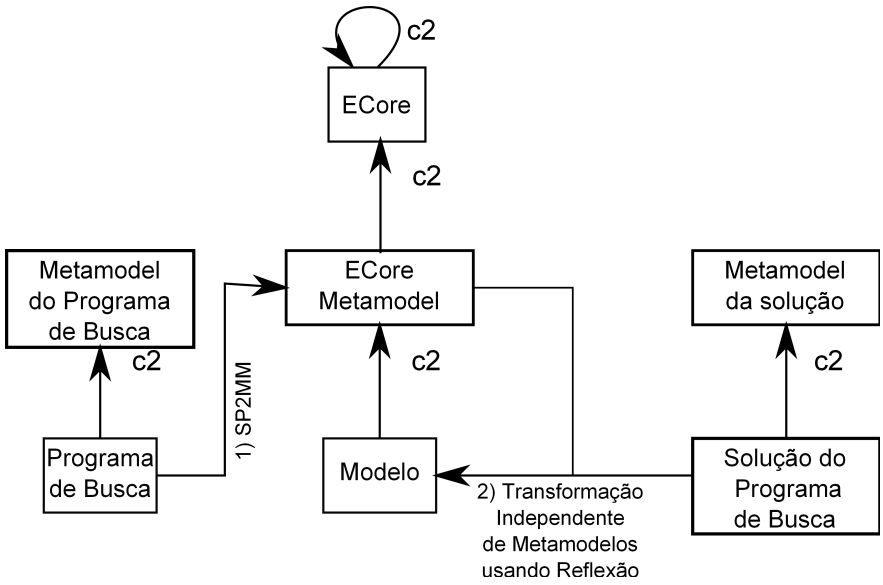


Figura 4.1: Abordagem Independente de Resolvedor

A transformação SP2MM pode ser implementada com qualquer linguagem de transformação. Na transformação SP2MM as restrições não precisam ser transformadas, basta os elementos do metamodelo, em seguida podemos apenas importar as restrições do problema original. As soluções encontradas pelo resolvedor já satisfazem as restrições originais.

O programa de busca, que contém o modelo parcial, restrições e metamodelo de saída, está em outro espaço tecnológico, geralmente em formato de texto da linguagem do resolvedor. Por isso, dependendo do resolvedor utilizado é preciso realizar uma injeção prévia.

A TIMeR possui duas entradas: a solução do resolvedor e o metamodelo de saída. Cada elemento da solução do resolvedor é processado, e então é criado um elemento no modelo de saída, se houver uma correspondência entre o elemento do metamodelo de saída e o elemento do programa de busca.

A seguir temos a definição da Transformação de Modelos Independente de Metamodelos usando Reflexão:

Definição 4.1 *Definição da Transformação de Modelos Independente de Metamodelos usando Reflexão:*

Seja, $M_I =< G_I, MM_I, \mu_I >$, $MM_I =< G_a, MMM, \mu_a >$ *e* $MM_O =< G_b, MMM, \mu_b >$, *onde* M_I *é o modelo de entrada a ser transformado.* M_I *está conforme* MM_I, *e* MM_O *é o metamodelo de saída. A Transformação de Modelos Independente de Metamodelos usando Reflexão cria um modelo* $M_O =< G_O, MM_O, \mu_O >$, *tal que* M_O *está conforme* MM_O *e* $\exists x \exists y | ((x \in G_I \wedge y \in G_b \wedge y := \mu_I(x)) \leftrightarrow (\exists z | z \in M_O \wedge \mu_O(z) := \mu_I(x)))$

Onde $:=$ *é a operação reflexiva que retorna verdadeiro se os termos a direita e esquerda são correspondentes (na maioria dos casos, se possuem o mesmo nome).*

Analisando o último termo da definição, podemos ver que um elemento do modelo de saída (z) corresponde a um elemento do modelo de entrada (x), se e somente se, o elemento do metamodelo de entrada (que está atrelado a x) possui alguma correspondência com o metamodelo de saída.

Um fator importante é a definição da operação ":=". Nos exemplos testados considerou-se que se os elementos possuem o mesmo nome (label), então eles são similares. Outras definições podem aumentar a expressividade desta operação, mas ao custo de aumentar sua complexidade de projeto e implementação.

A seguir veremos mais detalhes de como isso é feito na implementação da Injeção da Solução usando o framework Alloy.

4.2 Instanciação com o Framework Alloy

Como o programa Alloy gerado e suas respectivas soluções estão em outro espaço tecnológico, é preciso injetá-los no espaço tecnológico MDE, para que sejam compatíveis com o ECore. Para realizar isso, temos duas transformações (injeções) adicionais às apresentadas na seção anterior. Para evitar a reimplementação de soluções já existentes explicitamos quais passos desta cadeia de transformações são reutilizados ou então implementados por este trabalho.

Como podemos ver na figura 4.2 ao todo temos quatro transformações:

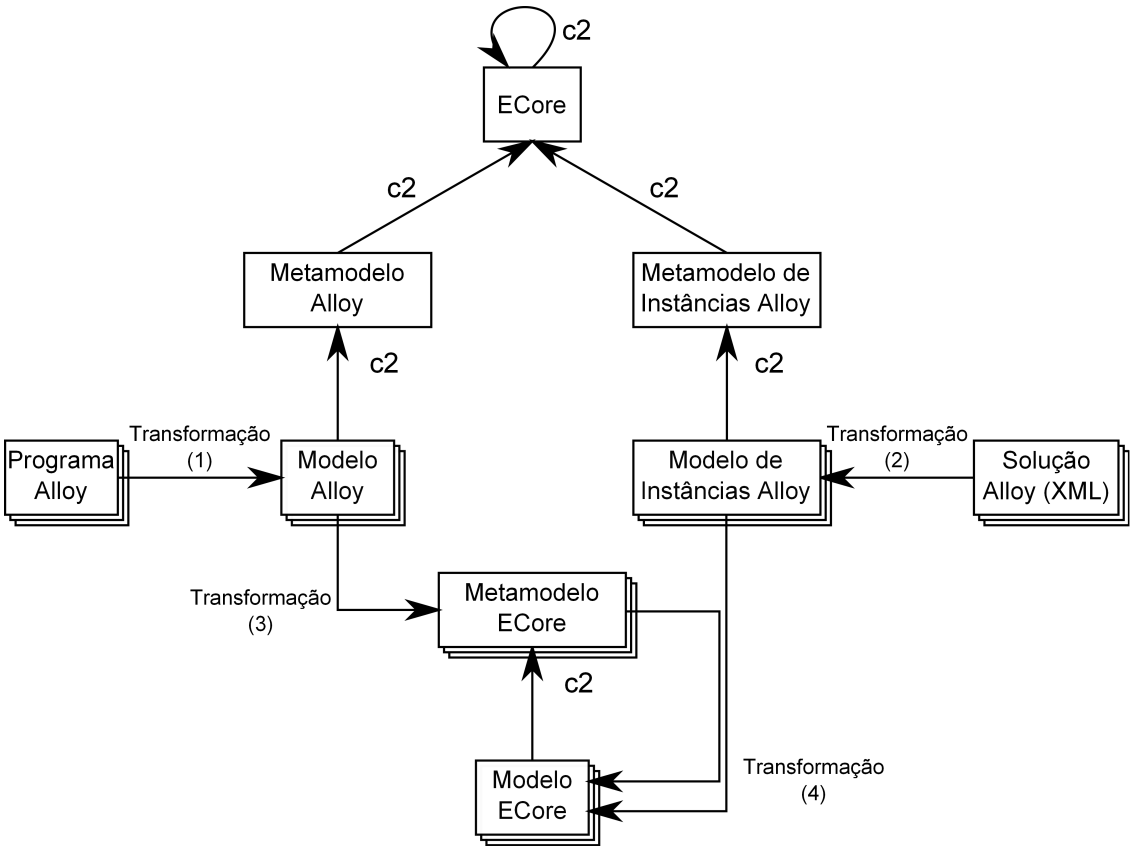


Figura 4.2: Transformação da Solução

1. Injeção do Programa Alloy em um Modelo Alloy, implementado em [37];

2. Injeção da Instância Alloy em um Modelo de Instâncias Alloy;

3. Transformação do Modelo de Programa Alloy em um metamodelo ECore;

4. Transformação de Modelos Independente de Metamodelos usando Reflexão.

Também se pode notar pelos retângulos triplos que podemos ter diferentes modelos e metamodelos de entrada mantendo as mesmas transformações.

A primeira transformação é injeção do programa Alloy em um modelo de programa Alloy. Nesta etapa foi utilizado o metamodelo de programa Alloy (figura 4.3) e o TCS Parser Generator criados em [37], que permite a injeção/extração entre a versão textual e o modelo de programa Alloy. Este TCS foi disponibilizado publicamente como um *plugin* do eclipse em [36].

Para ilustrar o funcionamento de cada etapa, foi escolhido o estudo de caso de linhas de produto de software para a modelagem como busca utilizado em [37]. Este estudo de caso

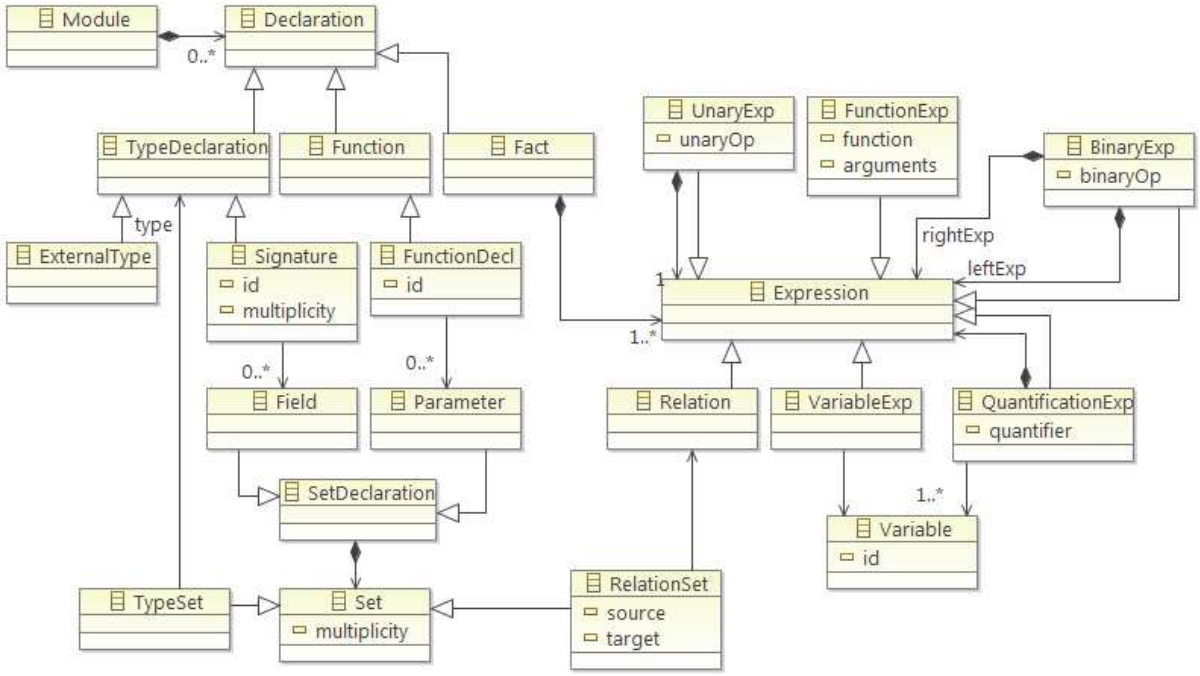


Figura 4.3: Metamodelo Alloy [37]

ilustra os possíveis métodos que uma classe que simula o funcionamento de um relógio deve possuir. Esta classe pode possuir os métodos: Start, Stop, DisplayTime, StartAlarm, StartVisualAlarm, StopAlarm, StartSoundAlarm. As restrições limitam quais combinações de métodos são possíveis. Por exemplo, um relógio deve possuir os métodos Start e DisplayTime e se existir o método StartAlarm, o método StopAlarm também deve estar presente.

Abaixo, temos um programa Alloy produzindo pela terceira etapa da cadeia da modelagem como busca. Este programa Alloy contém o metamodelo de saída, restrições e o modelo parcial, e é usado como entrada na Transformação 1. Esta transformação injeta esse código num modelo de programa Alloy em conformidade com o metamodelo da figura 4.3.

Código 4.1: Programa Alloy - Entrada para a Transformação 1

```
1 module watch
2 // Metamodelo
3 sig Root {
4         classifiers : set Class
5 }
6 abstract sig Class {
7         methods : set Method
```

```
 8 }
 9 abstract sig Method {
10 }
11 // Modelo parcial
12 sig Watch extends Class {}
13 one sig Start extends Method {}
14 one sig Stop extends Method {}
15 one sig StartAlarm extends Method {}
16 one sig DisplayTime extends Method {}
17 one sig StartVisualAlarm extends Method {}
18 one sig StopAlarm extends Method {}
19 one sig StartSoundAlarm extends Method {}
20 // Restrições
21 fact { one Root, all r: Root | #r.classifiers = 1 }
22 fact { all c : Class | c in Watch }
23 fact { all c : Class | one m : Start | m in c.methods }
24 fact { all c : Class | one m : DisplayTime | m in c.methods }
25 fact { all c : Class | Start in c.methods => Stop in c.methods }
26 fact { all c : Class | StartAlarm in c.methods <=> StopAlarm in c.↩
      methods }
27 fact { all c : Class | StartSoundAlarm in c.methods => StartAlarm in c↩
      .methods }
28 fact { all c : Class | StartVisualAlarm in c.methods => StartAlarm in ↩
      c.methods }
```

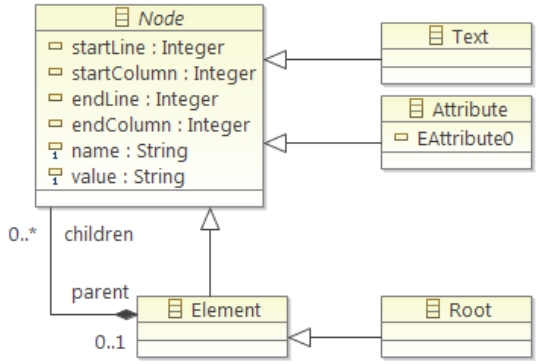


Figura 4.4: Metamodelo XML

O segundo passo é injetar a solução encontrada pelo framework Alloy em um modelo de instâncias Alloy. Como esta solução é um documento XML, este passo é subdivido em duas etapas:

1. Injeção em um modelo XML, usando um metamodelo XML já existente em [22] (Figura 4.4). Esta injeção é feita através do injetor XML do AtlanMod MegaModel Management (AM3) [6].

2. Posteriormente, este modelo XML é transformado usando ATL para um modelo em conformidade com o metamodelo de instâncias Alloy (figura 4.5). O metamodelo

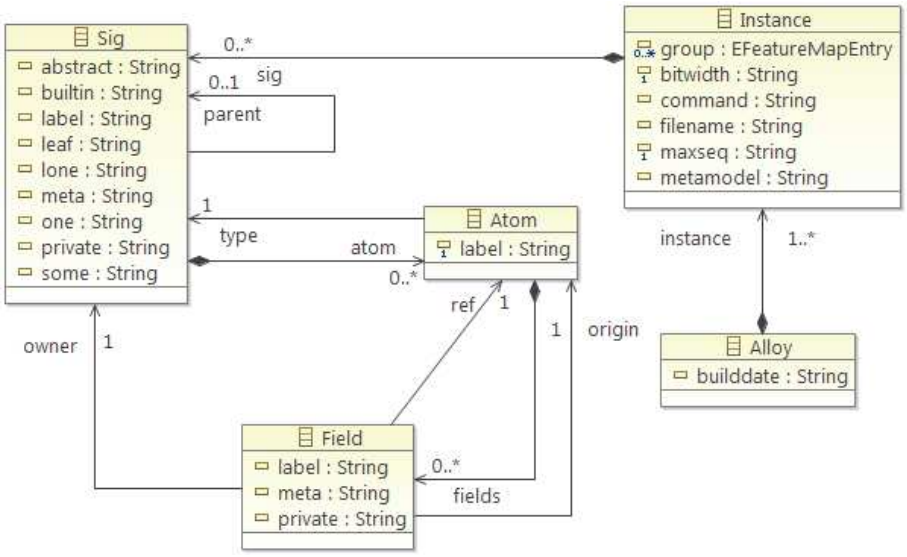


Figura 4.5: Metamodelo de Instâncias Alloy

de instâncias Alloy foi criado neste trabalho baseado nas especificações do esquema XSD do XML da instância do Alloy, o que permite capturar as soluções.

Esta transformação foi implementada usando ATL e associa cada elemento do documento XML com seu respectivo elemento do metamodelo de instâncias Alloy. A seguir temos um exemplo de instância (solução) gerada pelo Alloy que é utilizada como entrada desta transformação. Esta instância contém um relógio com três métodos: Start, Stop e DisplayTime (linhas 19-21).

Código 4.2: Solução do Resolvedor - Entrada para a Transformação 2

```
1  <alloy builddate="2012-09-25 15:54 EDT">
2  <instance bitwidth="4" maxseq="4" command="Run example for 1 Root, 1 ↩
       Class, 7 Method" filename="C:\Untitled 1.als">
3  <sig label="seq/Int" ID="0" parentID="1" builtin="yes"></sig>
4  <sig label="Int" ID="1" parentID="2" builtin="yes"></sig>
5  <sig label="String" ID="3" parentID="2" builtin="yes"></sig>
6  <sig label="this/Root" ID="4" parentID="2">
7     <atom label="Root$0"/>
8  </sig>
9  <field label="classifiers" ID="5" parentID="4">
10    <tuple> <atom label="Root$0"/> <atom label="Watch$0"/> </tuple>
11    <types> <type ID="4"/> <type ID="6"/> </types>
12 </field>
13 <sig label="this/Watch" ID="7" parentID="6">
14    <atom label="Watch$0"/>
15 </sig>
16 <sig label="this/Class" ID="6" parentID="2" abstract="yes">
17 </sig>
18 <field label="methods" ID="8" parentID="6">
```

```
19     <tuple> <atom label="Watch$0"/> <atom label="Start$0"/> </tuple>
20     <tuple> <atom label="Watch$0"/> <atom label="Stop$0"/> </tuple>
21     <tuple> <atom label="Watch$0"/> <atom label="DisplayTime$0"/> </↩
          tuple>
22     <types> <type ID="6"/> <type ID="9"/> </types>
23  </field>
24  <sig label="this/Start" ID="10" parentID="9" one="yes">
25     <atom label="Start$0"/>
26  </sig>
27  <sig label="this/Stop" ID="11" parentID="9" one="yes">
28     <atom label="Stop$0"/>
29  </sig>
30  <sig label="this/StartAlarm" ID="12" parentID="9" one="yes">
31     <atom label="StartAlarm$0"/>
32  </sig>
33  <sig label="this/DisplayTime" ID="13" parentID="9" one="yes">
34     <atom label="DisplayTime$0"/>
35  </sig>
36  <sig label="this/StartVisualAlarm" ID="14" parentID="9" one="yes">
37     <atom label="StartVisualAlarm$0"/>
38  </sig>
39  <sig label="this/StopAlarm" ID="15" parentID="9" one="yes">
40     <atom label="StopAlarm$0"/>
41  </sig>
42  <sig label="this/StartSoundAlarm" ID="16" parentID="9" one="yes">
43     <atom label="StartSoundAlarm$0"/>
44  </sig>
45  <sig label="this/Method" ID="9" parentID="2" abstract="yes">
46  </sig>
47  <sig label="univ" ID="2" builtin="yes">
48  </sig>
49  </instance>
50  </alloy>$
```

A terceira transformação utiliza como entrada o resultado da transformação 1 e cria um metamodelo ECore. Os mapeamentos desta transformação estão resumidos na tabela 4.1. Basicamente as assinaturas (signatures) do programa Alloy são mapeadas em EClass do ECore, e os campos (fields) são mapeados em EReference (quando uma assinatura faz referência a outra assinatura) ou EAttribute, quando as assinatura possui campos de tipos primitivos. As cardinalidades foram implementadas de acordo com a referência da linguagem Alloy [44].

Na figura 4.6 temos o resultado dessa transformação para o exemplo. Comparando com a entrada da Transformação 1 (Código 4.1), podemos ver que o resultado é correto e o eclipse verifica corretamente que o modelo é bem formado.

Finalmente temos o quarto passo: a Transformação de Modelos Independente de Metamodelos usando Reflexão.

Elementos do Metamodelo de Alloy	Elementos do ECore
Module	EPackage
Signatures	EClass
Field	EReference
Field	EAttribute
ExternalType	EDataType

Tabela 4.1: Mapeamento entre o metamodelo Alloy e ECore

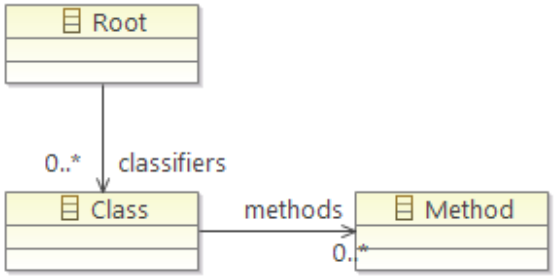


Figura 4.6: Metamodelo ECore produzido pela transformação 3.

4.3 Transformação de Modelos Independente de Metamodelos usando Reflexão

Nesta etapa os resultados das transformações 2 e 3 (modelo com a solução do resolvedor e o metamodelo de saída) são utilizados como entrada para produzir um modelo que está em conformidade com o metamodelo de saída. Esta transformação foi implementada em java devido a sua fácil integração com o framework Alloy, e utiliza a API do EMF. Em especial os métodos de reflexão, que permite conhecer e manipular os elementos dos modelos carregados em tempo de execução. Esta transformação (para o caso do framework Alloy) é genérica e produz um modelo para qualquer metamodelo de saída, pois utiliza a API reflexiva do EMF para obter as informações necessárias para criar o modelo de saída. A Tabela 4.2 lista as principais classes e métodos reflexivos utilizados na implementação.

Classes	Métodos
EObject, EStructuralFeature, EClass, EReference	getName(), eGet(), getEType()

Tabela 4.2: Principais classes e métodos do ECore usados na implementação

Esta transformação se baseia no fato de que os metamodelos de entrada e saída pos-

suem alguma relação entre eles, determinada pelo operador :=. A relação entre modelo e metamodelo permite identificar quais elementos do modelo estão ligados aos elementos do metamodelo de saída. Com isso, o metamodelo de instâncias Alloy serve como uma ponte entre as soluções encontradas e o metamodelo de saída gerado a partir do programa Alloy que gerou essa solução. As soluções são apenas átomos (atoms) (gerados a partir de assinaturas) e as relações entre eles (campos). Dada a definição de modelo $M =< G, MM, \mu >$, os átomos são vértices do grafo G do modelo, os campos representam as arestas deste grafo. A relação entre átomo e assinatura é dada pela função μ (isto é, se A é um átomo e S é uma assinatura $\mu(A) = S$), esta relação é dada pela capacidade de rastreamento fornecida pelo resolvedor e pela reflexão. Por exemplo, na figura 4.5 temos as classes Atom e Sig, que representam átomos e assinaturas, respectivamente. A associação 'type' entre Atom e Sig fornece a informação de restreamento, indicado a qual assinatura (classe) pertence este átomo (instância da classe). Neste caso a função μ é definida como $\mu(A) = A.type$. Com isso, podemos criar uma relação entre átomos e campos para obter qualquer modelo de saída desejado.

Note que para outros resolvedores, basta usar esta informação de rastreamento para auxiliar a definição da ligação dos elementos da solução com as instâncias dos elementos do metamodelo do problema (isto é, o operador :=). Portanto a rastreabilidade da solução do resolvedor é um dos requisitos desta abordagem.

O apêndice A lista o código java utilizado nessa transformação, e a seguir (algoritmo 4.1)temos o pseudocódigo do núcleo dessa transformação:

Os atributos label, type, owner, ref, origin estão definidos no metamodelo de instâncias da figura 4.5, e correspondem, respectivamente, a: nome do elemento, assinatura a qual o átomo pertence, assinatura a qual o campo pertence, átomo de destino do campo e átomo de origem do campo.

A linha 1 cria um modelo de saída vazio em conformidade com o metamodelo de saída apresentado à TIMeR.

As linhas 2-3 indicam que para cada átomo da solução é criada uma instância da classe que possui o nome A.type.label. A.type representa a assinatura de origem deste

Algoritmo 4.1 Pseudocódigo da TIMeR

TIMeR (Metamodelo De Saída MM, Modelo de Instâncias Alloy MI)

```
1: ModeloSaída = new ModeloComforme(MM);
2: for each Átomo A in MI do
3:    C' = criaInstânciaDaClasse(pegaClassePeloNome(A.type.label));
4:    ModeloSaida.AdicionaObjeto(C');
5: end for each
6: for each Campo C in MI do
7:    R = pegaReferênciaPeloNome(C.label);
8:    if R != nulo then
9:       ModeloSaída.pegaObjetoPeloNome(C.origin.label).criaInstânciaDaReferência(R,
         C.ref.label);
10:   end if
11:   T = pegaAtributoPeloNome(C.label);
12:   if T != nulo then
13:      ModeloSaída.pegaObjetoPeloNome(C.origin.label).criaInstânciaDoAtributo(T,
         C.ref);
14:   end if
15: end for each
16: return ModeloSaída
```

átomo, é uma informação de rastreamento da solução do resolvedor. Esta assinatura está atrelada a um elemento do metamodelo de saída porque, a função ":=" foi definida como a igualdade do atributo label, e este atributo é obtido através da API de reflexão do EMF. A linha 4 adiciona esta instância ao modelo.

Tendo criados todas as instâncias necessárias, basta agora adicionar atributos e referências entre as classes. Na linha 5 cada campo da solução e iterado e é realizada a criação dos atributos e referências necessários.

Nas linhas 7-8, a função pegaReferênciaPeloNome busca no conjunto de referências do metamodelo se existe uma que possui o nome C.label. Caso afirmativo, na linha 9, esta referência é adicionada ao objeto da solução que possui o nome C.origin.label para o objeto C.ref.label.

Nas linhas 11-12, a função pegaAtributoPeloNome, de modo similar, retorna o atributo do metamodelo que possui o nome C.label. Caso exista este atributo, na linha 13, a instância do atributo é adicionada ao objeto da solução que possui o nome C.origin.label e valor C.ref. O tipo do atributo é inferido através da reflexão, onde a classe do metamodelo com nome C.owner.label possui o atributo de nome C.label. Para tratar casos onde classes

diferentes possuem atributos com o mesmo nome, é usado o valor de C.owner.label para obter a classe correta através da reflexão.

Finalmente, na linha 16 o modelo completo é retornado.

Na figura 4.7 temos o resultado da transformação 4. Um modelo que está em conformidade com o metamodelo produzido pela transformação 3. Mais uma vez, este modelo foi corretamente verificado pelo Eclipse. Nesta saída o relógio produzido possui os métodos Start, Stop e DisplayTime.

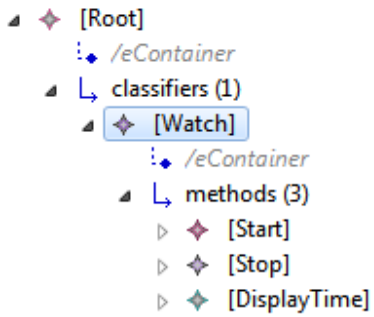


Figura 4.7: Saída da Transformação 4

Finalmente, com estas quatro transformações combinadas, temos um único código para injetar a solução de qualquer programa Alloy no espaço tecnológico MDE.

Porém, esta abordagem possui suposições que introduzem algumas restrições: (1) a implementação depende do resolvedor utilizado; (2) apenas os aspectos estruturais dos modelos foram considerados.

Para utilizar outro resolvedor, como por exemplo o ECLiPSe, é preciso reimplementar as quatro etapas. Para a transformação 1 é preciso definir um novo metamodelo do programa ECLiPSe e implementar o seu respectivo injetor. Para a transformação 2 é preciso criar um metamodelo de instâncias e seu respectivo injetor. Para a transformação 3 é preciso reimplementar a transformação utilizando o metamodelo criado na transformação 1. Finalmente, código da TIMeR deve ser reescrito utilizando o metamodelo de instâncias utilizado na transformação 2.

O metamodelo de instâncias do resolvedor é fixo porque depende do resolvedor utilizado. O código da transformação usa os elementos deste metamodelo para fazer a ponte entre o modelo da solução do resolvedor e o modelo de saída. Por exemplo, um campo da

solução Alloy possui um átomo de partida e outro de destino. A transformação usa essa informação para criar referências entre as classes correspondentes a esses átomos. No caso da injeção da solução, e onde apenas o metamodelo de saída é diferente, esta abordagem é muito boa, temos uma única transformação para diversos metamodelos de saída.

A implementação atual considera apenas os aspectos estruturais dos modelos. Para suportar modelos comportamentais é preciso estender a abordagem atual e considerar o elemento EOperation do Ecore.

Esta abordagem é diferente da JTL porque não é necessário criar uma transformação específica em tempo de execução para cada metamodelo de saída.

A seguir temos os resultados obtidos com um estudo de caso de linha de produto de software e a transformação de modelo de classe para relacional.

CAPÍTULO 5

EXPERIMENTOS

Neste capítulo apresentaremos os resultados da implementação utilizando dois estudos de caso. O primeiro estudo de caso é a Linhas de Produto de Software (Software Product Lines, SPL) [15] para a modelagem como busca, e o segundo estudo de caso é a transformação Class2Relational [22] para a transformação como busca. Nestes dois estudos de caso, apenas os dois arquivos de entrada (programa Alloy e solução Alloy) foram modificados. O código das transformações usado em todas as fases da cadeia é o mesmo. Os arquivos com as soluções foram gerados pelo Alloy usando o resolvedor SAT4J [10].

As quatro transformações foram executadas em sequência. Em seguida, o modelo de saída produzido foi verificado empiricamente usando as ferramentas da IDE Eclipse. Esta verificação consiste em determinar se o modelo está em conformidade com o metamodelo e se a saída está de acordo com a solução produzida pelo resolvedor.

O computador usado em todos os testes possui a seguinte configuração: Processador: Intel Core i5 2.8 GHz com 8GB de memória RAM com o sistema operacional Windows 7 64-bits. A versão utilizada do Java foi a 1.7.1 32-bits.

5.1 Linhas de Produto de Software

As linhas de produto de software permitem criar um conjunto de sistemas de software similares a partir de uma especificação comum a todos estes sistemas [37]. O primeiro é definir um modelo para o domínio compartilhado (especificação do sistema, suas características, componentes e pontos de variação). No contexto da modelagem como busca, este modelo de domínio compartilhado é o metamodelo com restrições.

O exemplo do SPL utilizado é o mesmo de [37]. O metamodelo utilizado neste estudo de caso (figura 5.3) permite definir classes, e estas classes possuem métodos. O objetivo é gerar classes que simulam a execução de um relógio. No caso, há cinco possíveis tipos

de relógios a serem produzidos: 1) simples; 2) com alarme; 3) com alarme sonoro; 4) com alarme sonoro e visual; e 5) com alarme visual. Estas combinações são determinadas pelas restrições do metamodelo. Estas restrições estão no código 4.1 do capítulo anterior.

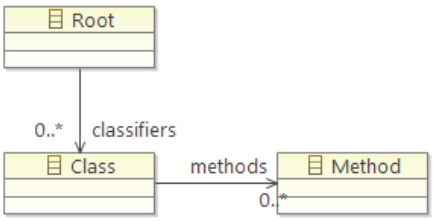


Figura 5.1: Metamodelo de Classe do estudo de caso SPL

Abaixo temos o resultado da cadeia de transformações: um documento XML produzido corretamente utilizando a TIMeR. Graficamente este modelo é exibido na figura 5.2. A solução encontrada pelo Alloy possui os métodos Start, Stop e DisplayTime. Da mesma forma, podemos ver que no modelo gerado Watch possui referências aos métodos Start, Stop e DisplayTime. Para as outras soluções os resultados foram similares.

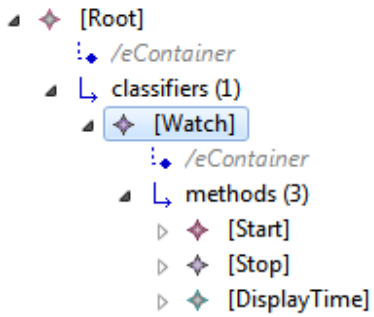


Figura 5.2: Modelo produzido pela TIMeR para o estudo de caso SPL

Etapa	**Tempo Médio**
Transformação 1	0,0952
Transformação 2	2,5007
Transformação 3	0,9516
Transformação 4	0,6792
Total	4,2267

Tabela 5.1: Tempo médio em segundos de cada etapa do estudo de caso SPL

Na tabela 5.1 temos os tempos de execução em segundos de cada etapa da cadeia. A

Entrada	Tamanho
Programa Alloy	11 assinaturas, 2 campos
Instância Alloy	15 assinaturas, 9 átomos, 2 campos, 4 tuplas

Tabela 5.2: Tamanho da entrada de cada etapa do estudo de caso SPL

cadeia foi executada 20 vezes e inclui o tempo de serialização e deserialização dos modelos em cada etapa. O tamanho das entradas é dado na tabela 5.2.

5.2 Transformação Classe para Relacional

O segundo estudo de caso testado foi o primeiro cenário da transformação como busca em [38], isto é, a transformação Class2Relational. O objetivo desta transformação é transformar um modelo de classe em um modelo relacional.

A figura 4.1 ilustra o modelo de classe utilizado como entrada para a transformação como busca.

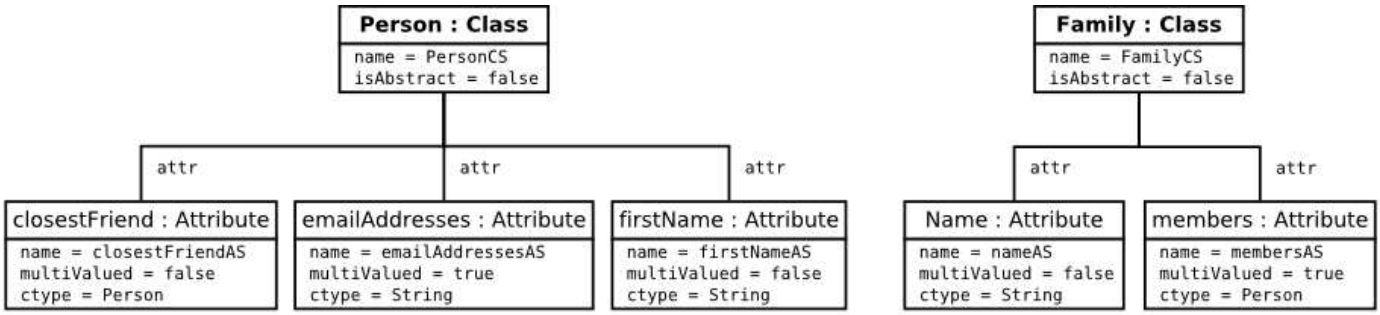


Figura 5.3: Metamodelo de Classe usado como entrada

Mais uma vez, o programa Alloy e sua respectiva solução são utilizados como entrada para a cadeia de transformações. Na figura 5.4 temos o modelo gerado e verificado pelo Eclipse.

Neste caso a transformação como busca produz como saída os dois modelos, o de classe e o relacional, pois a TIMeR recebeu como entrada um metamodelo que continha a unificação dos metamodelos de entrada e saída. Como vimos anteriormente, na transformação como busca há um passo posterior chamado 'cut' que extrai desta solução o modelo de interesse. Também é possível obter apenas a solução do modelo relacional se o metamodelo de saída apresentado à TIMeR for apenas o metamodelo relacional. A figura

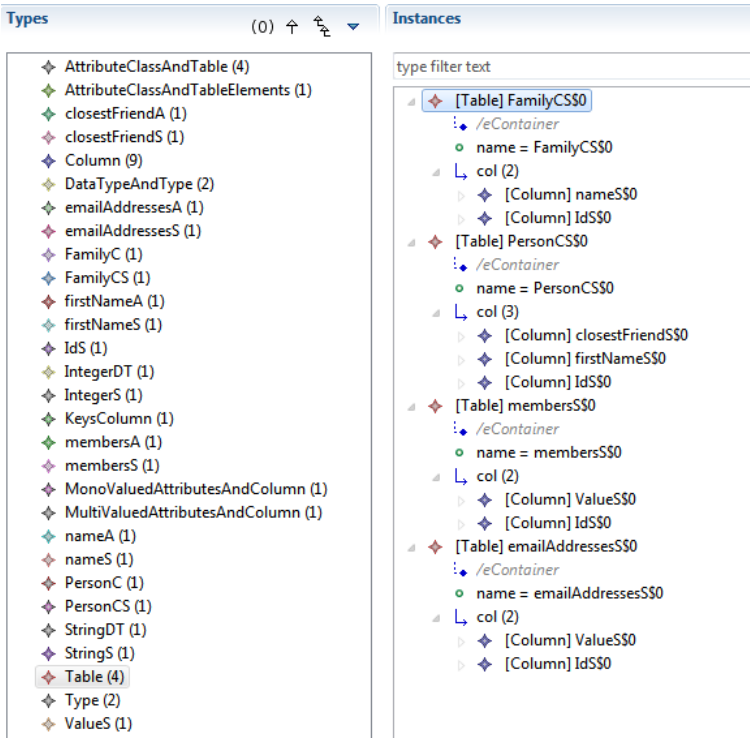


Figura 5.4: Saída do estudo de caso Class2Relational

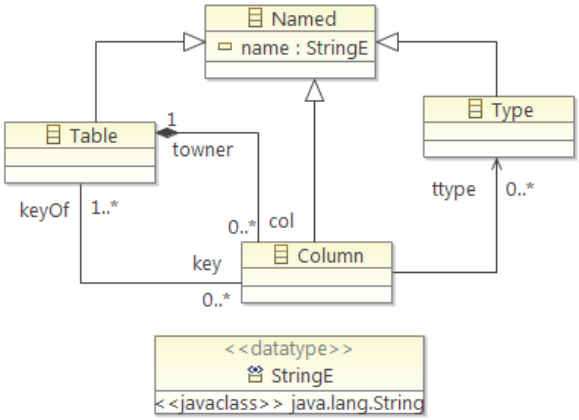


Figura 5.5: Metamodelo relacional [22]

5.6 ilustra este exemplo como o metamodelo de saída na figura 5.5 apresentado à TIMeR e o modelo produzido na figura 5.6. Neste caso o arquivo de entrada da solução foi o mesmo.

Na tabela 5.3 temos os tempos médios de execução em segundos, e na tabela 5.4 temos o tamanho das entradas da cadeia.

Podemos ver mais uma vez que o tempo de execução da transformação 2 foi responsável por grande parte do tempo de processamento (98%). Apesar do tamanho da

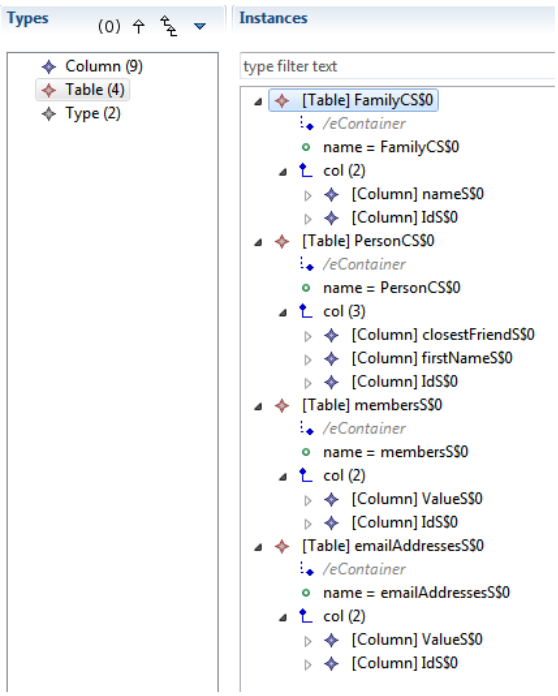


Figura 5.6: Saída do estudo de caso Class2Relational com a operação cut

Etapa	**Tempo Médio**
Transformação 1	0,1266
Transformação 2	185,084
Transformação 3	1,2015
Transformação 4	0,7901
Total	187,202

Tabela 5.3: Tempo médio em segundos de cada etapa da cadeia para o estudo de caso Class2Relational

Entrada	**Tamanho**
Programa Alloy	39 assinaturas, 23 campos
Instância Alloy	43 assinaturas, 55 átomos, 23 campos, 116 tuplas

Tabela 5.4: Tamanho da entrada de cada etapa do estudo de caso Class2Relational

solução do estudo de caso Class2Relational conter mais elementos, o tempo das outras três transformações não aumentaram proporcionalmente.

No próximo capítulo será apresentado as conclusões e sugestões para trabalhos futuros.

CAPÍTULO 6

CONCLUSÕES E TRABALHOS FUTUROS

Nesta dissertação foi apresentada uma abordagem que possibilita a interoperabilidade de modelos escritos no espaço tecnológico da engenharia dirigida por modelos (MDE) com programas escritos no espaço tecnológico de resolvedores de busca. Esta interoperabilidade permite que aplicações MDE tirem proveito das capacidades de resolvedores de busca para executar operações complexas.

Para permitir esta interoperabilidade, foi desenvolvida uma cadeia com quatro transformações de modelos. Estas transformações realizam uma injeção dos dados do programa do resolvedor de busca em um modelo e metamodelos no formato de uma plataforma MDE.

Destas quatro transformações, uma tem a particularidade de poder ser implementada sem o conhecimento do metamodelo de saída. Este será interpretado no momento da execução. Por isso a abordagem é chamada de Transformação independente de Metamodelo.

O desenvolvimento da solução teve diferentes contribuições, descritas abaixo:

1. Implementação de um caso de uso de transformação de modelos usando um resolvedor de busca, para avaliação da abordagem chamada "Transformação como Busca". Foi escolhida a transformação Class2Relational, por ser uma referência em aplicações MDE. Esta implementação fortaleceu a hipótese da necessidade de automatização da cadeia de interoperabilidade entre os diferentes espaços tecnológicos.

2. Definição de uma cadeia de quatro transformações para transformar a solução no formato do resolvedor em um modelo em conformidade com o metamodelo de saída para uma operação de modelagem como busca. A cadeia de operações foi instanciada usando o framework Alloy. As três primeiras transformações são relativas a interoperabilidade entre os espaços tecnológicos:

2.1 Injeção do programa Alloy em um modelo de programa Alloy no espaço tecnológico MDE.

2.2 Injeção da solução do framework Alloy num modelo XML, e em seguida este modelo é transformado em um modelo de instâncias Alloy. Este modelo de instâncias Alloy está em conformidade com um metamodelo criado a partir do esquema XSD utilizado pelos documentos XML que armazenam a solução do framework Alloy.

2.3 Extração do metamodelo de saída a partir do modelo de programa Alloy.

2.4 Definição de uma transformação independente do metamodelo de saída (TIMeR). Esta transformação permite injetar um modelo no espaço tecnológico MDE em conformidade com um metamodelo conhecido apenas durante o tempo de execução. A TIMeR, diferente das abordagens atuais, recebe como entrada tanto o metamodelo de saída quanto o modelo com a solução do resolvedor. A TIMeR produz como saída um modelo conforme a este metamodelo de entrada. Para que a transformação possa ser executada, foi necessário definir uma função de mapeamento entre o modelo e seu respectivo metamodelo.

Pelos resultados encontrados foi verificado que esta abordagem é viável e o objetivo foi alcançado. As soluções foram injetadas corretamente no espaço tecnológico MDE em dois casos de estudo. A escrita de uma única transformação foi capaz de realizar transformações para diferentes metamodelos de saída. O uso dos métodos de reflexão do framework EMF permitiu a TIMeR obter as informações necessárias dos modelos e metamodelos, para que durante o tempo da execução o modelo de entrada fosse transformado de forma adequada.

Analisando os pontos em aberto da abordagem, verificou-se que a implementação depende do resolvedor utilizado. Caso seja utilizado outro resolvedor, é preciso reimplementar todas as etapas da cadeia. Como já explicitado, uma das suposições da TIMeR é considerar que os modelos de entrada e saída possuam uma estrutura similar, definida por uma função de similaridade entre os elementos dos metamodelo. Na implementação atual esta similaridade é definida por elementos de nomes iguais e de tipos correspondentes.

Isto é uma limitação que a impede de ser usada de forma geral, porém no caso de injeções de soluções de resolvedores não há este problema. Entretanto, a abordagem é extensível por permitir a trocar a função de mapeamento. Outra limitação da abordagem é que são considerados apenas os aspectos estruturais do modelo, para tratar os aspectos dinâmicos dos modelos é preciso estender a abordagem atual.

Há diferentes possibilidades de trabalhos futuros, como descrevemos abaixo:

- Utilizar uma variação desta abordagem para implementar a geração dos dados de busca (transformação M2SP) da cadeia da modelagem como busca. Nesta transformação o metamodelo de entrada é desconhecido, e o modelo de entrada deve ser transformado em um modelo de programa Alloy.

- Otimização das quatro etapas da cadeia. Principalmente uma nova implementação da transformação 2, possivelmente usando a própria API do EMF, de modo a melhorar o tempo de execução.

- Pode ser investigado o uso de um modelo de correspondências entre as entradas da transformação 4 para permitir flexibilizar ainda mais esta abordagem.

- Implementação destas transformações usando outros resolvedores de programação por restrições.

BIBLIOGRAFIA

[1] IBM ILOG CPLEX Optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`, 2010.

[2] Atl developer guide. `http://wiki.eclipse.org/ATL/Developer_Guide`, 2013.

[3] Ocl+ usecase. `http://www.lsis.org/kleinerm/MS/OCLP_mm.html`, 2013.

[4] SAPos - sistema de apoio à pós-graduação. `http://web.inf.ufpr.br/didonet/sapos-sistema-de-apoio-a-pos-graduacao`, 2013.

[5] Francis Alizon, Mariano Belaunde, Grégoire DuPre, Bertrand Nicolas, Sébastien Poivre, e Jacques Simonin. Les modèles dans l'action à France Télécom avec SmartQVT. *Génie logiciel: Congrès Journées Neptune No5*, 2007.

[6] Freddy Allilaire, Jean Bezivin, Hugo Bruneliere, e Frederic Jouault. Global Model Management In Eclipse GMT/AM3. 2006.

[7] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003.

[8] Krzysztof R. Apt e Mark Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[9] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, e Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place emf model transformations. DorinaC. Petriu, Nicolas Rouquette, e Oystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, páginas 121–135. Springer Berlin Heidelberg, 2010.

[10] Daniel Le Berre e Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[11] Matthias Biehl. Literature Study on Model Transformations. Relatório Técnico ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology, julho de 2010.

[12] Marco Brambilla, Jordi Cabot, e Manueal Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering.

[13] Jordi Cabot, Robert Clarisó, e Daniel Riera. Umltocsp: A tool for the formal verification of uml/ocl models using constraint programming. *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, páginas 547–548, New York, NY, USA, 2007. ACM.

[14] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, e Alfonso Pierantonio. Jtl: A bidirectional and change propagating transformation language. Brian A. Malloy, Steffen Staab, e Mark van den Brand, editors, *SLE*, volume 6563 of *Lecture Notes in Computer Science*, páginas 183–202. Springer, 2010.

[15] Paul Clements e Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[16] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, páginas 151–158, New York, NY, USA, 1971. ACM.

[17] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, e Dániel Varró. Viatra "visual automated transformations for formal verification and validation of uml models. *Proceedings of the 17th IEEE international conference on Automated software engineering*, ASE '02, páginas 267–, Washington, DC, USA, 2002. IEEE Computer Society.

[18] Krzysztof Czarnecki e Simon Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, julho de 2006.

[19] Marcos Didonet Del Fabro. *Metadata management using model weaving and model transformations*. Tese de Doutorado, University of Nantes, setembro de 2007.

[20] Eclipse Modeling Project. Xpand / xtend reference, 2012.

[21] Eclipse.org. Atl transformation example - book to publication. `http://www.eclipse.org/atl/atlTransformations/Book2Publication/ExampleBook2Publication[v00.02].pdf`, 2013.

[22] Eclipse.org. Atl transformation examples. `http://www.eclipse.org/atl/atlTransformations/`, 2013.

[23] Niklas Eén e Niklas Sörensson. An extensible sat-solver. Enrico Giunchiglia e Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, páginas 502–518. Springer, 2003.

[24] Sven Efftinge, Peter Friese, Arno Haase, Clemens Kadura, Bernd Kolb, Dieter Moroff, Karsten Thoms, e Markus Völter. *open Architecture Ware User Guide*, 2007.

[25] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, e Gordon Woodhull. Graphviz - Open Source Graph Drawing Tools. *Graph Drawing*, páginas 483–484, 2001.

[26] Ramez A. Elmasri e Shankrant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[27] Martin Gogolla, Fabian Büttner, e Mark Richters. Use: A uml-based specification environment for validating uml and ocl. *Sci. Comput. Program.*, 69(1-3):27–34, dezembro de 2007.

[28] Philipp Huber. The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches. Dissertação de Mestrado, Technische Universität Wien, maio de 2008.

[29] Aleksandar. Torlak Emina. Kang Eunsuk. Jackson, Daniel. Milicevic e Joe Near. Alloy 4.2. `http://alloy.mit.edu/alloy/`, 2013.

[30] Daniel Jackson. Automating first-order relational logic. *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, SIGSOFT '00/FSE-8, páginas 130–139, New York, NY, USA, 2000. ACM.

[31] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[32] Joxan Jaffar e Michael J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[33] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, e Ivan Kurtev. Atl: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, junho de 2008.

[34] Frédéric Jouault, Jean Bézivin, e Ivan Kurtev. Tcs:: A dsl for the specification of textual concrete syntaxes in model engineering. *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, páginas 249–254, New York, NY, USA, 2006. ACM.

[35] Stuart Kent. Model driven engineering. *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, páginas 286–298, London, UK, UK, 2002. Springer-Verlag.

[36] Mathias Kleiner, Marcos Didonet Del Fabro, e Patrick Albert. Alloy metamodel and tcs usecase for model search. `http://www.lsis.org/kleinerm/MS/Alloy_mm.html`, 2010.

[37] Mathias Kleiner, Marcos Didonet Del Fabro, e Patrick Albert. Model search: Formalizing and automating constraint solving in mde platforms. Thomas Kuhne, Bran Selic, Marie-Pierre Gervais, e François Terrier, editors, *ECMFA*, volume 6138 of *Lecture Notes in Computer Science*, páginas 173–188. Springer, 2010.

[38] Mathias Kleiner, Marcos Didonet Del Fabro, e Davi De Queiroz Santos. Transformation as search. Pieter Van Gorp, Tom Ritter, e Louis M. Rose, editors, *ECMFA*, volume 7949 of *Lecture Notes in Computer Science*, páginas 54–69. Springer, 2013.

[39] Dimitrios S. Kolovos, Richard F. Paige, e Fiona A.C. Polack. The epsilon transformation language. Antonio Vallecillo, Jeff Gray, e Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, páginas 46–60. Springer Berlin Heidelberg, 2008.

[40] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[41] Ivan Kurtev. State of the art of qvt: A model transformation language standard. Andy Schurr, Manfred Nagl, e Albert Zundorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, páginas 377–393. Springer Berlin Heidelberg, 2008.

[42] Jean marie Favre. Towards a basic theory to model model driven engineering. *In Proc. of the UML2004 Int. Workshop on Software Model Engineering*, 2004.

[43] Tom Mens e Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, março de 2006.

[44] MIT. Alloy language reference. `http://alloy.mit.edu/alloy/documentation/book-chapters/alloy-language-reference.pdf`, 2013.

[45] Naouel Moha, Sagar Sen, Cyril Faucher, Olivier Barais, e Jean-Marc Jezequel. Evaluation of kermeta for solving graph-based problems. *Int. J. Softw. Tools Technol. Transf.*, 12(3-4):273–285, julho de 2010.

[46] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, e Sharad Malik. Chaff: Engineering an efficient sat solver. *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, páginas 530–535, New York, NY, USA, 2001. ACM.

[47] Object Management Group, Inc. *UML Object Constraint Language (OCL) 2.0 Specification*, outubro de 2006.

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

[49] OMG. *Meta Object Facility (MOF) Core Specification*, 2006.

[50] OMG. *XML Metadata Interchange (XMI)*. OMG, 2007.

[51] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)*, 2011.

[52] Jean remy Falleri, Marianne Huchard, e Clementine Nebut. C.: Towards a traceability framework for model transformations in kermeta. *In: ECMDA-TW Workshop*, 2006.

[53] James Rumbaugh, Ivar Jacobson, e Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[54] Andy Schurr. Specification of graph translators with triple graph grammars. Ernst W. Mayr, Gunther Schmidt, e Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, páginas 151–163. Springer Berlin Heidelberg, 1995.

[55] Roger S. Scowen. Extended BNF - A Generic Base Standard. *Proceedings of the 1993 Software Engineering Standards Symposium (SESS'93)*, agosto de 1993.

[56] David Steinberg, Frank Budinsky, Marcelo Paternostro, e Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[57] Leon Sterling e Ehud Shapiro. *The Art of Prolog (2Nd Ed.): Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1994.

[58] W3C. Extensible markup language (xml) 1.0. `http://www.w3.org/TR/REC-xml/`, 2008.

[59] W3C. Xml schema 1.1. `http://www.w3.org/XML/Schema/`, 2010.

APÊNDICE A

CÓDIGO DA TRANSFORMAÇÃO 4

Esta é a implementação em Java da TIMeR para transformar um modelo de instância Alloy em um modelo em conformidade como o metamodelo de saída passado pelo segundo parâmetro de execução.

Código A.1: Código Java da Transformação 4

```
1  import java.io.IOException;
2  import java.security.AllPermission;
3  import java.util.ArrayList;
4  import java.util.Collections;
5  import java.util.HashMap;
6  import java.util.Iterator;
7  import java.util.List;
8  import java.util.Map;
9
10 import org.eclipse.emf.common.util.URI;
11 import org.eclipse.emf.ecore.EClass;
12 import org.eclipse.emf.ecore.EDataType;
13 import org.eclipse.emf.ecore.EObject;
14 import org.eclipse.emf.ecore.EReference;
15 import org.eclipse.emf.ecore.EStructuralFeature;
16 import org.eclipse.emf.ecore.resource.Resource;
17 import org.eclipse.emf.ecore.resource.ResourceSet;
18 import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
19 import org.eclipse.emf.ecore.util.EcoreUtil;
20 import org.eclipse.emf.ecore.xmi.XMLResource;
21 import org.eclipse.emf.ecore.xmi.impl.XMIResourceFactoryImpl;
22 import org.eclipse.emf.ecore.EPackage;
23
24
25
26 public class Transformacao {
27
28    private List<EObject> allInstances = new ArrayList<EObject>();
29    private List<EPackage> allPackages = new ArrayList<EPackage>();
30    private List<EClass> allClasses = new ArrayList<EClass>();
31    private List<EStructuralFeature> allSF = new ArrayList<↩
         EStructuralFeature>();
32    private List<EDataType> allDataTypes= new ArrayList<EDataType>();
33
34    private List<EObject> solutionAllInstances = new ArrayList<EObject↩
         >();
35    private List<EObject> solutionAllAtoms = new ArrayList<EObject>();
36    private List<EObject> solutionAllFields = new ArrayList<EObject>();
37    private List<EObject> solutionAllSigs = new ArrayList<EObject>();
```

```
38
39      private HashMap<EObject, EObject> Atom2EClass  = new HashMap<↩
            EObject, EObject>();
40      private HashMap<EObject, EObject> EClass2Atom = new HashMap<EObject↩
            , EObject>();
41
42      private HashMap<EObject, List<EObject>> AtomFields = new HashMap<↩
            EObject, List<EObject>>();
43      private HashMap<EObject, Map<String, List<EObject>>> AtomFields2 = ↩
            new HashMap<EObject, Map<String, List<EObject>>>();
44
45      public Transformacao(String inputECoreModel, String ↩
            inputAlloySolutionEcore, String inputAlloySolution, String ↩
            outputResultModel, String rootClassName)
46      {
47          Resource.Factory.Registry reg = Resource.Factory.Registry.↩
                INSTANCE;
48          Map m = reg.getExtensionToFactoryMap();
49          m.put("xmi", new XMIResourceFactoryImpl());
50          m.put("ecore", new XMIResourceFactoryImpl());
51
52          ResourceSet rs = new ResourceSetImpl();
53          Resource ecoreInRes = rs.getResource(URI.createURI(""+↩
                inputECoreModel), true);                  // Carrega ↩
                metamodelo ecore
54          Resource alloySolEcoreInRes = rs.getResource(URI.createURI(""+↩
                inputAlloySolutionEcore), true);      // carrega metamodelo ↩
                da solução para extrair o pacote
55          EPackage epkg = (EPackage)alloySolEcoreInRes.getContents().get↩
                (0);                                  // extrai o pacote
56          rs.getPackageRegistry().put(epkg.getNsURI(), epkg); ↩
                                                        // registra o ↩
                pacote
57          Resource alloySolInRes = rs.getResource(URI.createURI(""+↩
                inputAlloySolution), true);              // Carrega o modelo ↩
                com a soluçãoo.
58
59
60          processECoreModel(ecoreInRes);
61          processAlloySolution(alloySolInRes);
62
63          // Executa a transformação
64          List outElements = doTransformation();
65
66          // Salva o modelo
67
68          URI uri = URI.createURI("" + outputResultModel);
69          Resource outRes = rs.createResource(uri);
70
71          //System.out.println("\n\nResultado da Transformacao\n\n");
72          imprimeResultadoTransformacao(outElements);
73          //System.out.println("\n\nPreparando para salvar\n\n");
74
75          /**/
76          List objsToSearch = new ArrayList();
77          List<EObject> danglingObjs = new ArrayList();
78          for (Iterator iterator = outElements.iterator(); iterator.↩
                hasNext();) {
```

```
79          EObject object = (EObject) iterator.next();
80
81          if ((object.eContainer() == null && (rootClassName == null ||↩
                rootClassName.equals("")))
82                  || ( object.eContainer() == null && object.eClass↩
                       ().getName().equals(rootClassName)))
83             outRes.getContents().add(object);
84
85          if (EcoreUtil.getURI(object).toString().equals("#//")) {
86             danglingObjs.add(object);
87          }
88          else {
89             objsToSearch.add(object);
90          }
91       }
92       /**/
93       boolean completed = false;
94       Object currentObj = null;
95       while ( !completed && ! ( rootClassName == null || rootClassName↩
             .equals("")) )  {
96       for (Iterator<EObject> iterator = danglingObjs.iterator(); ↩
             iterator.hasNext();) {
97          EObject object = (EObject) iterator.next();
98          if (EcoreUtil.UsageCrossReferencer.find(object, objsToSearch)↩
             .size() > 0) {
99             outRes.getContents().add(object);
100            objsToSearch.add(object);
101            currentObj = object;
102            completed = false;
103            break;
104            }
105         completed = true;
106         }
107         if (completed || danglingObjs.size() == 0)
108            break;
109            danglingObjs.remove(currentObj);
110      }
111      /**/
112      try {
113         if (outRes.getContents().size() > 0 ) {
114            outRes.save(Collections.EMPTY_MAP);
115         }
116         else
117            System.out.println("Modelo vazio");
118      } catch (IOException e) {
119         System.out.println("Erro");
120         e.printStackTrace();
121      }
122      /**/
123      System.out.println("Pronto");
124
125   }
126
127
128
129   private List doTransformation() {
130      List outElements = new ArrayList();
131
```

```
132
133         // cria um elemento do modelo para cada atom
134         for (EObject atom : solutionAllAtoms) {
135
136            // Salva o label do atom e da sig do atom
137            String label = getValue(atom, "label").toString();
138            String sigLabel = getSigLabelFromAtom(atom).toString();
139            sigLabel = sigLabel.split("this/")[sigLabel.split("this/").↩
                  length-1];
140            System.out.println("atom "+ label);
141            //System.out.println("1 "+ sigLabel);
142
143            // Cria uma classe para cada elemento:
144            EClass eClass = getClassByName(sigLabel);
145            //System.out.println(eClass);
146            if(eClass == null) {
147               System.out.println("Nao tem uma class para a sig " + ↩
                     sigLabel);
148               continue;
149            }
150
151            EObject newObj = eClass.getEPackage().getEFactoryInstance().↩
                  create(eClass);
152            //System.out.println(newObj);
153
154            // Salva o elemento
155            EClass2Atom.put(newObj, atom);
156            Atom2EClass.put(atom, newObj);
157
158            List<EObject> listaFields = new ArrayList<EObject>();
159            HashMap<String,List<EObject>> listaFields2 = new HashMap<↩
                  String, List<EObject>>();
160            for (EObject field : solutionAllFields) {
161               //EObject owner = (EObject) getValue(field, "owner");
162               EObject origin = (EObject) getValue(field, "origin");
163               EObject dest = (EObject) getValue(field, "ref");
164
165               if(getValue(origin, "label").equals(getValue(atom, "label↩
                     ")))
166               {
167                  System.out.println("   " + getValue(field, "label"));
168                  System.out.println("       origem:" + getValue(origin, ↩
                        "label") + " destino:" + getValue(dest, "label"));
169                  listaFields.add(field);
170
171                  if(listaFields2.containsKey(getValue(field, "label").↩
                        toString())) {
172                     listaFields2.get(getValue(field, "label").toString()↩
                           ).add(field);
173                  }
174                  else {
175                     List<EObject> novaLista = new ArrayList<EObject>();
176                     novaLista.add(field);
177                     listaFields2.put(getValue(field, "label").toString()↩
                           , novaLista);
178                  }
179               }
180            }
```

```
181             AtomFields.put(newObj, listaFields);
182             AtomFields2.put(newObj, listaFields2);
183 
184             outElements.add( newObj );
185         }
186 
187         imprimeResultadoProcessamentoDosFields();
188 
189         System.out.println("-------------------------------------\n\n")↩
                ;
190 
191         // Itera sobre os elementos criados e completa os elementos.
192         for (Iterator<EObject> iterator = outElements.iterator(); ↩
                iterator.hasNext();) {
193             EObject modelElement =  iterator.next();
194             System.out.println(modelElement.eClass().getName() + " " + ↩
                    modelElement.hashCode());
195 
196             for (Iterator<EStructuralFeature> sfIterator = modelElement.↩
                    eClass().getEAllStructuralFeatures().iterator(); ↩
                    sfIterator.hasNext();) {
197                 EStructuralFeature feature = sfIterator.next();
198                 System.out.println("    " + feature.getName());
199 
200                 //if(feature.getName().equals("super")) System.out.println↩
                        ("    super!" + feature.getName());
201 
202                 List<EObject> fields = AtomFields.get(modelElement);
203                 Map<String, List<EObject>> fields2 = AtomFields2.get(↩
                        modelElement);
204                 List<EObject> valores = fields2.get(feature.getName());
205 
206                 //if(feature.getName().equals("super")) { System.out.↩
                        println("    vals=" + valores); }
207                 if (feature instanceof EReference)
208                 {
209                     if (feature.getUpperBound() > 1 || feature.↩
                            getUpperBound() == -1 )
210                     {
211                         if(valores == null) continue;
212 
213                         for (EObject e : valores) {
214                             EObject val = (EObject) getValue(e, "ref");
215                             System.out.println("        " + Atom2EClass.get(↩
                                    val).eClass().getName() + " " + Atom2EClass.↩
                                    get(val).hashCode());
216                             ((List)modelElement.eGet(feature)).add(  ↩
                                    Atom2EClass.get(val)  );
217                         }
218                     }
219                     else // mono valorado
220                     {
221                         if(valores == null) continue;
222 
223                         EObject val = (EObject) getValue(valores.get(0), "↩
                                ref");
224                         if(val == null) continue;
225 
```

```
226                 //System.out.println("        " + Atom2EClass.get(↩
                       val).eClass().getName() + " " + Atom2EClass.get(↩
                       val).hashCode());
227                 modelElement.eSet(feature, Atom2EClass.get(val));
228                 //modelElement.eSet(feature, val);
229              }
230           }
231           else      // ?  atributo
232           {
233              String typeName = feature.getEType().getName().↩
                    toLowerCase();
234
235              if (typeName.indexOf("int") >= 0 || typeName.indexOf("↩
                    long") >= 0) {
236                 EObject val = (EObject) getValue(valores.get(0), "↩
                       ref");
237                 Integer intValue = new Integer(Integer.parseInt(↩
                       getValue(val,"label").toString()));
238                 System.out.println("        " + intValue);
239                 modelElement.eSet(feature, intValue);
240              } else if (typeName.indexOf("string") >= 0)
241              {
242                 EObject val = (EObject) getValue(valores.get(0), "↩
                       ref");
243                 String str = getValue(val,"label").toString();
244                 modelElement.eSet(feature, str);
245                 System.out.println("    " +  str);
246              }
247              else
248                 System.out.println("Erro tipo desconhecido. " + ↩
                       typeName);
249
250           }
251
252           System.out.println("        resultado: " + modelElement.↩
                 eGet(feature));
253
254        }
255
256     }
257
258     return outElements;
259  }
260
261  private void imprimeResultadoTransformacao(List outElements) {
262     for (Iterator iterator = outElements.iterator(); iterator.↩
           hasNext();) {
263        EObject object = (EObject) iterator.next();
264        System.out.println(object.eClass().getName());
265
266        for (Iterator<EStructuralFeature> sfIterator = object.eClass↩
              ().getEAllStructuralFeatures().iterator(); sfIterator.↩
              hasNext();) {
267           EStructuralFeature feature = sfIterator.next();
268           System.out.println("    " + object.eGet(feature));
269        }
270     }
271  }
```

```
272
273     private void imprimeResultadoProcessamentoDosFields() {
274        System.out.println("\n\n-------------------------------------");
275        for (Map.Entry<EObject, Map<String, List<EObject>>> classes :
              AtomFields2.entrySet()) {
276           System.out.println(classes.getKey().eClass().getName());
277
278           Map<String, List<EObject>> sf = classes.getValue();
279           for (Map.Entry<String, List<EObject>> d : sf.entrySet()) {
280              System.out.println("    " + d.getKey());
281
282              List<EObject> valores = d.getValue();
283              for (EObject e : valores) {
284                 EObject origin = (EObject) getValue(e, "origin");
285                 EObject dest = (EObject) getValue(e, "ref");
286
287                 System.out.println("       " + getValue(origin, "label
                       ") + ", " + getValue(dest,"label"));
288              }
289           }
290        }
291     }
292
293
294     // Retorna o valor de uma structuralfeature
295     private Object getValue(EObject atom, String value)
296     {
297        return atom.eGet(atom.eClass().getEStructuralFeature(value));
298     }
299     private Object getSigLabelFromAtom(EObject atom)
300     {
301        EObject o = (EObject) getValue(atom,"type");
302        return getValue(o, "label");
303     }
304
305     // Retorna uma classe a partir do nome
306     private EClass getClassByName(String name) {
307        String className = name.toLowerCase();
308        for (Iterator<EClass> classes = allClasses.iterator(); classes.
              hasNext();) {
309           EClass eClass = classes.next();
310           if (eClass.getName().toLowerCase().equals( className ))
311              return eClass;
312        }
313        return null;
314     }
315     public String getElementName(EClass aclass, EStructuralFeature
           feature) {
316        return "d_" + aclass.getName().toLowerCase() + "_" + feature.
              getName().toLowerCase();
317     }
318
319
320     private String getClassNameAlloySol(EObject obj)
321     {
322        EClass objClass = obj.eClass();
323        return objClass.getName();
```

```
324     }
325
326     ////////////////////////////
327     private void processAlloySolution(Resource model) {
328        for (Iterator elements = model.getAllContents(); elements.↩
              hasNext();) {
329           Object element = elements.next();
330           if (element instanceof EObject) {
331              //System.out.println("EObject "  + element.toString());
332              solutionAllInstances.add((EObject) element);
333
334              // Saves the atoms, fields and signatures
335              if(getClassNameAlloySol((EObject) element).toLowerCase().↩
                    equals("atom"))
336                 solutionAllAtoms.add((EObject) element);
337              if(getClassNameAlloySol((EObject) element).toLowerCase().↩
                    equals("field"))
338                 solutionAllFields.add((EObject) element);
339              if(getClassNameAlloySol((EObject) element).toLowerCase().↩
                    equals("sig"))
340                 solutionAllSigs.add((EObject) element);
341           }
342        }
343     }
344
345     private void processECoreModel(Resource model) {
346        for (Iterator elements = model.getAllContents(); elements.↩
              hasNext();) {
347           Object element = elements.next();
348           if (element instanceof EClass) {
349              //System.out.println("EClass "  + element.toString());
350              allClasses.add((EClass) element);
351           }
352           if (element instanceof EPackage) {
353              //System.out.println("EPackage "  + element.toString());
354              allPackages.add((EPackage) element);
355           }
356           if (element instanceof EStructuralFeature) {
357              //System.out.println("EStructuralFeature "  + element.↩
                    toString());
358              allSF.add((EStructuralFeature) element);
359           }
360           if (element instanceof EStructuralFeature) {
361              //System.out.println("EDataType "  + element.toString());
362              allSF.add((EStructuralFeature) element);
363           }
364           if (element instanceof EObject) {
365              //System.out.println("EObject "  + element.toString());
366              allInstances.add((EObject) element);
367           }
368        }
369     }
370
371     private static void debugModel(Resource model, String out) {
372        System.out.println(out);
373        for (Iterator elements = model.getAllContents(); elements.↩
              hasNext();) {
374           Object element = elements.next();
```

```
375             System.out.println(element);
376         }
377     }
378 
379     public static void main(String[] args) {
380 
381 
382         if (args.length < 2) {
383             System.out.println("Arguments not valid : {↩
                    IN_Alloy_Problem_metamodel_path, ↩
                    IN_AlloySolution_model_path, OUT_model_path}.");
384         } else {
385             String in1 = args[0];
386             String in2 = args[1];
387             String out = args[2];
388             Transformacao t = new Transformacao(in1, "Transformacao/↩
                    AlloyXSD.ecore", in2, out, "");
389         }
390     }
391 }
```