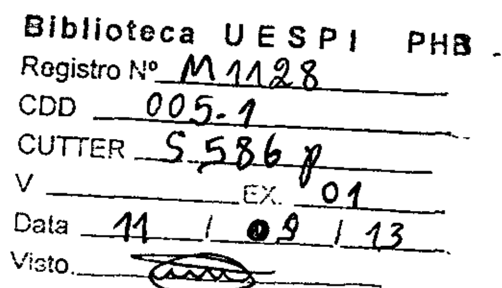
UNIVERSIDADE ESTADUAL DO PIAUÍ – UESPI

CAMPUS PROF. ALEXANDRE ALVES DE OLIVEIRA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCIANO KELVIN DA SILVA

UMA PROPOSTA DE FORMALIZAÇÃO DO DIAGRAMA DE CLASSE POR MEIO DA LINGUAGEM EVENT-B.


Biblioteca UESPI PHB
Registro Nº M1128
CDD 005.1
CUTTER S586p
V ______ EX. 01
Data 11 / 09 / 13
Visto

PARNAÍBA

2013

LUCIANO KELVIN DA SILVA

UMA PROPOSTA DE FORMALIZAÇÃO DO DIAGRAMA DE CLASSE POR MEIO DA LINGUAGEM EVENT-B.

Monografia submetida ao Curso de Bacharelado em Ciência da Computação da Universidade Estadual do Piauí, como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. M.Sc. Thiago Carvalho de Sousa

PARNAÍBA

2013

LUCIANO KELVIN DA SILVA

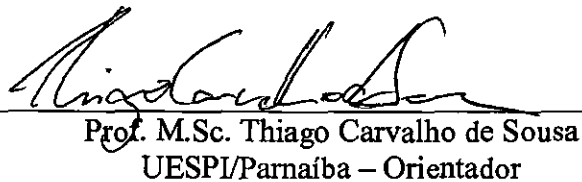
UMA PROPOSTA DE FORMALIZAÇÃO DO DIAGRAMA DE CLASSE POR MEIO DA LINGUAGEM EVENT-B

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Estadual do Piauí – UESPI, Campus Prof. Alexandre Alves de Oliveira, como parte das exigências da disciplina de Estágio Supervisionado, requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

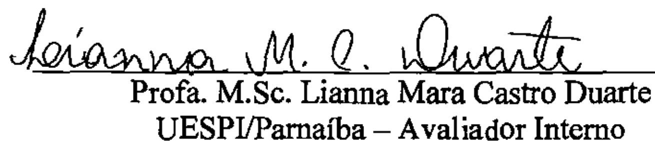
Orientador: M.Sc. Thiago Carvalho de Sousa

Monografia Aprovada em: **02 de agosto de 2013.**

Banca Examinadora:



Prof. M.Sc. Thiago Carvalho de Sousa
UESPI/Parnaíba – Orientador



Profa. M.Sc. Lianna Mara Castro Duarte
UESPI/Parnaíba – Avaliador Interno

Prof. M.Sc. Denival Araújo dos Santos
IFPI/Parnaíba – Avaliador Externo

DEDICATÓRIA

Dedico à minha mãe e minha irmã, que foram minhas principais incentivadoras, e sempre estiveram comigo.

Aos meus amigos e familiares que me apoiaram, e, sobretudo a Deus, por me capacitar, e me dar forças.

AGRADECIMENTO

Agradeço primeiramente a Deus por ter me dado saúde, força e sabedoria, nos momentos em que mais precisei.

A toda minha família, amigos e meus irmãos em cristo, por sempre me apoiarem e estarem comigo, em especial a minha mãe Maria do Socorro da Silva que sempre cuidou e me amou incondicionalmente, além de ter estado comigo em todos os momentos de minha vida, e a minha irmã Luciana Maria Silva Ferreira que sempre me incentivou e compartilhou de todas as minhas alegrias e tristezas.

À UESPI, e ao CNPQ pelo financiamento parcial deste projeto e ao professor Thiago C. de Sousa, por sua orientação, apoio, motivação e paciência durante o período de realização deste trabalho.

"Ser o homem mais rico do cemitério não é o que mais importa pra mim...

Ir para a cama à noite e pensar que foi feito alguma coisa grande.

Isso é o que mais importa pra mim."

Steve Jobs

RESUMO

Um dos primeiros passos em um processo de desenvolvimento de um software é o processo de modelagem, onde são levantados requisitos e feito os modelos que servirão de base para todo o decorrer do projeto. Porém, muitas vezes estes modelos são criados de forma inconsistente ou ambígua, causando problemas no decorrer do processo de desenvolvimento, obrigando os desenvolvedores a voltarem para a etapa de análise, para alterar novamente o modelo, causando assim um impacto negativo nos custos e prazos do projeto. Assim, este trabalho possui como objetivo principal o desenvolvimento de uma ferramenta de modelagem, na qual o analista possa desenhar um diagrama de classe e as inconsistências e ambiguidades dele já possam ser verificadas e corrigidas ainda no processo de análise por meio do seu mapeamento para a linguagem formal Event-B.
PALAVRAS-CHAVES: Desenvolvimento de Software, Engenharia de Software, Diagrama de Classe, Métodos Formais, Event-B, UML.

ABSTRACT

One of the first steps in a software development process is the modeling remover, where requirements are raised, and the models are the basis for the entire course of the project. But often these models are created in an inconsistent and ambiguous way, causing problems during the development process, forcing the developers to return to the analysis phase, and change the model again, impacting the project cost and deadlines. So, this work has as main goal, the development of a modeling tool, where the analyst can draw the class diagram, and check its inconsistency and ambiguity issues during the analysis phase through the Event-B formal language.

KEY-WORDS: Software Development, Software Engineering, Class Diagram, Formal Methods, Event-B, UML.

LISTA DE ILUSTRAÇÕES

Figura 2.1 – Hierarquia dos diagramas principais da UML. 16

Figura 3.1 – Representação gráfica de uma classe 19

Figura 3.2 – Representação gráfica de uma classe abstrata 20

Figura 3.3 – Representação de uma interface. 20

Figura 3.4- Representação de uma interface 21

Figura 3.5 – Relação de Dependência entre classes. 21

Figura 3.6- Herança de classe 21

Figura 3.7 – Agregação 22

Figura 3.8 – Composição. 22

Figura 4.1 – Modelo Event-B (*Machine*) com o nome "*changeMe*". 26

Figura 4.2 – Invariante (*Invariant*), representando a regra de que a propriedade saldo nunca pode ser menor que zero. 27

Figura 4.3 - Evento (*Event*), representando uma venda de ingresso. 27

Figura 4.4 - Relacionamento de um Modelo (*machine*) com um contexto (*context*). 28

Figura 5.1 - Dashboard do GMF. 30

Figura 5.2 - Arquitetura do QVT. 31

Figura 6.2 - Editor gráfico para o diagrama de classes. 37

Figura 6.3 – Contexto referente ao diagrama de classe com nome "Concessionaria". 38

Figura 6.4 – CarrierSet criado a partir de um classe com nome "Veiculo". 39

Figura 6.5 – Variável (*variable*) e invariante (*invariant*) criadas a partir de uma classe com nome "Veiculo". 39

Figura 6.6 – Eventos gerados a partir de uma classe nomeada "Veiculo". 40

Figura 6.7 – Invariante mapeada referente a um atributo "direcaoHidraulica" do tipo booleana em um uma classe "Veiculo" 41

Figura 6.8 – Invariante (*Invariant*) resultado do mapeamento de uma generalização de uma classe "Veiculo" para uma classe "Pickup" 41

Figura 6.9 – Invatiante (invariant) resutadod e uma agregação entre "Veículo" e "Acessorio" com o nome "itens" e com cardinalidade [0..n] e [0..n]. 42

Figura 6.10 – Regra de transformação resultada de um mapeamento de uma agregação 43

LISTA DE ABREVIATURAS E SIGLAS

POO – Programação Orientada a Objeto

OOSE - Object-Oriented Software Engineering

OMT - Object Modeling Technique

UML - Unified Modelling Language

OMG - Object Management Group

RTF - Revision Task Force

GMF – Graphical Modeling Framework

EMF - Eclipse Modeling Framework

QVT - Queries/Views/Transformations

GEF - Graphical Editing Framework

SUMÁRIO

1. Introdução 12

2. UML 15

2.1. História 15

2.2. Visão geral 16

2.3. Diagramas da UML 16

3. Diagrama de Classe 19

4. Métodos Formais 23

4.1. História 23

4.2. O que fazer com métodos formais 23

4.3. Taxonomia 24

4.4. Como Funciona 24

4.5. Visão geral 25

4.6. Event-B 26

5. GMF e QVT 29

5.1. Graphical Modeling Framework (GMF) 29

5.2. QVT (Queries/Views/Transformations) 30

6. Formalização do Diagrama de Classe 32

6.1. Modelagem 32

6.2. Descrição das meta-classes 33

6.3. Geração do Editor para o Diagrama de Classe 36

6.4. Regras de Transformação 37

7. Conclusão 44

7.2. Trabalhos futuros 44

8. Referências 45

9. Apêndice 47

10. Anexo 55

1 INTRODUÇÃO

Uma das grandes características do desenvolvimento de sistemas de médio e grande porte é a construção de inúmeros modelos parciais produzidos. Estes modelos são as especificações de requisitos, modelo de domínio, modelo de arquitetura, modelos de implantação, e, geralmente descrevem o software por ângulos distintos e em diferentes níveis de abstração. Para a construção destes modelos, muitas vezes são utilizados diferentes tipos de notações. Esta heterogeneidade em muitos casos leva a inúmeros problemas de inconsistência no decorrer do processo de desenvolvimento.

Um dos problemas é como resguardar a consistência horizontal [1], onde é preciso analisar se algumas alterações no modelo irão impactar em outros modelos do mesmo nível de abstração. Outro obstáculo é a consistência vertical [1], no qual é necessário verificar se os sucessivos refinamentos não afetarão as propriedades semânticas do modelo.

A falta de um meio de verificação dessas inconsistências no início do projeto, durante a fase de análise e especificação de requisitos, pode causar vários problemas durante o desenvolvimento, ou o sistema pode ser completamente implementado, porém com erros de inconsistências, o que comprometeria as regras de negócio. Além disso, uma pesquisa realizada em 2002 pelo órgão americano NIST (*National Institute of Standards and Technology*) indicou que, em alguns sistemas, cerca de 80% dos custos do desenvolvimento são consumidos pelo processo de remodelagem, processo esse que ocorre devido a diversos problemas, entre eles os problemas de inconsistência. Então se mostra cada vez mais necessário uma forma eficaz de detectar estes problemas, a fim de resolvê-los logo, antes da fase de desenvolvimento.

Atualmente os processos de desenvolvimento mais difundidos (RUP [2], Iconix [3], XP [4], Scrum [5], etc) possuem uma tarefa para verificação dessas inconsistências na fase de construção do sistema. Porém na maioria das vezes se utilizam de técnicas baseadas em inspeção de modelos e execução de uma bateria de testes. A inspeção de modelo comumente é feita manualmente e de forma visual, pelo fato dos modelos geralmente serem expressos por linguagens imprecisas, o que torna o trabalho bem mais custoso e totalmente dependente das habilidades do analista ou desenvolvedor. Em relação à bateria de testes,

mesmo tendo ferramentas para automatizar esta tarefa, ela somente representa apenas uma parte do sistema, visto que se tornaria inviável realizar bateria de testes para absolutamente todas as funcionalidades do sistema.

Já os métodos formais, que são uma subárea da engenharia de software, proveem uma série de técnicas baseadas em notações e linguagens com precisão lógicas e matemáticas, para especificação, verificação e desenvolvimento de sistemas de software. Dentre os métodos formais alguns deles (B [6], Event-B [7], VDM [8], Z [9]) são baseados em modelos, e podem detectar de maneira rápida e precisa as inconsistências apresentadas. Porém a utilização dos métodos formais ainda não se tornou uma prática comum na indústria de software, devido ao fato de que existem poucos profissionais qualificados que compreendam todos os conceitos matemáticos envolvidos.

1.1 Trabalhos relacionados

A verificação de modelagem de sistemas vem se tornando cada vez mais importante, visto o tamanho e a complexidade dos sistemas atuais. A utilização de métodos formais é uma das alternativas mais estudadas para esta tarefa, como no trabalho [10], onde é indicada a utilização de modelos matemáticos para analisar a consistência dos modelos UML, porém a falta de uma ferramenta CASE para integrar o modelo do sistema e os modelos matemáticos, dificulta a sua aplicação.

Outro tipo de análise é apresentado no trabalho [11] onde ele se aplica a OCL para verificação do diagrama de classe, porém ele se utiliza de uma validação estática, onde se faz necessária a construção de cenários de testes, o que faz com que aumente o tempo e o trabalho da verificação da modelagem. Embora possua algumas vantagens, como velocidade de execução, este estilo de verificação também apresenta pontos negativos, como o trabalho manual necessário na implementação dos cenários de testes, além de não haver uma garantia de que uma restrição feita não será violada.

No trabalho [12], ele se utiliza da linguagem OCL para descrever as definições das relações de consistência, porém essa linguagem necessita de um conhecimento maior sobre os conceitos matemáticos aplicados se comparada com a linguagem Event-B.

Percebe-se através dos trabalhos [10], [11] e [12] que a utilização de métodos formais para verificar a integridade dos diagramas da UML vem sendo constantemente estudada, porém não se tem ferramenta que realizem esta verificação diretamente.

Podemos notar que apesar da necessidade da verificação das inconsistências ser algo essencial no processo de criação de um software, os processos de desenvolvimento mais utilizados no mercado não proveem ao desenvolvedor formas eficazes de realizar esta verificação. Ao mesmo tempo os métodos formais fornecem técnicas eficientes para suprir esta necessidade dos desenvolvedores, porém ainda não é muito difundido no mercado. Vemos então que para auxiliar os desenvolvedores, bastaria incluir a utilização de métodos formais em algum processo de desenvolvimento já difundido no mercado, a fim de que os desenvolvedores possam realizar as verificações em seus modelos sem alterar sua maneira de desenvolvimento.

1.2 Objetivos

Este trabalho possui os seguintes objetivos:

- Elaborar regras matemáticas de transformação para mapear os elementos do diagrama de classe em elementos da linguagem Event-B.
- Desenvolver uma ferramenta CASE, que permita a criação de diagramas de classe, e possibilite executar as regras de transformação propostas.

No próximo capítulo iremos falar sobre a UML, sua história e seus diagramas. No capítulo 3 iremos nos aprofundar no diagrama de classe, que é o diagrama foco deste trabalho. No capítulo 4 explanaremos sobre os métodos formais, como eles funcionam, quais seus conceitos e componentes, e nos aprofundaremos na linguagem Event-B que será a linguagem formal que utilizaremos. As tecnologias utilizadas para a criação da ferramenta (GMF e QVT) são apresentadas no capítulo 5. No capítulo 6 iremos mostrar o processo de criação da ferramenta, assim como as regras de transformação Event-B. Por fim, no capítulo 7 iremos apresentar a conclusão, os trabalhos relacionados além de propostas para trabalhos futuros.

2 UML

2.1 História

No final da década de 70 e início da de 80, houve o surgimento de inúmeras linguagens orientadas a objeto (Objective C, C++, Eiffel), criando-se a necessidade de uma linguagem de modelagem que atendesse todas as necessidades para quem trabalhasse com POO (Programação Orientada a Objeto).

Foi então que começaram a aparecer algumas linguagens de modelagem com o propósito unificador. Com maior destaque apareceram as linguagens OOSE (*Object-Oriented Software Engineering*)[13] de Ivar Jacobson, Booch[13] do próprio Grady Booch e o OMT (*Object Modeling Technique*)[14] de James Rumbaugh.

Na década de 90, Jacobson, Booch e Rumbaugh começaram a utilizar métodos uns dos outros a fim de melhorar suas modelagens. Foi então que em 1994 eles se juntaram e decidiram unificar as três em uma só linguagem de modelagem, a UML *(Unified Modelling Language*)[15].

Segundo os próprios criadores da UML, as principais razões para unificarem seus modelos foram o fato de que os três já estavam evoluindo individualmente, então seria muito proveitoso se eles se juntassem e evoluíssem juntos, tirando assim a possibilidades de que algumas diferenças entre eles viessem a confundir os usuários. Outro motivo foi a possibilidade de trazer estabilidade ao mercado da orientação a objeto, apresentando uma linguagem de modelagem bem completa e que atendesse todas suas necessidades. Por último, o fato de que essa união ajudaria os três métodos anteriores, pois cada método possui características que melhorariam os outros.

Em 1994 quando Booch, Rumbaugh e Jacobson realmente se unificaram, eles tinham como objetivo criar uma linguagem que atingisse três metas, sendo elas:

- Uma linguagem para modelagem de sistemas, que abrange desde o conceito até o artefato executável, se utilizando de técnicas orientadas a objeto.
- Uma linguagem para tratar das questões de escala inerente aos sistemas complexos.
- Criar uma linguagem que pudesse ser utilizada tanto por pessoas como por computadores.

Em 1997 foram submetidos os documentos da UML à OMG (*Object Management Group*) que é o órgão que dá as diretrizes da indústria de software, para a avaliação e a proposta de adoção da linguagem como padrão para a arquitetura do desenvolvimento de software orientado a objeto. E foi no dia 17 de Novembro do mesmo ano que a OMG aprovou e instituiu a UML como padrão e se responsabilizou pelas revisões da mesma através da RTF (*Revision Task Force*). Após isto a UML, vem sendo atualizada, e atualmente encontra-se na versão 2.4.1.

2.2 Visão geral

A UML é uma linguagem bastante robusta e com vários recursos, que nos permite visualizar determinadas partes do sistema por diferentes perspectivas, além de facilitar o desenvolvimento em equipe, pois seus diagramas facilitam a compreensão dos membros sobre a modelagem do sistema.

A UML diferente do que muitos pensam não é usada somente no processo de modelagem de software, ela também se adéqua a inúmeros processos como projetos de hardware, fluxos de trabalho, eletrônica médica entre outros processos.

2.3 Diagramas da UML

A UML é uma linguagem de notação gráfica utilizada para construir, especificar, documentar e visualizar os artefatos de um sistema. Segundo [13], a UML é composta por treze diagramas, sendo nove considerados principais. Estes são categorizados em diagramas de comportamento e estruturais. Na Figura 2.1 podemos ver a estrutura da UML.

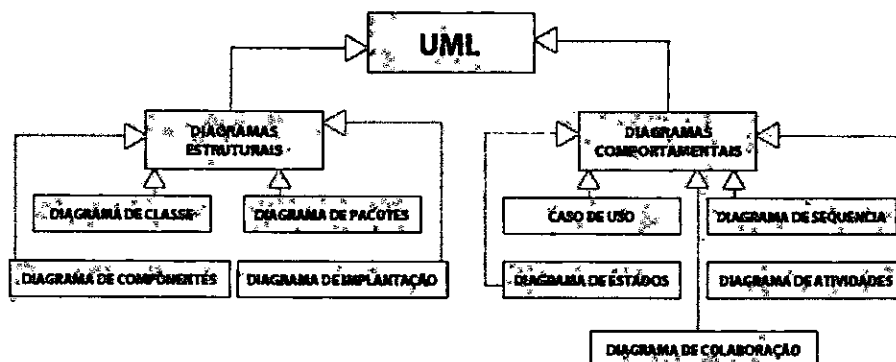


Figura 2.1 – Hierarquia dos diagramas principais da UML – Fonte : o autor.

A seguir apresentaremos um breve resumo sobre cada um dos principais diagramas que compõem a UML.

2.3.1 Diagrama de Classe

O Diagrama de classe demonstra a estrutura de classes, interfaces e seus relacionamentos entre si. Considerado um dos principais diagramas da UML, o diagrama de classe pode também ser usado como um projeto para o banco de dados do software.

2.3.1 Diagrama de Objetos

O Diagrama de objetos demonstra um conjunto de instâncias de objetos se relacionando em um determinado momento no sistema. Objetos esses que são instâncias das classes descritas no Diagrama de Classe.

2.3.3 Caso de Uso

O Diagrama de caso de uso possibilita uma visão externa do sistema, e permite representarmos como o sistema irá interagir com o mundo exterior.

2.3.4 Diagrama de Sequência

O Diagrama de Sequência mostra um conjunto de objetos e a interação entre eles através da troca de mensagens. O Diagrama tem foco no tempo e ordem das trocas de mensagens.

2.3.5 Diagrama de Colaboração

O Diagrama de Colaboração, semelhantemente ao de sequência mostra a colaboração entre os objetos do sistema, porém o diagrama de colaboração foca somente no relacionamento entre os objetos, sem considerar a ordem cronológica das mensagens, como no de sequência.

2.3.6 Diagrama de Estados

O Diagrama de estados é utilizado para descrever o comportamento dos objetos. Ele permite visualizar quais estados uma determinada classe do Diagrama de classes pode possuir.

2.3.7 Diagrama de Atividades

O Diagrama de Atividades é semelhante ao de estados, porém possui um foco em capturar as ações e seus resultados, diferente do de estado que tem foco somente em demonstrar os estados possíveis de uma instância de uma determinada classe.

2.3.8 Diagrama de Componente

O Diagrama de Componente permite visualizar como os componentes do sistema estarão organizados, e como estarão relacionados entre si.

2.3.9 Diagrama de Implantação

O Diagrama de implantação permite descrever o ambiente do software, no qual podemos mostrar os componentes de hardware e software.

3 DIAGRAMA DE CLASSE

O Diagrama de classe pertence ao grupo de diagramas estáticos da UML, ou seja, permite representar uma visão estática do sistema. Ele pode ser usado para documentar, descrever e visualizar diferentes aspectos do sistema, além de permitir a descrição de um código executável do software.

O grande objetivo deste diagrama é permitir a modelagem da visão estática do sistema, e tem como diferencial o fato de ser o único diagrama da UML no qual se pode mapear diretamente uma linguagem orientada a objeto, o que faz com que esse diagrama se torne fortemente usado no momento da construção do software.

Este diagrama é a base para a maioria dos sistemas orientados a objeto, Nele podemos especificar todas as classes do sistema, assim como as interfaces, e como elas se comunicam. Os elementos que compõem esse diagrama são as classes, interfaces, dependência, generalização, e associação.

A seguir apresentaremos um resumo sobre qual a função de cada uma dos componentes que fazem parte do diagrama de classes, assim como a sua representação gráfica dentro do diagrama.

3.1 Classe

A classe define o modelo de um objeto dentro do sistema, descrevendo suas características (Atributos) e suas ações (Métodos).

A classe é graficamente representada por um retângulo dividido em três seções, sendo a primeira onde estará o nome da classe, a segunda onde serão listados os atributos da classe, e na terceira seção onde serão exibidos os métodos da classe. Em alguns casos, dependendo da necessidade do analista, podem se omitir as seções de atributos e a dos métodos, ou optar por apenas listar os mais importantes em cada um dos setores. Na figura 3.1 temos a representação gráfica de uma classe.

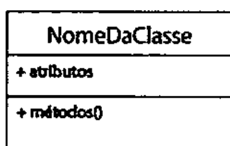


Figura 3.1 – Representação gráfica de uma classe. Fonte: o autor

Um tipo diferente de classe que existe são as classes abstratas, que são classes que servirão apenas de modelos para outras, sendo assim não podem ser instanciadas, podem simplesmente serem herdadas por outras classes.

Graficamente uma classe abstrata é semelhante a uma classe concreta apresentada anteriormente, sendo a única diferença o fato de a fonte do nome da classe está em itálico, como mostrado na figura 3.2.

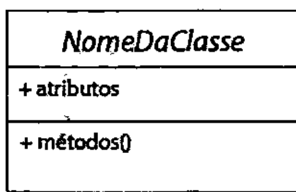


Figura 3.2 – Representação gráfica de uma classe abstrata. Fonte: o autor

3.2 Interfaces

Interfaces podem definir um conjunto de métodos, semelhantemente a uma classe, porém nas interfaces esses métodos somente são declarados, não podendo ser implementados, sendo essa tarefa incumbida à classe que for implementar a interface.

Pode-se dizer que uma interface funciona como um contrato com a classe, no qual ela estará obrigando qualquer classe que a implemente a estar desenvolvendo os métodos que ela declara.

Graficamente uma interface também se assemelha a uma classe, porém ela possui o texto <<interface>> antes do nome, como apresentado na figura 3.3.

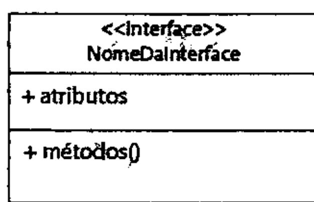


Figura 3.3 – Representação de uma interface. Fonte: o autor.

Outra forma de representar uma interface no diagrama de classe é através de um círculo com o nome da interface ligado por uma linha a uma classe, representado que aquela classe implementa a interface, assim como representado na figura 3.4.

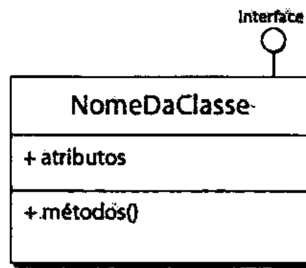


Figura 3.4- Representação de uma interface. Fonte: o autor.

3.3 Dependência

Uma dependência determina uma relação entre duas classes, uma dependente e outra independente, no qual a classe dependente só poderá existir em conjunto com a classe independente.

Graficamente ela é representada por uma linha tracejada que liga as duas classes, com uma seta na ponta do lado da classe independente, e com um rótulo sobre a linha, como demonstrado na figura 3.5.

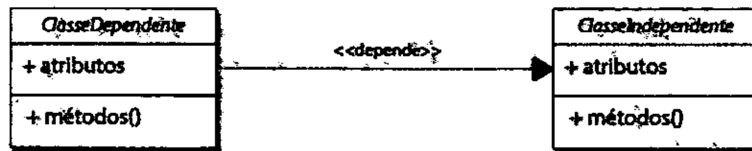


Figura 3.5 – Relação de Dependência entre classes. Fonte: o autor.

3.4 Generalização

É a relação entre duas classes, uma classe pai e outra classe filha, no qual a classe filha irá herdar todas as características da classe pai.

A generalização é graficamente representada por uma linha ligando duas classes, com uma seta branca no lado da classe pai, como exibido na figura 3.6.

Figura 3.6- Herança de classe. Fonte: o autor.

3.5 Associação

É uma relação simples entre duas classes, no qual representa que instâncias daquelas classes estão se relacionando no sistema.

As associações possuem dois tipos principais que são: agregação e composição, ambas detalhadas a seguir.

3.5.1 Agregação

Agregação representa uma associação entre um "objeto-todo" e um "objeto-parte" no qual o "objeto-parte" faz parte do todo, porém mesmo esses objetos se relacionando, eles conseguiriam existir um independente do outro.

Graficamente a agregação é representada por uma linha ligando as duas classes e um diamante branco na ponta do lado da classe que representa o "objeto-todo". Uma representação gráfica da agregação é demonstrada na figura 3.7.

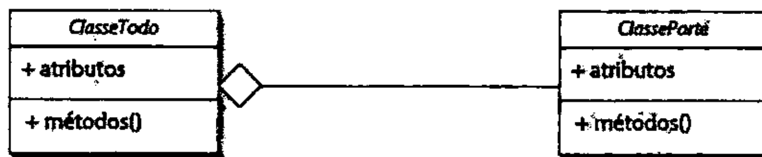


Figura 3.7 – Agregação. Fonte: o autor.

3.5.2 Composição

Na Composição o relacionamento entre os objetos é mais forte, pois ela determina que o "objeto-parte" não pode existir sem que haja um "objeto-todo" o qual ela faça parte.

Graficamente a composição é representada por uma linha ligando as duas classes e um diamante preto na ponta do lado da classe que representa o "objeto-todo". Uma representação gráfica de uma composição é demonstrada na figura 3.8.

Figura 3.8 – Composição. Fonte: o autor.

4 MÉTODOS FORMAIS

4.1História

Com o grande crescimento da computação na década de 70, a quantidade e complexidade dos softwares começaram a aumentar expressivamente. Foi então que começou a dificuldade em garantir a qualidade dos softwares, pois o fato de serem grandes e complexos tornavam as inspeções informais cada vez mais obsoletas, pois já não se conseguia garantir a confiabilidade do sistema.

O fato destes métodos informais de verificação serem imprecisos, em alguns casos incompletos, fizeram com que se tornasse cada vez mais necessário algo no processo de desenvolvimento que reduzisse os erros e garantisse a qualidade do software ao final do projeto.

Foi então que começaram a surgir os métodos formais, que são métodos baseados em formalismos matemáticos e teoria de conjuntos, que podem ser usados para descrever propriedades do sistema. Estes métodos têm como objetivo garantir maiores níveis de qualidade ao software, além de aumentar a confiabilidade na correção do sistema, se utilizando de provas formais, testes e refinamentos.

4.2 O que fazer com métodos formais.

Estes métodos formais podem ser usados para especificar, desenvolver e verificar os sistemas de uma maneira mais sistemática, tornando-se parte fundamental em todo o processo de desenvolvimento do software.

4.2.1 Especificação formal

Permite a descrição dos detalhes do software a ser desenvolvido. Esta especificação pode posteriormente ser usada como base para o desenvolvimento e verificação do sistema.

4.2.2 Desenvolvimento

Depois de feita a especificação formal, ela pode servir de base para o desenvolvimento. Este será feito com todas as regras descritas na especificação, garantindo assim que suas operações estejam realizando suas funções de forma correta.

4.2.3 Verificação

Após o término do desenvolvimento do sistema, a mesma especificação formal pode ser usada como base na verificação do software. Aqui serão desenvolvidos os testes para validar e verificar as propriedades e funcionalidades do sistema.

4.3 Taxonomia

Os métodos formais podem ser categorizados em três níveis distintos, sendo eles:

4.3.1 Nível 0

É feita uma descrição formal do sistema, que é usada como base no momento da implementação do software. É a forma mais simples de utilizar métodos formais, e sua vantagem é o baixo-custo em relação aos outros níveis.

4.3.2 Nível 1

Neste nível são feitas as verificações, a fim de desenvolver um software de maneira mais formal. Permite que através da especificação se possa verificar as propriedades do software. Este nível se torna a melhor escolha para os sistemas de alta-integridade que exigem bastante segurança.

4.3.3 Nível 2

São realizados testes completos e automatizados no sistema, se utilizando de teoremas. Esse nível se torna extremamente trabalhoso, sendo viável somente em sistemas onde o custo por erro possua um valor muito alto.

4.4 Como Funciona

Como dito anteriormente, os métodos formais se baseiam na matemática, que é uma ciência exata, para garantir o funcionamento correto das funcionalidades do software. Para realizar essas provas matemáticas, ele se utiliza de elementos comuns da matemática como teoria de conjuntos e cálculo de predicados, como lógica booleana, indução e quantificadores, além de se utilizar de algumas estruturas matemáticas como grafos e árvores. Porém, os métodos formais se baseiam em três conceitos formais fundamentais, que são os dados invariantes, os estados e as operações.

4.4.1 Dados invariantes

Representa uma propriedade do sistema que deve ser válida em qualquer estado do sistema. Essas condições são verificadas a cada execução de um método dentro do sistema. Um exemplo seria um saldo de uma conta de banco que não pode assumir valores negativos, então em qualquer operação que o banco realize o sistema teria tem que estar fazendo esta verificação.

4.4.2 Estados

Representa uma propriedade do sistema que possuem um número de estados finitos, ou seja, só podem assumir o valor de um determinado conjunto de dados. Um exemplo seria um livro de uma biblioteca que somente pode assumir os estados de disponível, emprestado e reservado.

4.4.3 Operações

As operações são divididas em pré-condições e pós-condições, que são funções de verificação realizadas antes e após a execução do método respectivamente. A pré-condição verifica se os parâmetros estão dentro dos conformes, como por exemplo, verificar se o valor de um saque bancário é maior que zero, antes de uma operação saque. E a pós-condição verifica se o método realmente executou sua função corretamente como, por exemplo, analisar se o saldo de uma conta bancária está igual ao seu saldo anterior menos o valor do saque realizado.

4.5 Visão geral

Percebemos que os métodos formais veem com a proposta de auxiliar no processo de modelagem e desenvolvimento do software, a fim de garantir através de conceitos matemáticos, a integridade do sistema, a confiabilidade de suas operações, a redução do período de testes, além de um produto final com a quantidade menor possível de erros.

Dentre os muitos métodos formais que surgiram, alguns atualmente possuem grandes destaques como B [6], Event-B [7], VDM [8], e Z [9]. Nesse trabalho utilizaremos a linguagem Event-B, como linguagem alvo para a transformação do diagrama de classe em notação matemática a fim de prover um mecanismo de verificação para o mesmo. Mais detalhes sobre essa linguagem formal serão apresentados na próxima seção.

4.6 Event-B

Event-B é uma linguagem formal que pode ser usada na modelagem e análise de sistemas de alto nível, ele assim como a maioria dos métodos formais, se baseia na teoria de conjuntos e lógica de predicados, para através desses conceitos matemáticos, conseguir garantir a consistência do sistema.

Esta linguagem formal é uma evolução do Método-B[16], que foi desenvolvido por Jean-Raymond Abrial em 1980, na França e Reino Unido. O método B é um método de modelagem modular, ou seja, se subdivide em módulos onde cada módulo irá representar um mesmo objeto em cada nível de abstração do sistema. Os níveis do método B seguem a ordem crescente de abstração, ou seja, vão do nível mais abstrato até o mais concreto, sendo eles *Machines*, *Refinements* e *Implementation.*

O Event-B permite a criação de modelos (*machines*), onde esses mesmos serão definidos através de um nome, um conjunto de variáveis que definirão os estados, invariantes, eventos e um contexto. Cada um desses elementos serão explicados e exemplificados a seguir.

4.6.1 Nome

Funciona simplesmente como identificação para o modelo, como pode ser visto na Figura 4.1.

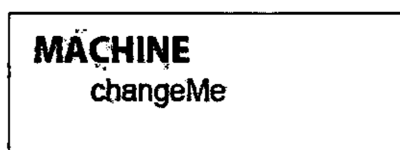


Figura 4.1 – Modelo Event-B (*Machine*) com o nome "*changeMe*". Fonte: o autor.

4.6.2 Invariantes

São regras impostas ao software, a fim de restringir valores à uma determinada variável do sistema, tornando assim mais segura a transição para um estado válido. Essas regras serão sempre verificadas na ocorrência de qualquer evento, para validar o estado dos objetos. Um exemplo de invariante (*Invariant*) pode ser visto na figura 4.2.

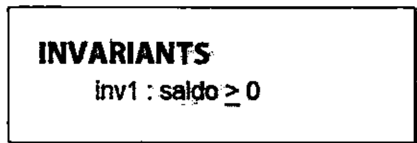


Figura 4.2 – Invariante (*Invariant*), representando a regra de que a propriedade saldo nunca pode ser menor que zero. Fonte: o autor.

4.6.3 Eventos

São métodos que podem alterar o estado dos objetos. Neles são feitas verificações como dito anteriormente, e a alteração no estado do objeto só é realizada após as regras serem satisfeitas. Esses eventos são compostos por um nome, por guardas e por ações. Um exemplo de evento pode ser visualizado na figura 4.3.

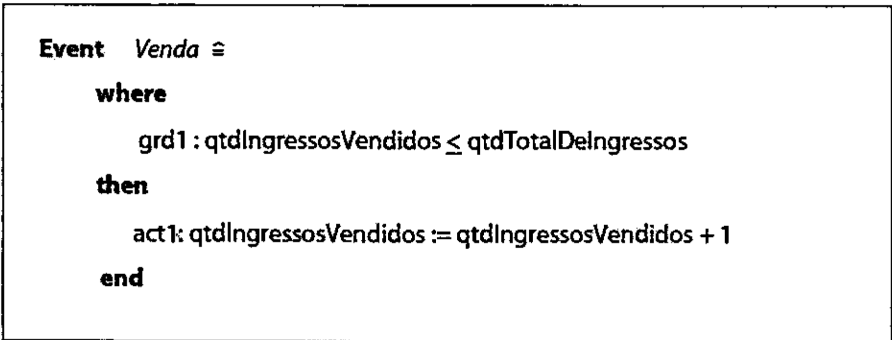


Figura 4.3 - Evento (*Event*), representando uma venda de ingresso. Fonte: o autor.

4.6.3.1 Nome

Utilizado simplesmente para identificar o evento.

4.6.3.2 Guardas

Guardas são condições pré-estabelecidas em um evento, onde o evento somente poderá executar suas ações e alterar o estado dos objetos caso estas condições sejam satisfeitas.

4.6.3.3 Ações

São ações que definem os estados dos objetos, podendo somente ser executadas mediante as condições de guardas serem satisfeitas.

Para realizar a troca de estados, ele utiliza uma técnica chamada de substituições generalizadas, onde ele realiza substituições nas variáveis de estado presente no modelo. Essas substituições podem se associar a um "*before-after predicated*", onde ele relaciona o valor do estado anterior do evento ao valor do estado após a execução de evento.

4.6.4 Contextos

No Event-B além dos modelos com as características apresentadas acima, também existem os contextos, que possibilitam definir a forma de como um modelo irá ser parametrizado para ser instanciado.

O contexto é definido por um nome, uma lista de conjuntos globais, uma lista de constantes e o nome das propriedades. Estes contextos podem também serem associados a um modelo, fazendo assim que seus conjuntos globais e constantes possam ser vistos pelo modelo o qual ele está associado. Na figura 4.4 podemos ver essa relação.

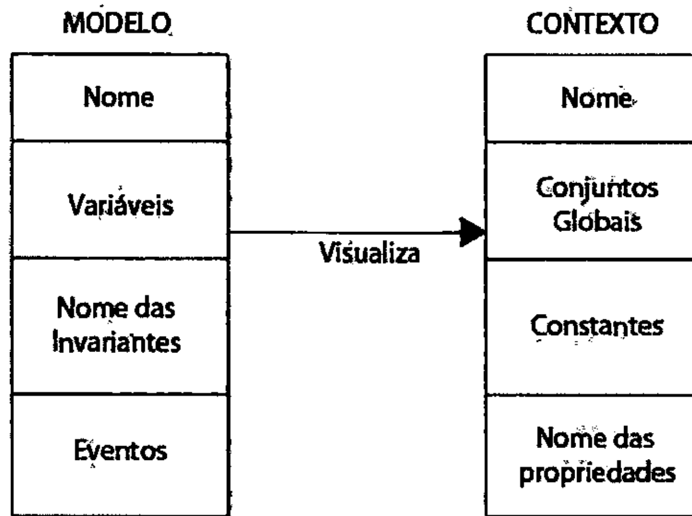


Figura 4.4 - Relacionamento de um Modelo (*machine*) com um contexto (*context*). Fonte: o autor.

A ferramenta Rodin [17] é um ambiente de desenvolvimento open-source, baseado na plataforma Eclipse[18], que provê suporte para a linguagem Event-B. Atualmente ela se encontra na versão 2.8, e já fornece mecanismos para utilização de provas matemáticas, análises sintáticas, animações, e suporte a interfaces gráficas.

5.0 GMF E QVT

Este trabalho tem como objetivo desenvolver uma ferramenta CASE para que o desenvolvedor possa criar um diagrama de classe, além de executar as regras de transformação do mesmo para Event-B a fim de prover um mecanismo de verificação formal. Essa ferramenta irá funcionar sob a plataforma Rodin, que por sua vez é baseada no Eclipse, uma das IDEs mais bem consolidadas na área de desenvolvimento de software. A seguir serão apresentadas as tecnologias usadas para o desenvolvimento deste plugin.

5.1 Graphical Modeling Framework (GMF)

É um framework que permite o desenvolvimento de editores gráficos dentro da plataforma Eclipse, a partir de um modelo de domínio. Ele é o resultado de uma fusão de dois outros frameworks, o Eclipse Modeling Framework (EMF)[19] que é um framework de modelagem que permite o desenvolvimento de um modelo e criação a automática de códigos a partir dele, e o Graphical Editing Framework (GEF)[20] que é um framework usado para criação de editores gráficos genéricos.

O projeto GMF é composto por um conjunto de modelos que devem ser feitos para a criação de um editor gráfico. São eles:

5.1.1 Modelo de Domínio

É a definição do modelo dos objetos que serão criados no editor. A partir deste modelo, o EMF cria um novo modelo que possibilita a geração do código base. São representados pelos arquivos com extensão ecore e genmodel, que é o modelo gerado pelo EMF a partir do ecore.

5.1.2 Paleta

Permite a definição dos links que serão apresentados na paleta de componentes, assim como os menus e barra de ferramentas do editor gráfico. É representado pelo arquivo com extensão gmftoo.

5.1.3 Gráfico

Possibilita a definição gráfica de todos os componentes (linhas, quadrados, títulos) que irão compor o editor gráfico, assim como o comportamento e funções destes elementos. É representado pelo arquivo com extensão gmfgrapf.

5.1.4 Mapeamento

É onde é realizado o sincronismo entre o modelo de domínio, a paleta e os componentes gráficos. É representado pelo arquivo com extensão gmfmap.

5.1.5 Gerador de Código

É o responsável pela geração do código do editor gráfico. Ele é um arquivo gerado a partir do arquivo de mapeamento (gmfmap) já criado, possibilitando ainda modificações antes da geração do código final.

O GMF se mostra altamente poderoso com um vasto conjunto de funções gráficas, além de oferecer inúmeros recursos avançados para o desenvolvedor. Ele ainda possui um *dashboard* em seu plugin para Eclipse que possibilita realizar de forma gráfica a criação e sincronismo dos arquivos. Este *dashboard* pode ser visualizado na figura 5.1.

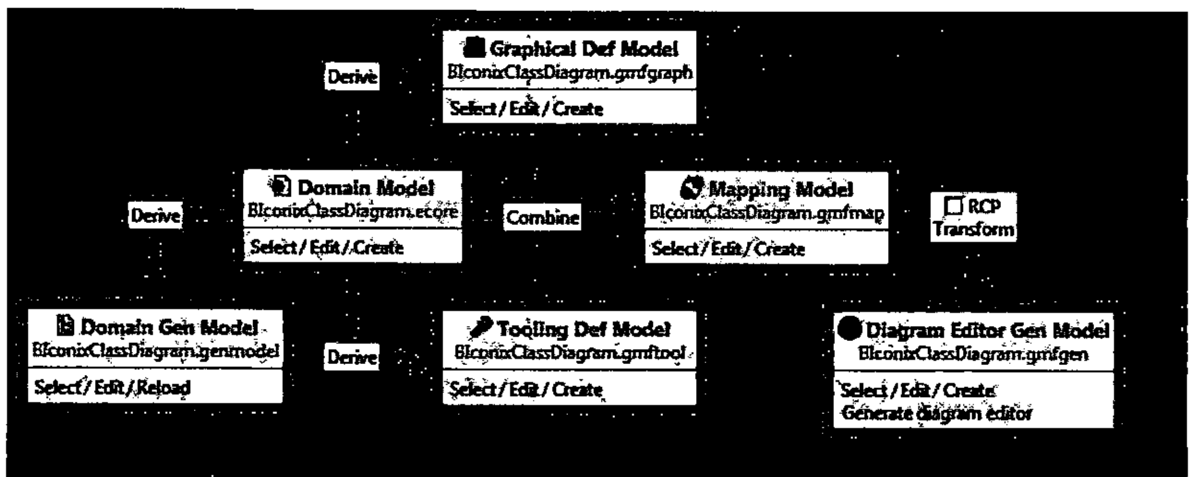


Figura 5.1 - Dashboard do GMF. Fonte: o autor.

5.2 QVT (Queries/Views/Transformations)

QVT (Queries/Views/Transformations)[21] é uma linguagem de transformação de modelos, criado no início do século 21 pela OMG (*Object Management Group*). Ela veio com a proposta da criação de um padrão de transformação compatível com alguns outros padrões da OMG como UML, OCL, MOF e outros. Essa linguagem é composta pro três eixos. Sendo eles:

5.2.1 *Query*

Recebe-se um modelo como entrada e seleciona elementos específicos deste modelo.

5.2.2 *View*

São modelos que derivam de outros modelos, sendo o resultado de uma consulta em outro modelo.

5.2.3 *Transformation*

Recebe-se um modelo como parâmetro de entrada e o atualiza, ou então se cria outro modelo como saída.

O QVT possui uma arquitetura híbrida, se dividindo em duas partes. A primeira é a parte imperativa que se divide nas camadas "*Relations*" e "*Core*", sendo responsáveis pelo processo de transformação. A segunda parte é a parte declarativa, que é composta pelas camadas "*operational mappings*" e "*black box*", sendo responsável pelo mapeamento. Podemos ver um modelo da estrutura do QVT na figura 5.2.

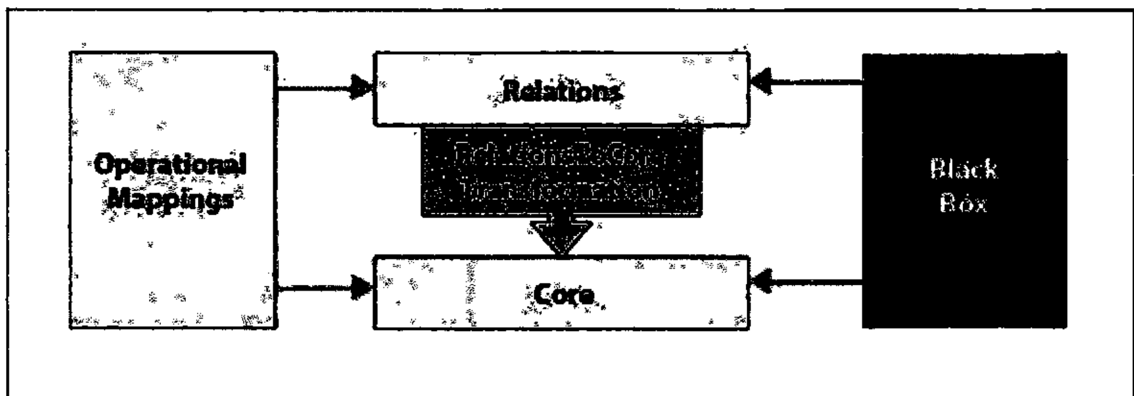


Figura 5.2 - Arquitetura do QVT. Fonte: o autor.

6 FORMALIZAÇÃO DO DIAGRAMA DE CLASSE

A aplicação desenvolvida é uma ferramenta CASE, utilizada como plugin para a plataforma Rodin, que permitirá ao usuário desenhar, salvar, editar e excluir diagramas de classe, além da execução das regras de transformação deste mesmo diagrama em uma especificação Event-B.

Para o desenvolvimento deste software foram utilizadas as tecnologias GMF e QVT para a construção da ferramenta de desenho e para a codificação das regras de transformação respectivamente. A seguir, mostraremos os passos seguidos para o desenvolvimento do plugin.

6.1 Modelagem

O desenvolvimento do software começou com a modelagem do sistema, onde definimos os elementos que estarão presentes no diagrama. É importante saber que neste plugin foi utilizada a proposta de modelagem do diagrama de classe do Iconix, uma vez que esse trabalho faz parte do projeto europeu Advance[22], um processo cujo objetivo é fundir o Iconix com o Event-B. Há portanto algumas suaves diferenças para o padrão do diagrama de classe da OMG, como a incorporação da associação do tipo agregação e remoção de alguns outros componentes. Neste padrão do processo Iconix o diagrama de classe é representado por apenas quatro elementos (classes, atributos, operações, associações), tornando assim o diagrama mais simplificado.

Outro ponto a se destacar é o fato de apesar da metodologia Iconix[3] diferenciar o modelo de domínio do diagrama de classes, em nosso caso ambos possuem a mesma estrutura com os mesmos componentes, possibilitando assim o uso do diagrama de classe do sistema no próprio modelo de domínio.

Na figura 6.1, é possível visualizar o meta-modelo do diagrama de classes conforme o padrão do Iconix, sendo que este é o meta-modelo usado como base para nosso sistema. Nele adicionamos o prefixo "BIconix" no nome de cada uma das classes para evitar classes homônimas com as do padrão OMG. Além disso, na figura 6.1 podemos perceber uma coloração diferente na classe BIconixOperation, que se deve ao fato de esta classe não ser diretamente criada no diagrama de classe (importadas do diagrama de sequência). Como este trabalho está se tratando do diagrama de classe individualmente, ainda não incorporaremos esta classe em nosso sistema. Na próxima subseção serão apresentados mais detalhes sobre

cada um dos componentes deste diagrama, tais como descrição, generalizações, atributos e associações.

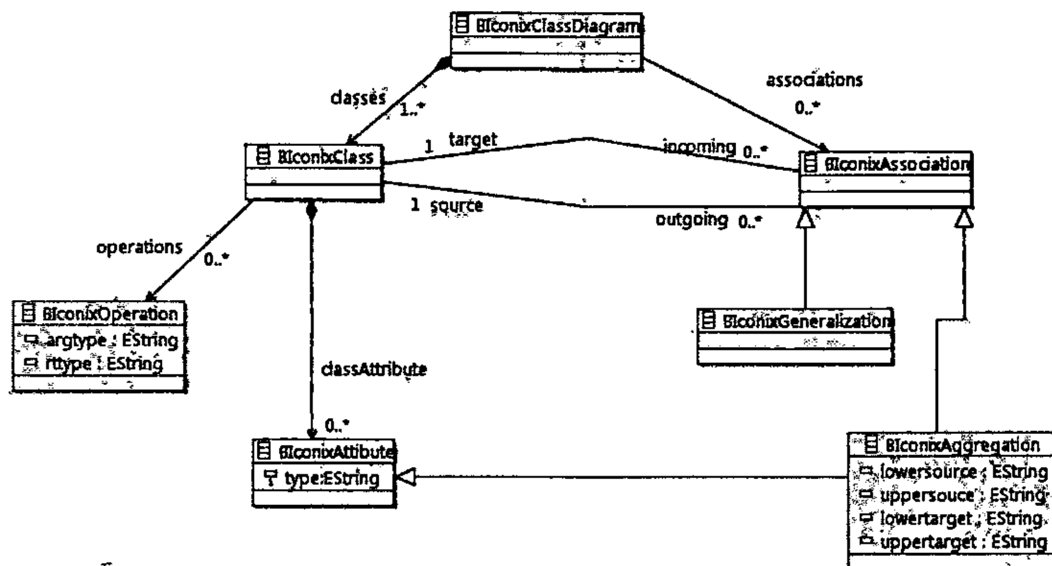


Figura 6.1 - Meta-modelo do Diagrama de Classe segundo padrão Iconix. Fonte: o autor.

6.2 Descrição das meta-classes

6.2.1 BIconixClassDiagram

A classe BIconixClassDiagram permite a criação do diagrama de classe como um todo, e pode ser considerado o modelo de domínio dentro do processo do Iconix. É nessa classe onde ficam guardadas as classes, as associações e todas outras propriedades do diagrama.

Esta classe possui duas associações, sendo elas:

- classes : BIconixClass[1..*], representa a associação que guarda as classes criadas dentro do diagrama, fazendo-se necessária a existência de pelo menos uma classe.
- associations : BIconixClassAssociation[0..*], guarda as associações feitas entre classes dentro do diagrama.

6.2.2 BIconixClass

A classe BIconixClass representa uma classe dentro do diagrama no Iconix. Os objetos desta classe devem obrigatoriamente possuir um nome, que embora a propriedade não seja demonstrada no diagrama, será herdada de classes do Event-B do qual BIconixClass é filho.

Esta classe possui quatro associações, sendo elas:

- *outgoing*: BIconixAssociation[0..*], armazena as associações (agregação e generalização) que saem da classe. Esta relação é uma relação de contrapartida à relação "source" da classe BIconixClassAssociation, ou seja, toda vez que uma classe for definida na propriedade source de uma associação, automaticamente esta associação é adicionada a propriedade "*outgoing*" da classe.
- *incoming*: BIconixAssociation[0..*], armazena as associações (agregação e generalização) que chegam à classe. Esta relação é uma relação de contrapartida à relação "*target*" da classe BIconixClassAssociation, ou seja, toda vez que uma classe for definida na propriedade "*target*" de uma associação, automaticamente esta associação é adicionada a propriedade "*incoming*" da classe.
- *classAttributes*: BIconixAttribute[0..*], guarda todos os atributos criados na classe, inclusive os criados a partir das associações.
- *operations*: BIconixClassOperation[0..*], guarda os métodos existentes na classes. Porém, como citado anteriormente, esta associação não será utilizada agora devido a classe BIconixClassOperation ser dependente de outro diagrama no processo Iconix.

Uma restrição a esta classe, é que as associações "*outgoing*" e "*incoming*" devem ser conjuntos disjuntos, ou seja, eles não podem conter elementos em comum.

6.2.3 BIconixAttribute

A classe BIconixClassAttribute representa a um atributo que pode ser criado dentro de uma BIconixClass. Objetos instanciados desta classe deverão conter possuir

obrigatoriamente um nome e um tipo, sendo que na propriedade "*type*" só poderão ser usados os tipos primitivos (inteiro ou booleano), ou uma BIconixClass.

6.2.4 BIconixAssociation

A classe BIconixAssociation representa a associação entre duas classes. Ela é uma classe abstrata que não pode ser instanciada, servindo apenas para ser especializada pelas classes BIconixGeneralization e BIconixAggregation.

Esta classe possui duas associaçõe, sendo elas:

- *source*: BIconixClass[1], armazena a classe de onde parte a associação.
- *target*: BiconixClass[1], guarda a classe "atingida" pela associação.

A única restrição a esta classe é que uma associação nunca pode possuir um "*source*" igual ao seu "*target*".

6.2.5 BIconixGeneralization

A classe BIconixGeneralization representa um tipo de associação entre duas classes, definindo que a classe "*source*" herdará as propriedades da classe "*target*". Esta classe é uma classe herdeira da classe BIconixAssociation, portanto já possui as propriedades "*source*" e "*target*" definidas.

Esta generalização representa uma herança simples, ou seja, uma classe só poderá possuir no máximo uma associação deste tipo, e também vale lembrar que a hierarquia das generalizações devem ser acíclicas.

6.2.6 BIconixAggregation

A classe BiconixAggregation representa uma associação de agregação entre duas classes, onde a classe agregada faz parte de um todo para formar a classe que a agrega. Esta classe é herdeira da classe BIconixAssociation, e da classe BIconixAttribute, pois além de representar uma associação entre duas classes, ela também se torna um atributo (com o tipo da classe agregadora) dentro da classe agregada.

Além dos atributos herdados de BIconixAssociation e BIconixAttribute, esta classe possui atributos referentes à cardinalidade da relação, sendo eles:

- *lowersource*: limite inferior da multiplicidade referente à classe source da associação.
- *uppersource*: limite superior da multiplicidade referente à classe source da associação.
- *lowertarget*: limite inferior da multiplicidade referente à classe target da associação.
- *uppertarget*: limite superior da multiplicidade referente à classe target da associação.

Este tipo de associação deve possuir um nome, que também será o nome do atributo na classe agregada. Seus atributos "*lowersource*" e "*lowertarget*" só podem assumir os valores 0 ou 1, enquanto os atributos "*uppersource*" e "*uppertarget*" só podem ter os valores 1 ou n.

6.3 Geração do Editor para o Diagrama de Classe

Após o desenho do meta-modelo, utilizamos o EMF para realizar a geração do arquivo "genmodel" (*Genereator Model*) baseado no modelo recém-criado. Através dele poderemos realizar a geração automática do código das classes pertencentes ao modelo e ainda cria um editor para este mesmo modelo.

Com os códigos das classes geradas, começamos a criar parte gráfica do editor, assim como também a parte gráfica do próprio diagrama. Para auxiliar este desenvolvimento utilizamos o GMF *dashboard* do Eclipse, apresentado na figura 5.1, que permite vincularmos, derivarmos e criarmos os arquivos gmf de forma gráfica, sem precisar codificá-los.

Através desse dashboard podemos derivar o modelo para a criação do arquivo gmftool que representa a paleta e os menus do editor gráfico. Após a criação da paleta, derivamos o modelo para a criação do arquivo gmfgrapf, que representa a definição gráfica e comportamental de cada componente (classes, associações e atributos) do editor. Com estes arquivos criados, é necessária a combinação dos dois, a fim de mapeá-los especificando quais componentes gráficos contidos no gmfgraph serão criados pelos componentes da paleta e pelos menus do gmftool, gerando assim o arquivo gmfmap, onde estará contido este mapeamento.

Com o mapeamento criado, o GMF permite transformá-lo no arquivo gmfgen, que será o arquivo próprio para a geração automática do código das classes que irão formar o editor gráfico.

Com o código criado, o editor já se encontra funcionando, permitindo a criação, edição, exclusão e remoção de diagramas de classes. Na figura 6.2 vemos o editor gráfico criado, em funcionamento, sendo executado dentro da plataforma Rodin.

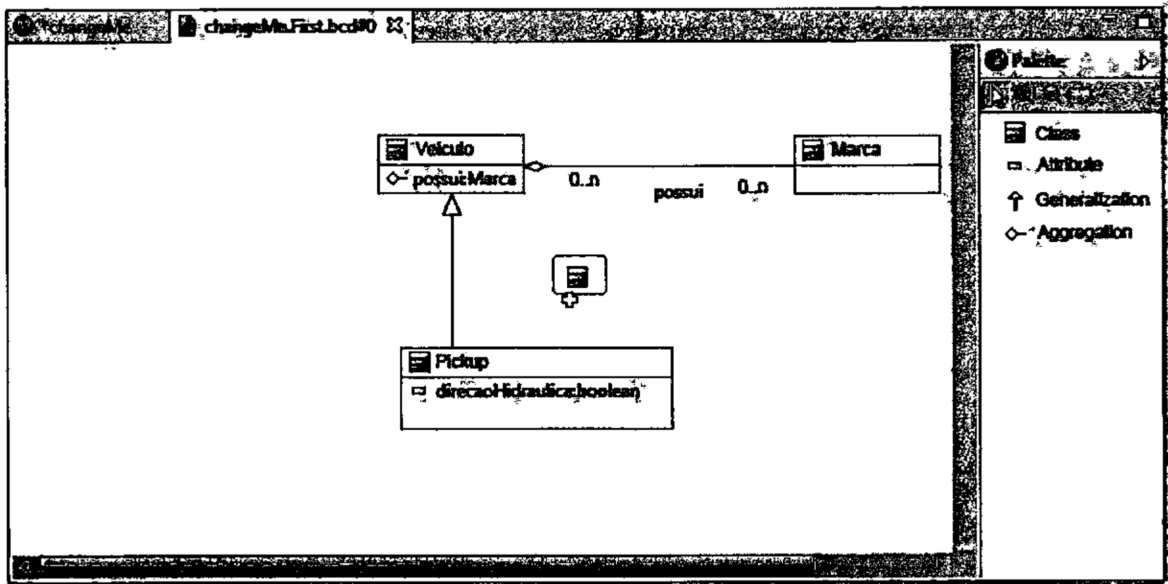


Figura 6.2 - Editor gráfico para o diagrama de classes. Fonte: o autor.

Com o editor gráfico funcionando, é hora de criarmos os comandos para transformar o diagrama de classe criado em código Event-B. Na próxima subseção apresentaremos como criamos as regras de transformação, para validação, e como aplicarmos elas no diagrama criado.

6.4 Regras de Transformação

Com a ferramenta de desenho do diagrama de classes construída, cada diagrama desenhado resultará em uma instância do meta-modelo do diagrama de classe, que será usado como entrada para a transformação. O meta-modelo do Event-B que será o meta-modelo de saída, já foi desenvolvido por [23] e simplesmente foi anexado a este projeto. O meta-modelo do Event-B pode ser visualizado no anexo A deste trabalho.

Durante esta etapa iremos definir as regras de transformação que mapeiam os elementos destas linguagens. Para isso usaremos a abordagem QVT, que é uma linguagem baseada em OCL e padronizada pela OMG, como ferramenta de descrição das regras.

Para a criação das regras é necessário a criação de um projeto qvt, um arquivo de mapeamento, além de uma classe Java para realizar a ligação entre as regras desenvolvidas e a instância do diagrama de classe criado.

A seguir, iremos apresentar as regras em modo descritivo, onde apresentaremos o mapeamento de cada componente do diagrama de classe para os componentes equivalentes do Event-B. Para um maior aprofundamento, o código qvt das regras de transformação se encontra no apêndice A deste trabalho.

6.4.1 BIconixClassDiagram

Cada intância da meta-classe BIconixClassDiagram será mapeada para uma meta-classe Context do Event-B, recebendo o mesmo nome do diagrama.

Exemplo

Quando criado um diagrama classe com o nome "Concessionaria", deve-se ter um contexto como o apresentado na figura 9.1.

```
CONTEXT Concessionaria
END
```

Figura 6.3 – Contexto referente ao diagrama de classe com nome "Concessionaria". Fonte: o autor.

6.4.2 BIconixClass

A meta-classe BIconixClass irá gerar cinco mapeamentos no Event-B, o primeiro gerado no Contexto criado pela instância da meta-classe BIconixClassDiagram o qual pertence, e os outros quatro na Machine.

O primeiro elemento que será gerado no Contexto, será uma intância de um conjunto global (CarrierSet), seguindo o seguinte padrão de sintaxe : {BiconixClass.name}_SET.

Exemplo

Quando criado uma classe com o nome "Veiculo", deve-se ter um conjunto global (CarrierSet) como o apresentado na figura 6.4.

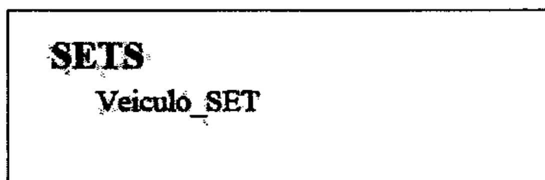


Figura 6.4 – CarrierSet criado a partir de um classe com nome "Veiculo". Fonte: o autor.

Os próximos elementos a serem criados são uma variável (*Variable*), tendo o mesmo nome da classe, e uma invariante (*Invariant*), como sub-conjunto do conjunto global criando no contexto (*Context*) anteriormente.

Exemplo

Quando criado uma classe com o nome "Veiculo", deve-se ter uma variável e uma invariante como o apresentado na figura 6.5.

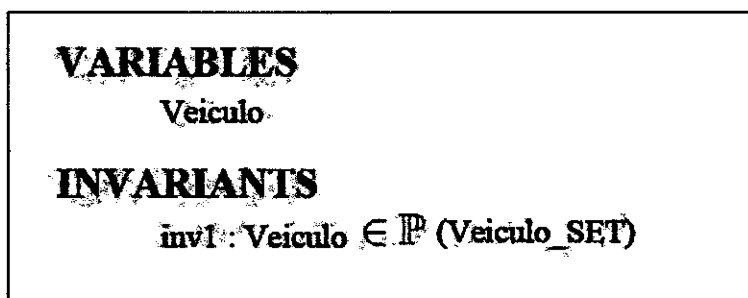


Figura 6.5 – Variável (*variable*) e invariante (*invariant*) criadas a partir de uma classe com nome "Veiculo". Fonte: o autor.

Após a criação da invariante e da variável, são criados dois eventos. Um será o construtor da classe, com um parâmetro (*Parameter*) chamado "*self*", uma guarda (*Guard*) nomeada "self ∈ {BIconixClass.name}_SET\ {BIconixClass.name}", uma ação (*Action*) designada "{BIconixClass.name}:= {BIconixClass.name} ∪ {self}", com o nome seguindo o padrão de nomenclatura: Cons_{BIconixClass.name}". O outro evento será o evento destrutor da classe, ele possuirá uma parâmetro (*Parameter*) chamado "*self*", uma guarda (*Guard*)

designada "self ∈ {BIconixClass.name}", uma ação (*Action*) denominada "{BIconixClass.name} := {BIconixClass.name}\ {self}" e com o nome do evento seguindo o padrão de nomenclatura "Des_{BIconixClass.name}".

Exemplo

Quando uma classe com o nome "Veiculo" é criada, deve-se ter os dois eventos criados como o apresentado na figura 6.6.

```
EVENTS
Event Cons_Veiculo ≘
    any
        self
    where
        grd1 : self ∈ Veiculo_SET\ Veiculo
    then
        act1 : Veiculo := Veiculo ∪ {self}
    end
Event Des_Veiculo ≘
    any
        self
    where
        grd1 : self ∈ Veiculo
    then
        act1 : Veiculo := Veiculo\ {self}
    end
```

Figura 6.6 – Eventos gerados a partir de uma classe nomeada "Veiculo". Fonte: o autor.

6.4.3 BIconixAttribute

Cada instância da meta-classe BIconixAttibute é mapeada como uma instância de uma invariante (*Invariant*), sendo um elemento que pertence à função total cujo o domínio é o próprio nome da classe, e a imagem é o tipo do atributo.

Exemplo

Quando uma classe chamada "Veiculo" possui um atributo comeado "direcaoHidraulica" do tipo booleano, deve-se ter uma invariante como a apresentado na figura 6.7.

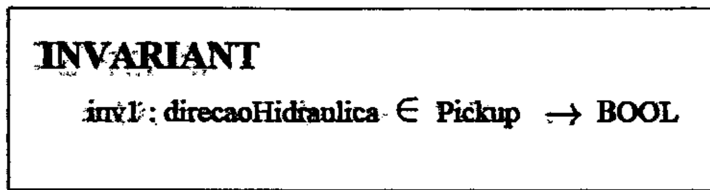


Figura 6.7 – Invariante mapeada referente a um atributo "direcaoHidraulica" do tipo booleana em um uma classe "Veiculo". Fonte: o autor.

6.4.4 BIconixGeneralization

Um instância da meta-classe BIconixGeneralization será mapeada como uma intância de uma invariante (*Invariant*), fazendo com que a classe filha corresponda a um subconjunto da classe pai.

Exemplo

Quando uma classe chamada "Veiculo" possui uma classe especializada denominada "Pickup", então teremos uma invariante como a apresentada na figura 6.8.

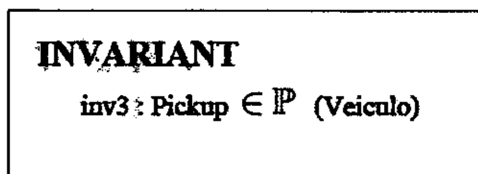


Figura 6.8 – Invariante (*Invariant*) resultado do mapeamento de uma generalização de uma classe "Veiculo" para uma classe "Pickup". Fonte: o autor.

6.4.5 BIconixAggregation

Uma instância da meta-classe BIconixAggregation é mapeada em uma invariante (*Invariant*) correspondendo ao elemento pertencente a uma relação entre as classes associadas. O tipo da relação (sobrejetora, injetora, total, função, etc.) será determinado pela

multiplicidade da agregação, ou seja, será baseado nos valores dos atributos de multiplicidade (*lowersouce*, *uppersource*, *lowertarget*, *uppertarget*). Vale ressaltar que os atributos "*lowersource*" e "*uppersource*" só podem assumir os valores 0 ou 1, enquanto os atributos "*uppersource*" e "*uppertarget*" só podem conter os valores 1 ou n.

Exemplo

Quando uma classe nomeada "Veiculo" possui uma agregação chamada "itens" com uma classe com o nome "Acessorio", e a cardinalidade [0..n] e [0..n] então teremos uma invariante como apresentada na figura 6.9, porém podemos ter várias definições da invariante, dependendo da multiplicidade da agregação. Todas as definições para cada cardinalidade podem ser vista na figura 6.10.

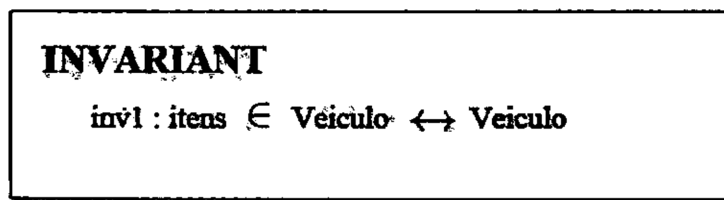


Figura 6.9 – Invatiante (invariant) resutadod e uma agregação entre "Veículo" e "Acessorio" com o nome "itens" e com cardinalidade [0..n] e [0..n] – Fonte: o autor.

lower source	upper source	lower target	upper target	Event-B
0	n	0	n	BIconixAttribute.name $\in$ BIconixClass.name $\leftrightarrow$ BIconixClass.name
0	n	1	n	BIconixAttribute.name $\in$ BIconixClass.name $\trel$ BIconixClass.name
0	n	0	1	BIconixAttribute.name $\in$ BIconixClass.name $\pfun$ BIconixClass.name
0	n	1	1	BIconixAttribute.name $\in$ BIconixClass.name $\rightarrow$ BIconixClass.name
1	n	0	n	BIconixAttribute.name $\in$ BIconixClass.name $\srel$ BIconixClass.name
1	n	1	n	BIconixAttribute.name $\in$ BIconixClass.name $\strel$ BIconixClass.name
1	n	0	1	BIconixAttribute.name $\in$ BIconixClass.name $\psur$ BIconixClass.name
1	n	1	1	BIconixAttribute.name $\in$ BIconixClass.name $\tsur$ BIconixClass.name
0	1	0	n	BIconixAttribute.name $\in$ BIconixClass.name $\leftrightarrow$ BIconixClass.name BIconixAttribute.name~ $\in$ BIconixClass.name $\pfun$ BIconixClass.name
0	1	1	n	BIconixAttribute.name $\in$ BIconixClass.name $\trel$ BIconixClass.name BIconixAttribute.name~ $\in$ BIconixClass.name $\pfun$ BIconixClass.name
0	1	0	1	BIconixAttribute.name $\in$ BIconixClass.name $\pinj$ BIconixClass.name
0	1	1	1	BIconixAttribute.name $\in$ BIconixClass.name $\rightarrowtail$ BIconixClass.name
1	1	0	n	BIconixAttribute.name $\in$ BIconixClass.name $\srel$ BIconixClass.name BIconixAttribute.name~ $\in$ BIconixClass.name $\pfun$ BIconixClass.name
1	1	1	n	BIconixAttribute.name $\in$ BIconixClass.name $\strel$ BIconixClass.name BIconixAttribute.name~ $\in$ BIconixClass.name $\pfun$ BIconixClass.name
1	1	0	1	BIconixAttribute.name $\in$ BIconixClass.name $\psur$ BIconixClass.name BIconixAttribute.name~ $\in$ BIconixClass.name $\pfun$ BIconixClass.name
1	1	1	1	BIconixAttribute.name $\in$ BIconixClass.name $\tbij$ BIconixClass.name

Figura 6.10 – Regra de transformação resultada de um mapeamento de uma agregação. Fonte: o autor.

REFERÊNCIAS

[1] ENGELS, G.; KüSTER, J. M.; GROENWEGEN, L. Consistent interaction of software components. J. Integr. Des. Process Sci., IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 6, p. 2–22, December 2002. ISSN 1092-0617.

[2] KRUCHTEN, P. The Rational Unified Process: An Introduction. 3. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321197704.

[3] ROSENBERG, D.; STEPHENS, M. Use Case Driven Object Modeling with UML: Theory and Practice. [S.l.]: Apress, 2007. ISBN 0321278275.

[4] BECK, K.; ANDRES, C. Extreme Programming Explained : Embrace Change. 2nd. ed. [S.l.]: Addison-Wesley Professional, 2004. Paperback.

[5] SCHWABER, K.; BEEDLE, M. Agile Software Development with Scrum. Upper Saddle River, NJ: Prentice Hall, 2002. ISBN 978-0-13-067634-4.

[6] ABRIAL, J.-R. The B-book: assigning programs to meanings. 1st. ed. New York, NY, USA: Cambridge University Press, 1996. ISBN 0-521-49619-5.

[7] ABRIAL, J.-R. Modeling in Event-B: System and Software Engineering. 1st. ed. New York, NY, USA: Cambridge University Press, 2010. ISBN 0521895561,9780521895569.

[8] BJØRNER, D.; JONES, C. B. The Vienna Development Method: The Meta-Language. Lecture Notes in Computer Science, Springer-Verlag, London, UK, v. 61,1978.

[9] SPIVEY, J. The z notation - a reference manual. Prentice Hall International Series in Computer Science, Prentice Hall, p. I–XI, 1–155, 1989.

[10] MARTINEZ, F. J. L.;Álvarez, A. T. A Precise Approach for the Analysis of the UML Models Consistency. Software Egineering Research Group, Department of Informatics and Systems, University of Murcia (Spain).

[11]BARUZZO, A.;COMINI, M. Static Verification of UML consistency. Dipartimento di Matemstica e Informatica (DIMI), University of Udine.

[12] HNTKOWSKA, Bogumila;HUZAR, Zbigniew;MAGOTT, Jan. Consistency in UML Models. Departament of Computer Science, Wroclaw University of Technology, Wybrzeze Wyspianskiego.

[13] BOOCH, Grady et al. UML : Guia do Usuário, O mais avançado tutorial sobre Unified Modeling Language (UML), elaborado pelos próprios criadores da linguagem. 2 ed. Rio de Janeiro. Campus, 2005.

[14] Object Modeling Technique (OMT). Disponível em <http://www.idi.ntnu.no/grupper/su/publ/html/totland/ch0527.htm>. Acessado em: 05 de Junho de 2013.

[15] Object Management Group. UML 2.4.1 Superstructure. 2010. Disponível em: <http://www.omg.org/spec/UML/2.4.1/>. Acessado em : 05 de Junho de 2013.

[16] ABRIAL, J.-R. The B-book: assigning programs to meanings. 1st. ed. New York, NY, USA: Cambridge University Press, 1996. ISBN 0-521-49619-5.

[17] Plataforma Rodin (Rigorous Open Development Environment for Complex Systems). Disponível em <http://rodin.cs.ncl.ac.uk/>. Acessado em: 14 de Julho de 2013.

[18] Eclipse - The Eclipse Foundation open source community. Disponível em <http://www.eclipse.org/>. Acessado em 14 de Julho de 2013.

[19] EMF (Eclipse Modeling Framework). Disponível em <http://www.eclipse.org/modeling/emf/>. Acessado em: 14 de Julho de 2013.

[20] GEF (Graphical Editing Framework). Disponível em <http://www.eclipse.org/gef/> Acessado em: 05 de Junho de 2013.

[21] GARDNER, Tracy et al. A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard. In: MetaModelling for MDA Workshop. 2003. p. 178-197.

[22] Advanced Design and Verification Environment for Cyber-physical System Engineering. Disponível em <http://www.advance-ict.eu/>. Acessado em 14 de Agosto de 2013.

[23] SNOOK, COLIN, FRITZ, FABIAN AND ILLISAOV, ALEXEI (2010) An EMF Framework for Event-B. At Workshop on Tool Building in Formal Methods - ABZ Conference, Orford, Quebec, Canada.

APÊNDICE A

Neste apêndice encontram-se as regras de transformação que foram implementadas em QVT e descritas no capítulo 6.

```
modeltype eventb uses core('http://emf.eventb.org/models/core');
modeltype classdiagram uses biconixclassdiagram('http://biconixclassdiagram/1.0');

transformation class2eventb(inout inoutMachine:eventb, inout inoutContext:eventb,
in inClassDiagram:classdiagram);

// generator id key
property GENERATOR_ID_KEY : String = "org.eventb.emf.persistence.generator_ID";

//initialisation event name
property INITIALISATION : String = "INITIALISATION";

property B_EQ : String = " = ";
property B_BEQ : String = " \u2254 "; //:=
property B_IN : String = " \u2208 "; //∈
property B_INTEGER : String = " \u2124"; //Z
property B_MINUS : String = " \u2216"; //\
property B_BOOL : String = "BOOL";
property B_TRUE : String = "TRUE";
property B_FALSE : String = "FALSE";
property B_NOT : String = " \u00ac "; // ¬
property B_AND : String = " ∧ ";
property B_OR : String = " ∨ ";
property B_LPAR : String = "(";
property B_RPAR : String = ")";
property B_LBRC : String = "{";
property B_RBRC : String = "}";
property B_COM : String = ", ";
property B_IMPL : String = " \u21d2 "; //=>
property B_INTER : String = " \u2229 "; //
property B_UNION : String = " \u222a "; //
property B_POW : String = "\u2119"; // P
property B_REL : String = " \u2194 "; //<->
property B_TREL : String = " \ue100 "; // <<-->
property B_SURREL : String = " \ue101 "; //G  <-->>
property B_TSURREL : String = " \ue102 "; //<<-->>
property B_PFUN : String = " \u21f8 "; //  -|->
property B_TFUN : String = " \u2192 ";// ->
property B_PINJ : String = " \u2914 "; // >-|->
property B_TINJ : String = " \u21a3 "; // >->
property B_PSUR : String = " \u2900 "; // -|->>
property B_TSUR : String = " \u21a0 ";  //-->>
property B_TBIJ : String = " \u2916 "; // >-->>
property B_CONV : String = "\u223C"; // ~
property B_EMPTYSET : String = "\u2205"; // ∅
property INV : String = "inv";
property GRD : String = "grd";
property ACT : String = "act";
property SELF : String = "self";
property SEQ : Integer = 0;
property SEQ_GRD : Integer = 0;
```

```
property SEQ_ACT : Integer = 0;
main() {
      setupMachineContext(inoutMachine.rootObjects()![eventb::machine::Machine],
      inoutContext.rootObjects()![eventb::context::Context]);
      inoutContext.rootObjects()[eventb::context::Context]->map
classdiagram2eventbContext(inClassDiagram.rootObjects()![classdiagram::BIconixClas
sDiagram]);
      inoutMachine.rootObjects()[eventb::machine::Machine]->map
classdiagram2eventbMachine(inClassDiagram.rootObjects()![classdiagram::BIconixClas
sDiagram]);
}


/*
 * Transforms iumlb to eventb context.
 */
mapping inout eventb::context::Context::classdiagram2eventbContext(in
rootClassDiagram : classdiagram::BIconixClassDiagram) {
      var generatorID : String := rootClassDiagram.extensionId;
      self.sets := rootClassDiagram.map class2sets(generatorID);
}

mapping inout eventb::machine::Machine::classdiagram2eventbMachine(in
rootClassDiagram: classdiagram::BIconixClassDiagram) {
    var generatorID : String := rootClassDiagram.extensionId;

    self.map machine2nonGeneratedMachine(generatorID);
      self.seesNames->includes(rootClassDiagram.name);

    self.variables := rootClassDiagram.map classDiagram2variables(generatorID)->
        union(self.variables->asSequence());
    self.invariants := rootClassDiagram.map classDiagram2invariant(generatorID)->
        union(self.invariants->asSequence());
    self.events := rootClassDiagram.map classDiagram2events(generatorID);
}

//##################################################################################
// VARIABLES
//##################################################################################
mapping classdiagram::BIconixClassDiagram::classDiagram2variables(in generatorID :
String) : Sequence(eventb::machine::Variable) {
    init {
        result := self.classes[classdiagram::BIconixClass].map
class2variables(generatorID);
    }
}

mapping classdiagram::BIconixClass::class2variables(in generatorID : String) :
Sequence(eventb::machine::Variable) {
    init {
        result := self.map class2variable(generatorID)->asSequence()-
>union(self.classAttributes.map attribute2variable(generatorID)->asSequence());
    }
}

// Class to variable
mapping classdiagram::BIconixClass::class2variable(in generatorID : String) :
eventb::machine::Variable {
```

```
    name := self.name;
    generated := true;
    attributes += getGeneratedAttr(generatorID);
}

// Attribute to variable
mapping classdiagram::BIconixAttribute::attribute2variable(in generatorID :
String) : eventb::machine::Variable
 when { not self.oclIsKindOf(BIconixAggregation) }
      {
    name := self.name;
    generated := true;
    attributes += getGeneratedAttr(generatorID);
}

//####################################################################################
// INVARIANTS
//####################################################################################

mapping classdiagram::BIconixClassDiagram::classDiagram2invariant(in generatorID :
String) : Sequence(eventb::machine::Invariant) {
    init {
      result := self.classes[classdiagram::BIconixClass].map
class2typeInvariant(generatorID)->sortedBy(i | i.name)->asSequence()->
        union(self.classes.classAttributes[classdiagram::BIconixAttribute].map
attribute2invariant(generatorID)->asSequence())->
        union(self.associations[classdiagram::BIconixGeneralization].map
generalizationmain2invariant(generatorID)->asSequence());
     }
}

mapping classdiagram::BIconixAttribute::attribute2invariant(in generatorID :
String) : eventb::machine::Invariant
    disjuncts classdiagram::BIconixAttribute::attributeType2Invariant {}
mapping classdiagram::BIconixGeneralization::generalizationmain2invariant(in
generatorID : String) : eventb::machine::Invariant
    disjuncts classdiagram::BIconixGeneralization::generalization2Invariant {}

//Class to invariant
mapping classdiagram::BIconixClass::class2typeInvariant(in generatorID : String) :
Sequence(eventb::machine::Invariant)
{
    init {
        result := getInvariant(getInvariantName(),
        self.name + B_IN + B_POW + B_LPAR + self.name + "_SET" + B_RPAR,
generatorID)->asSequence()->union(
        self.outgoing[classdiagram::BIconixAggregation].map
aggregation2Invariant(self, generatorID)->asSequence())
    }
}

//Attribute to invariant
mapping classdiagram::BIconixAttribute::attributeType2Invariant(in generatorID :
String) : eventb::machine::Invariant
    when { not self.oclIsKindOf(BIconixAggregation) }
{
    init {
        result := getInvariant(getInvariantName(),
```

```
        self.name.trim() + B_IN + self.eContainer().oclAsType(BIconixClass).name +
B_TFUN +  getAttributeType(self), generatorID)
    }
}

//Generalization to Invariant
mapping classdiagram::BIconixGeneralization::generalization2Invariant(in
generatorID : String) : eventb::machine::Invariant{
    init {
        result := getInvariant(getInvariantName(),
        self.source.name + B_IN + B_POW + B_LPAR + self.target.name + B_RPAR,
generatorID)
    }
}

//Aggregation to Invariant
mapping classdiagram::BIconixAggregation::aggregation2Invariant(in clazz :
classdiagram::BIconixClass, in generatorID : String) :
Sequence(eventb::machine::Invariant)
{
    init {
        var invariants : Sequence(eventb::machine::Invariant) := null;
       switch {
        case (cardinality(self,"0","n","0","n"))
           invariants := getInvariant(getInvariantName(), self.name + B_IN +
clazz.name + B_REL + clazz.name, generatorID)->asSequence();
        case (cardinality(self,"0","n","1","n"))
           invariants := getInvariant(getInvariantName(), self.name + B_IN +
clazz.name + B_TREL + clazz.name, generatorID)->asSequence();
        case (cardinality(self,"0","n","0","1"))
           invariants := getInvariant(getInvariantName(), self.name + B_IN +
clazz.name + B_PFUN + clazz.name, generatorID)->asSequence();
        case (cardinality(self,"0","n","1","1"))
           invariants := getInvariant(getInvariantName(), self.name + B_IN +
clazz.name + B_TFUN + clazz.name, generatorID)->asSequence();
        case (cardinality(self,"1","n","0","n"))
           invariants := getInvariant(getInvariantName(), self.name + B_IN +
clazz.name + B_SURREL + clazz.name, generatorID)->asSequence();
        case (cardinality(self,"1","n","1","n"))
           invariants := getInvariant(getInvariantName(), self.name + B_IN +
clazz.name + B_TSURREL + clazz.name, generatorID)->asSequence();
        case (cardinality(self,"1","n","0","1"))
           invariants := getInvariant(getInvariantName(), self.name + B_IN +
clazz.name + B_PSUR + clazz.name, generatorID)->asSequence();
        case (cardinality(self,"1","n","1","1"))
           invariants := getInvariant(getInvariantName(), self.name + B_IN +
clazz.name + B_TSUR + clazz.name, generatorID)->asSequence();
        case (cardinality(self,"0","1","0","n"))
           invariants := getInvariant(getInvariantName(), self.name + B_IN +
clazz.name + B_REL+ clazz.name, generatorID)->asSequence()->
                         union(getInvariant(getInvariantName(), self.name +
B_CONV  + B_IN + clazz.name + B_PFUN + clazz.name, generatorID)->asSequence());
        case (cardinality(self,"0","1","1","n"))
           invariants := getInvariant(getInvariantName(), self.name + B_IN +
clazz.name + B_TREL + clazz.name, generatorID)->asSequence()->
           union(getInvariant(getInvariantName(), self.name+ B_CONV + B_IN  +
clazz.name + B_PFUN + clazz.name, generatorID)->asSequence());
        case (cardinality(self,"0","1","0","1"))
```