ESCOLA POLITÉCNICA
DE PERNAMBUCO

d.s.c. UPE
Departamento de Sistemas e Computação

ANALISADOR DE DIAGRAMAS DE CLASSE UML EM ECLIPSE

Trabalho de Conclusão de Curso

Engenharia da Computação

Thiago Wilson Nevares Trindade
Orientador: Prof. Eduardo Gonçalves Calábria

UNIVERSIDADE
DE PERNAMBUCO

THIAGO WILSON NEVARES TRINDADE

ANALISADOR DE DIAGRAMAS DE CLASSE UML EM ECLIPSE

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Recife, maio de 2009.

Aos meus pais.

Agradecimentos

A Deus, pela força e determinação para enfrentar essa jornada.

A minha família, pelo amor incondicional, pelo apoio essencial e pelo incentivo na busca de meus objetivos.

Ao professor Eduardo Calábria, pela orientação ao longo deste trabalho, pela sua confiança e amizade.

Ao professor Tiago Massoni, pela sua dedicação, amizade e pela sua presença mesmo distante.

A todos os professores do básico e do curso de Engenharia da Computação da Universidade de Pernambuco.

Aos amigos, pela companhia nessa odisséia, pelos risos que compartilhamos e pelos estudos e trabalhos que enfrentamos juntos.

Resumo

A modelagem é uma técnica, provada e bem aceita, da engenharia. Ela é um meio de capturar idéias, e no cerne do desenvolvimento de softwares encontramos as abstrações do mundo real. Uma abstração é uma idéia reduzida a sua forma essencial que pode ser mapeada ou expressa em modelos. A UML foi construída tendo softwares como objeto de trabalho. Ela se tornou a linguagem padrão para modelagem de aplicações de software e está crescendo em popularidade na área de modelagem em outros domínios. A *Alloy, por sua vez,* é uma linguagem de modelagem e análise. Ela fornece mecanismos para substituir a análise convencional de modelos por uma análise automática. Este trabalho visa juntar estes conceitos em uma ferramenta de construção e análise de diagramas UML com *Alloy* usando a Plataforma Eclipse, líder em numero de usuários e de ferramentas publicadas.

Abstract

Modeling is a proved and well accepted technique of engineering. It`s a way of capture ideas and at the core of software development are the abstractions of the real world. An abstraction is an idea reduced to its essential form that can be mapped or expressed in models. The UML was built having software as objects of work. It has become the standard modeling language of software applications and is growing in popularity in the modeling field and in other areas as well. Alloy, in other hand, is a language for modeling and analysis. It provides ways to replace conventional analyses with a fully automatic analysis. This work aims in join these concepts in a tool for building and analyzing UML diagrams with Alloy using the Eclipse Platform, leader in number of users and of tools published.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

UML – Unified Modeling Language (linguagem de modelagem unificada)

AA – Alloy Analyzer (analisador Alloy)

OCL – Object Constraint Language (linguagem de restrição sobre objetos)

EMF – Eclipse Modeling Framework (*framework* de modelagem do Eclipse)

API – Application Programming Interface (interface de desenvolvimento de aplicações)

MIT – Massachusetts Institute of Technology (Instituto de Tecnologia de Massachusetts)

EPL – Eclipse Public License (licença publica do Eclipse)

OSI – Open Source Initiative (Iniciativa de Código Aberto)

SDK – Software Development Kit (kit de desenvolvimento de software)

IDE – Integrated Development Environment (ambiente integrado de desenvolvimento

JDT – Java Development Tools (ferramentas de desenvolvimento Java)

PDE – Plug-in Development Environment (ambiente de desenvolvimento de *plug-ins*)

GEF – Graphical Editing Framework (*framework* de edição gráfica)

XMI – XML Metadata Interchange (XML de troca de metadados)

XML – Extensible Markup Language (linguagem de etiquetas extensíveis)

UI – User Interface (interface com o usuário)

SWT – Standard Widget Toolkit (kit de componentes padrão)

AWT – Abstract Window Toolkit (kit de janelas abstrato)

Capítulo 1

Introdução

Um modelo é uma simplificação da realidade [1]. Modelagem é uma técnica, provada e bem aceita, da engenharia. Modelagem é um meio de capturar idéias, relacionamentos, decisões e requisitos em uma notação bem definida que pode ser aplicada a diferentes contextos. No cerne do desenvolvimento de softwares encontramos as abstrações do mundo real. Uma abstração não é um modulo ou uma classe; ela é uma estrutura, uma idéia reduzida a sua forma essencial [2]. Uma abstração pode ser mapeada ou expressa em modelos.

A Unified Modeling Language (UML) é uma linguagem gráfica para visualização, especificação, construção e documentação de artefatos. A UML foi construída tendo softwares como objeto de trabalho. Ela se tornou a linguagem padrão *de facto* para modelagem de aplicações de software e está crescendo em popularidade na área de modelagem em outros domínios [3].

A Object Constraint Language (OCL) é uma adição à especificação UML que fornece formas de expressar restrições e lógica em modelos UML. Ela foi introduzida na versão 1.4 da linguagem UML, porém, na versão 2.0, ela foi formalizada usando a Meta-Object Facility e UML 2.0. Ela é, portanto, uma linguagem formal para descrever expressões livres de efeitos colaterais em modelos feitos usando a linguagem UML [5].

De forma similar, a *Alloy* é uma linguagem de modelagem e análise. Ela usa a idéia de uma notação precisa e expressiva sobre um pequeno conjunto de conceitos simples e robustos [2]. Além de uma linguagem que busca prevenir ambigüidade, Alloy fornece mecanismos para substituir a análise convencional de modelos por uma análise automática. Esta análise automática é conseguida através do *Alloy Analyzer* (AA) [4].

Na modelagem, assim com em qualquer outra parte da construção de sistemas de software, o processo pode se tornar complicado e laborioso sem o uso de uma ferramenta. Para todas as atividades, ferramentas são construídas para facilitar e agilizar o trabalho. Em 1999, quando a IBM revelou a Plataforma Eclipse

[6] a comunidade foi arrebatada pela magnitude do problema que a IBM estava tentando resolver.

A Plataforma Eclipse é uma plataforma computacional de propósito geral [7]. Foi com essa premissa que a IBM criou a Plataforma para unificar todos os seus ambientes de desenvolvimento e ferramentas [8]. Desde então, com a ajuda de uma comunidade que cresce a cada ano, a Plataforma Eclipse se tornou uma das Plataformas mais usadas pela comunidade de desenvolvedores Java.

Um dos muitos projetos ativos da Plataforma Eclipse é o Projeto de Modelagem. O Eclipse Modeling Framework (EMF) é um framework poderoso e gerador de código para construção de aplicações a partir de definições de modelos [9]. Este trabalho visa juntar todos estes conceitos e mecanismos em uma ferramenta que promoverá o uso de modelos e facilitará a sua construção e análise.

1.1 Objetivos

Para construir uma ferramenta capaz de consolidar os conceitos e mecanismo descritos acima, é necessário que alguns objetivos sejam atingidos. Dentre eles pode-se citar:

- Definir um mapeamento entre as construções da linguagem de modelagem visual – UML – e as construções da linguagem de modelagem e análise - *Alloy*.
- Construir, usando a Plataforma Eclipse, um mecanismo de transformação entre as linguagens usando as regras mapeadas.
- Construção do analisador de código *Alloy* usando a *Application Programming Interface* (API) do Analisador *Alloy – Alloy Analyzer*.
- Integrar, usando a Plataforma Eclipse, os módulos de transformação e análise em uma ferramenta de analise de modelos de classe UML.

1.2 Estrutura

Os capítulos da presente monografia estão organizados como segue:

O **Capítulo 2** descreve o que é UML e um de seus principais modelos, o modelo de classes. É também nele que são descritos os elementos básicos de um

modelo de classe e a linguagem OCL, usada para definir restrições em modelos UML.

O **Capítulo 3** ilustra os conceitos básicos de *Alloy* como ferramenta de modelagem e análise, alguns conceitos básicos que regem a lógica pro trás da análise e construções usadas pela linguagem para definir modelos e restrições sobre eles.

O **Capítulo 4** discorre sobre a Plataforma Eclipse. Inicialmente falando sobre o Projeto Eclipse, seus componentes e subprojetos e depois examinando mais detalhadamente a arquitetura e facilidades que a Plataforma Eclipse fornece para a criação de extensões – *plug-ins*.

O **Capítulo 5** define a estrutura do Analisador de Diagramas, produto deste trabalho. Nele é descrita a arquitetura básica do Analisador e como esta se relaciona com o *Eclipse Modeling Framework* – o framework de modelagem do Eclipse.

O **Capítulo 6** fala da transformação de modelos em UML em estruturas em *Alloy*. É mostrado o mapeamento entre as construções de uma linguagem com a outra e a API usada para executar as analises usando *Alloy*.

O **Capítulo 7** traz as conclusões e contribuições deste trabalho, alem de trabalhos futuros que podem ser derivados desta monografia.

Capítulo 2

Unified Modeling Language

Na superfície, a UML é uma linguagem visual para capturar designs e padrões de software. Ela, porém, foi idealizada para ser um meio comum de captura e de expressão de relacionamentos, comportamentos e idéias de alto nível em uma notação fácil de aprender e eficiente de escrever [3]. A UML pode ser aplicada em inúmeras áreas e pode capturar e comunicar desde a organização de uma empresa, os processos de um negócio até softwares empresariais complexos.

A UML é visual; tudo nela pode ser tem uma representação gráfica. Este trabalho procura usar o poder de representação da UML e sua disseminação na área de modelagem e especificação de software, fazendo da UML a porta de entrada para a utilização da ferramenta proposta.

2.1 Diagramas da UML

A UML 2.0 divide os diagramas em duas categorias: diagramas estruturais e diagramas comportamentais. Diagramas estruturais são usados para capturar a organização física das partes que compõem o sistema. Existem diversos diagramas estruturais em UML entre eles o diagrama de classes é dos mais utilizado em especificações de software e é o foco deste trabalho.

2.2 Diagramas de Classe

O diagrama de classes é um dos mais fundamentais tipos de diagramas em UML [1]. Eles são usados para capturar os relacionamentos estáticos em um projeto de software. Os diagramas de classes usam classes e interfaces para detalhar as entidades que fazem parte do sistema e as relações entre elas.

2.3 Classes

Uma classe representa um grupo de coisas com o mesmo estado e o mesmo comportamento. Em UML, uma classe é um tipo de classificador. Cada instância de

uma classe é chamada de objeto. Uma classe pode representar um conceito concreto e tangível, como um carro, ela pode representar um conceito abstrato, como um documento ou ela pode ainda representar um conceito intangível, como uma estratégia de investimento.

Uma classe é representada com uma caixa retangular dividida em compartimentos. Um compartimento é uma área no retângulo onde se escreve suas informações. O primeiro compartimento guarda o nome da classe, o segundo compartimento guarda os atributos da classe e o terceiro compartimento guarda as operações da classe. Outros compartimentos podem ser adicionados para agregar mais informações à classe, mas isto foge da notação usual.

A UML sugere que o nome de uma classe comece com letra maiúscula, esteja centralizado no compartimento superior, seja escrito em negrito e em itálico se a classe for abstrata.

Uma classe abstrata é uma classe que descreve assinaturas de operações, mas nenhuma implementação. Uma classe abstrata é útil para identificar funcionalidades comuns entre vários tipos de objetos. Outro classificador que pode ser usado no lugar de uma classe abstrata é uma interface. Uma interface descreve propriedades e operações, mas, como as classes abstratas, não definem implementação.

2.3.1 Atributos

Os atributos representam os detalhes de uma classe. Eles podem ser primitivas simples ou relacionamentos com outra classe. Para cada caso existe uma notação indicada. Não há diferença de semântica entre cada tipo de atributo. A diferença é a quantidade de detalhes que se pode expressar. Nesta seção será descrita a forma mais simples, atributos contidos ou *inlined attributes*.

Atributos contidos são aqueles listados no segundo compartimento da representação de uma classe. A UML referencia os atributos contidos como notação de atributos – *attribute notation*. A notação de atributo é a seguinte.

visibilidade / nome : tipo multiplicidade = valor inicial

A visibilidade pode ser publica, quando uma característica pode ser acessada por qualquer classe, protegida, quando ela pode ser acessada na classe que a define e nas suas subclasses, privada, quando ela é acessada apenas pela classe

que a define e de pacote, quando ela pode ser acessada pelos membros do mesmo pacote da classe que a define.

2.3.2 Operações

Operações são características de uma classe que especificam como invocar um comportamento em particular. A UML faz uma distinção clara entre uma operação e a implementação do comportamento que ela invoca – método.

Operações são listadas no terceiro compartimento de uma classe. A UML define, junto com a notação de operações, a notação de parâmetros.

visibilidade nome (parâmetros) : tipo

parâmetro ::= direção nome : tipo multiplicidade = valor padrão

2.3.3 Relacionamentos

Atributos podem ser representados também usando a notação de relacionamento. Esta notação resulta em diagramas de classe maiores, mas, também, mais detalhados. A notação de relacionamento também determina como exatamente o atributo se relaciona com a classe.

Dependência

É o tipo mais fraco de relacionamento. Ela significa que uma classe usa ou conhece a outra. A dependência é normalmente lida como: “A usa um B”.

Associação

É mais forte que a dependência e indica, usualmente, que o relacionamento dura um período grande de tempo. O tempo de vida dos objetos associados não é, necessariamente, interligado. A associação é usualmente lida como: “A tem um B”

Agregação

É uma versão mais forte da associação. Diferente da associação, ela geralmente implica propriedade e pode implicar na ligação entre o tempo de vida dos objetos. A agregação é tipicamente lida como: “A possui um B”.

Composição

É um tipo de relacionamento muito forte entre classes. A composição é usada para capturar os relacionamentos do tipo parte/todo. O tempo de vida da parte é geralmente ligado ao tempo de vida do todo. A composição é comumente lida como “B é parte de A”.

Generalização

É um relacionamento que define que o destino é uma versão mais geral, ou menos especifica, da classe fonte. Ela é usada principalmente para isolar partes em comum de classificadores diferentes. A generalização é geralmente lida como: “A é um B”.

2.4 OCL

A *Object Constraint Language* (OCL) é uma adição à especificação UML 2.0 que fornece mecanismos de expressar restrições e lógica nos modelos. Ela é uma linguagem especial que foi desenvolvida especificamente para consultas, ou seja, ela não pode modificar o modelo.

OCL pode ser usado em qualquer lugar em UML e é tipicamente associado a uma classe através de uma nota. Ela pode ser usada para expressar precondições, pós-condições, invariantes, guardar condições e resultados de chamadas de métodos. Neste trabalho o foco serão as restrições sobre classes.

2.4.1 Sintaxe

Toda expressão em OCL deve ter um senso de contexto com o qual a expressão se relaciona. Muitas vezes o contexto pode ser deduzido pela localização da expressão. Em UML pode-se usar uma nota para ligar expressões em OCL aos elementos do modelo. O contexto pode também ser explicito, como mostra o exemplo abaixo.

context nome_da_classe

Se o contexto for explicito, o tipo de expressão deve ser também explicito. Quando este contexto é uma classe a palavra-chave ***self*** pode ser usada para referenciar uma instancia da classe. Por exemplo, se um atributo de uma classe deve ser mantido sempre acima de um limiar inferior predefinido, podemos escrever a seguinte expressão OCL.

inv: self.atributo > **limiar_inferior**

É importante notar que esta expressão OCL é uma invariante, o que significa que ela deve ser sempre avaliada para verdadeiro. O sistema estaria em um estado invalido se o atributo estivesse com o valor abaixo do limiar. Uma invariante é representada pela palavra-chave ***inv***.

Para navegar pelas características de uma classe o operador **“.”** é usado. Este operador pode ser usado para acessar atributos e navegar por relacionamentos. A invariante a seguir mostra como se pode usar o ponto para navegar por um relacionamento.

inv: self.destino_do_relacionamento.atributo_do_destino > limiar

Para classes como contexto existem algumas operações especiais como *allInstances*. A chamada desta operação retorna uma coleção de instancias da classe no modelo.

inv: nome_da_classe.allInstances()->size()

Quando a multiplicidade de uma associação diferente de um o destino pode ser tratado como uma coleção. O operador passa a ser o de coleção: **“->”**. A operação *size* é usada para retornar o tamanho da coleção. Em OCL existem varias operações predefinidas que podem ser usadas em coleções. Um exemplo de operação que pode ser usada em uma coleção é a operação *notEmpty*. Ela retorna verdadeiro se a coleção não é vazia. A expressão OCL que segue mostra um exemplo de uso da operação *notEmpty*.

inv: self.coleção_de_objetos->notEmpty()

Outra operação de coleção muito usada é a *isEmpty*. Ela é a operação complementar da *notEmpty*, ou seja, ela retorna verdadeiro se a coleção é vazia. Se uma associação não tem um nome especificado, pode-se usar o nome da classe de destino. A expressão abaixo exemplifica estas duas funcionalidades.

inv: self.nome_da_classe_da_coleção->isEmpty()

Existem também as operações *select* e *reject*. Elas atuam sobre coleções para restringir valores. O resultado desta operação é uma nova coleção com os objetos apropriados, mas, a coleção original não é alterada.

inv: self.coleção_de_objetos->select(expressão_lógica)

Dentre as outras operações existentes pode-se ainda destacar as operações *exists* e *includes*. A operação *exists* é útil para determinar se existe pelo menos um item em uma coleção que satisfaça uma determinada condição. A operação *includes* é usada para determinar se uma coleção contém um determinado objeto.

inv: self.coleção_de_objetos->exists(expressão_lógica)

inv: self.coleção_de_objetos->includes(objeto)

Um operador que merece destaque é o operador *implies*. O operador *implies* avalia uma expressão lógica e, se ela for verdadeira, retorna o valor da segunda expressão.

inv: self.atributo <> 0 implies self.coleção_de_objetos->notEmpty()

Capítulo 3

Alloy

A *Alloy* é uma linguagem de modelagem estrutural baseada em lógica de primeira ordem e cálculo relacional [10]. Ela é usada para expressar restrições estruturais e comportamentos complexos. Ela usa a idéia de uma notação precisa e expressiva, baseada em um cerne pequeno com conceitos simples e robustos, de especificações formais. Diferente de outras linguagens de especificações formais, ela substitui a analise convencional, baseada em prova de teoremas, por uma analise totalmente automatizada que gera retornos imediatos.

Esta análise, no entanto, não é completa. A análise é feita em um espaço finito de casos. Mas, graças a avanços nas tecnologias de resolução de restrições, o espaço de casos examinado é, normalmente, enorme – bilhões de casos ou mais – e, portanto, oferece um grau de cobertura inatingível em testes comuns.

O *Alloy Analyzer* – AA ou Analisador *Alloy* – é a ferramenta usada para realizar a análise de modelos em *Alloy*. O *Alloy Analyzer* é um solucionador de restrições que provê simulações e checagens de modelos. Assim como a própria linguagem, o *Alloy Analyzer* vem sendo desenvolvido pelo Grupo de Design de Software do MIT – Massachusetts Institute of Technology. A sua primeira versão foi publicada em 1997.

Usando *Alloy* e o Analisador *Alloy*, o usuário fornece um modelo e uma propriedade a ser checada, que pode ser geralmente, expressa sucintamente como um simples caso de teste. Mas diferente de um caso de teste, a checagem em *Alloy* não necessita nenhuma linha de código na linguagem de programação convencional. Possibilitando assim, uma análise incremental do modelo com a profundidade e clareza de uma especificação formal.

3.1 A Lógica

No centro de toda linguagem de modelagem existe uma lógica que fornece os conceitos fundamentais. Ela deve ser simples, pequena e expressiva. A lógica de *Alloy* suporta três diferentes estilos: calculo de predicados, expressão navegacional e calculo relacional.

No estilo de cálculo de predicados, existem apenas dois tipos de expressões: nomes de relações, que são usadas como predicados, e *tuplas* formadas por variáveis quantificadas. Neste estilo, uma restrição que diz que um objeto possui um mapeamento único entre outros dois objetos, através de uma relação, poderia ser escrita da seguinte forma.

all o: Objeto_Origem, d1, d2: Objeto_Destino | n->d1 in Nome_Relação_Pai and n->d2 in Nome_Relação_Pai implies d1 = d2

No estilo de expressão navegacional, expressões denotam conjuntos, formados pela "navegação" de variáveis quantificadas através de relações. Neste estilo, a mesma restrição é escrita como segue.

all o: Objeto_Origem | lone o.Nome_Relação_Pai

No estilo de calculo relacional, expressões denotam relações e não há quantificadores. Usando operadores suportados pela linguagem, a restrição é escrita abaixo no estilo de calculo relacional.

no ~Nome_Relação_Pai.Nome_Relação_Pai -iden

O estilo de calculo de predicados, tipicamente, requer muita digitação, enquanto o estilo de calculo relacional tende a ser críptico. O estilo mais usado, portanto, o navegacional, com usos ocasionais dos outros estilos onde apropriados.

3.1.1 Átomos e Relações

Nos modelos em *Alloy*, todas as estruturas são construídas de átomos e relações, que correspondem às entidades básicas e as relações entre elas. Os átomos são entidades primitivas que não podem ser divididas em partes menores, que suas propriedades não mudam com o tempo e que não possuem nenhuma propriedade pré-definida.

Qualquer coisa que se queira modelar que não siga estas premissas, deve usar relações. Uma relação é uma estrutura que relaciona átomos. Ela consiste em um conjunto de *tuplas*, cada *tupla* sendo uma seqüência de átomos. Se visualizado como uma tabela, cada célula contém um átomo. A ordem das colunas é importante, mas a ordem das linhas não. Todas as células devem ser preenchidas.

Uma relação pode ter qualquer numero de linhas, chamado de tamanho. O numero de colunas em uma relação é chamado de ordem e deve ser maior que um. Relações com ordem um, dois e três são chamadas: unárias, binárias e ternárias. Uma relação com ordem maior que três é chamada de multi-relação. Uma relação

unária, correspondente a uma tabela com apenas uma coluna, representa um conjunto de átomos. Uma relação unária com apenas uma *tupla*, que corresponde a uma tabela com apenas uma célula, representa um escalar.

3.1.2 Operadores

Assim como outras linguagens, *Alloy* possui constantes e operadores. Os operadores podem ser classificados como operadores de conjunto e operadores relacionais. Para os operadores de conjunto, a estrutura das *tuplas* é irrelevante. Já para operadores relacionais, a estrutura das *tuplas* é essencial.

Constantes

As constantes de *Alloy* são mostradas na Tabela 1. Vale ressaltar que ***none*** e ***univ*** são conjuntos unários e que ***iden*** é binário.

Tabela 1. Constantes de *Alloy*.

none	Conjunto vazio. Nenhum átomo.
univ	Conjunto universo. Todos os átomos.
iden	Conjunto identidade. *Tuplas* que relacionam todo átomo a ele mesmo.

Operadores de Conjunto

Os operadores de conjunto são listados na Tabela 2. Estes operadores podem ser aplicados a qualquer par de relações contanto que elas tenham a mesma ordem.

Tabela 2. Operadores de Conjunto de *Alloy*.

+	União.
&	Interseção.
-	Diferença.
in	Subconjunto.
=	Igualdade.

Operadores Relacionais

A Tabela 3 contém os operadores relacionais de *Alloy*. Dada a importância destes operadores, uma breve descrição de alguns deles é dada em seguida.

O produto é o produto entre duas relações é obtido fazendo a combinação de cada *tupla* da primeira relação com todas as *tuplas* da segunda e concatenando-as.

A ordem do novo conjunto é a soma das ordens dos conjuntos operados. Se os dois operandos forem escalares o resultado é um par.

Tabela 3. Operadores Relacionais de *Alloy*.

->	Arrow – Produto.
.	Dot – Junção ou Composição.
[ ]	Box – Junção ou Composição.
~	Transposição.
^	Fechamento Transitivo.
*****	Fechamento Transitivo Reflexivo.
<:	Restrição de Domínio.
:>	Restrição de Escopo.
++	Sobreposição.

A junção é um dos operadores mais importantes de *Alloy*. Ele pode ser usado para combinar *tuplas* e relações. Na junção de *tuplas*, se o ultimo elemento do primeiro operando for diferente do primeiro elemento do segundo, o resultado é vazio. Mas, se eles forem iguais, o resultado é a junção dos elementos da primeira *tupla*, no inicio, com os elementos da segunda, no fim, menos o elemento em comum.

O resultado da aplicação do operador de junção em duas relações é obtido através da aplicação do operador de junção em todas as combinações de *tuplas* dos operandos. A ordem das relações pode ser qualquer, contanto que não sejam ambas unárias.

3.1.3 Restrições

Restrições podem ser formadas pro duas expressões usando os operadores de comparação **in** e **=**. Restrições maiores e mais complexas podem ser construídas a partir de restrições menores e combinando-as com os operadores lógicos e quantificando restrições que possuem variáveis livres.

Operadores Lógicos

Existem duas formas para cada operador lógico. Na Tabela 4 são exibidos os operadores lógicos nas suas duas formas. A negação pode ser usada com os operadores de comparação. O operador *else* é usado junto com o operador de

implicação. O operador *implies* é, normalmente, usado de forma aninhada, ou seja, formando construções de implicações e alternativas.

Tabela 4. Operadores Lógicos de *Alloy*.

not	**!**	Negação.
and	**&&**	Conjunção.
or	**||**	Disjunção.
implies	**=>**	Implicação.
else	**,**	Alternativa.
iff	**<=>**	Implicação mutua.

Quantificadores

As restrições quantificadas são formadas por um quantificador, uma variável e um conjunto com os valores que a variável pode assumir e a restrição sobre a variável. **Qualificador variável : conjunto | restrição**. Os qualificadores estão listados na Tabela 5.

Tabela 5. Quantificadores de *Alloy*.

all x : e | F	A restrição F se aplica a todo x em e.
some x : e | F	A restrição F se aplica a algum x em e.
no x : e | F	A restrição F se aplica a nenhum x em e.
lone x : e | F	A restrição F se aplica a no máximo um x em e.
one x : e | F	A restrição F se aplica a exatamente um x em e.

3.1.4 Outros Elementos de *Alloy*

Existem ainda outros elementos de *Alloy* que podem ser usados para melhorar as descrições dos modelos e torná-los mais robustos e completos. **Funções** e **predicados** que facilitam a escrita de expressões e restrições, **fatos** que descrevem restrições do modelo, **afirmações** que são usadas para propriedades que se espera atingir, **comandos** como *run* e *check* para procurar soluções e contra-exemplos nos modelos e outros operadores e facilidades que podem ser usados em *Alloy*.

Capítulo 4

Eclipse

O Eclipse é um projeto open source. Seu propósito é prover uma plataforma integrada de ferramentas. O projeto inclui um projeto principal, que prove um *framework* genérico para integração de ferramentas, e um ambiente de desenvolvimento Java. Outros projetos estendem o projeto principal para suportar tipos específicos de ferramentas e ambientes de desenvolvimento. Os projetos no Eclipse são desenvolvidos usando Java e executam em vários sistemas operacionais.

Através do envolvimento de desenvolvedores entusiastas e comprometidos em um ambiente organizado para facilitar a livre troca de idéias e tecnologias, Eclipse aspira criar a melhor plataforma de integração possível. Todo software produzido pelo projeto é disponibilizado pelos termos de uma licença própria do projeto. A *Eclipse Public License* [15] (EPL), que declara os termos legais de uso dos produtos gerados pelo projeto, permite que qualquer pessoa use, modifique e redistribua o software sem qualquer custo. A licença permite também a distribuição do projeto ou parte dele junto com software proprietário como parte de uma ferramenta comercial. A licença EPL é aprovada pelo grupo *Open Source Initiative* (OSI) e reconhecido pelo *Free Software Foundation* como uma licença de software livre. Ainda pelos termos da EPL, qualquer software dado como contribuição ao projeto deve também ser licenciado pelos termos da EPL.

O desenvolvimento no projeto Eclipse é supervisionado pelo *Eclipse Foundation*, uma organização independente e sem fins lucrativos. A fundação inclui cerca de 100 empresas que apóiam Eclipse e oferecem ferramentas comerciais baseadas no Eclipse alem de indivíduos que participam sem uma representação comercial. O *Eclipse Foundation* opera de acordo com regras e processos de desenvolvimento que definem papéis e responsabilidades dos participantes incluindo o quadro de direção, o comitê de organização e gerenciamento, os comitês de gerenciamento de projetos, os membros, os usuários e os desenvolvedores do Eclipse.

4.1 O Projeto

O trabalho em desenvolvimento no projeto Eclipse é dividido em projetos de alto nível como o Projeto Eclipse, o Projeto de Modelagem, o Projeto de Ferramentas e o Projeto de Tecnologia. O Projeto Eclipse engloba os componentes centrais necessários ao desenvolvimento utilizando a plataforma. Seus componentes são essencialmente fixos e distribuídos como um único pacote. Este pacote é referenciado como o *Eclipse Software Development Kit (SDK)*. Os componentes dos outros projetos são usados para fins específicos e são geralmente distribuídos independentemente. Novos projetos são criados e novos componentes adicionados a projetos em andamento constantemente.

4.1.1 O Projeto Eclipse

O Projeto Eclipse suporta o desenvolvimento de uma plataforma, ou *framework,* para a criação de ambientes integrados de desenvolvimento – *integrated development environments (IDEs)* – e outras aplicações. O *framework* do eclipse é desenvolvido usando Java, mas é usado para criar ferramentas de desenvolvimento para outras linguagens, e.g., C++, Lua, XML, etc.).

O Projeto em si é dividido em quatro subprojetos principais: Equinox, a Plataforma, as ferramentas de desenvolvimento Java – *Java Development Tools (JDT)*, e o ambiente de desenvolvimento de *plug-ins* – *Plug-in Development Environment (PDE).* Juntos, os quatro projetos dispõem os meios necessários para estender o *framework* e desenvolver ferramentas baseadas em Eclipse.

Equinox e a Plataforma são os componentes principais do Projeto Eclipse e, juntos, são considerados, pela maioria dos participantes do projeto, como sendo o Eclipse. Equinox é uma instancia da especificação central de *framework* OSGi R4 [7], que especifica o modelo de componentes sobre o qual todo o Projeto Eclipse se baseia. A Plataforma define outros *frameworks* e serviços necessários para dar suporte a criação e integração de ferramentas. Estes serviços incluem, entre outros, um ambiente de trabalho e interface com o usuário padrões – *workbench* – e mecanismos para gerenciamento de projetos, arquivos e pastas.

O JDT é um ambiente de desenvolvimento Java completo construído usando Eclipse [6]. Suas ferramentas são altamente integradas e representam o poder da Plataforma Eclipse. Ele pode ser usado para desenvolver projetos para Eclipse ou

outras plataformas. O JDT é o ambiente usado para desenvolver o próprio Projeto Eclipse.

O PDE disponibiliza editores e visões para facilitar a criação de *plug-ins* para Eclipse [6]. O PDE é construído sobre o JDT e o estende provendo suporte para as partes não relacionadas com Java na atividade de criação de *plug-ins*, como registrar extensões *plug-in*, entre outras.

4.1.2 O Projeto de Modelagem

O Projeto de Modelagem do Eclipse é o ponto focal para a evolução e promoção de tecnologias de desenvolvimento baseada em modelos em Eclipse. No seu centro esta o *Framework* de Modelagem do Eclipse – *Eclipse Modeling Framework (EMF)* – que prove um *framework* básico para modelagem. Outros subprojetos de modelagem são construídos sobre o EMF, provendo integração com banco de dados, transformações de modelos e geração de editores gráficos. Faz parte também do projeto de modelagem instâncias de vários padrões de modelagem. Por exemplo, o projeto UML2 [16] é uma instancia do meta-modelo UML 2.x baseado em EMF.

4.1.3 O Projeto de Ferramentas

O Projeto de Ferramentas desenvolve uma serie de ferramentas de desenvolvimento extensíveis baseadas na Plataforma Eclipse. Ele inclui uma grande diversidade de subprojetos. Alguns disponibilizam ferramentas para lidar com linguagens de programação, incluindo C/C++, COBOL e PHP. Outros como o *Framework* de Editores Gráficos – *Graphical Editing Framework [17] (GEF)*, que fornece a base para varias categorias de ferramentas. EMF foi inicialmente criado como um subprojeto do Projeto de Ferramentas, antes do Projeto de Modelagem ser criado.

4.1.4 O Projeto de Tecnologia

O Projeto de Tecnologia da à oportunidade para pesquisadores, acadêmicos e educadores de se envolver na evolução constante da Plataforma. Este projeto serve como ponto de entrada e repositório temporário de trabalhos novos ou experimentais que podem atingir um fim natural ou evoluir e ser alocado em outro

projeto, novo ou existente. Outros projetos de alto nível podem ter projetos incubados para este propósito.

4.1.5 Outros Projetos

Um crescente número de outros projetos suporta e fornece ferramentas mais especializadas. Estes incluem o Projeto da Plataforma de Ferramentas de Dados, o Projeto da Plataforma de Desenvolvimento de Softwares Embarcados, o Projeto da Plataforma de Ferramentas para Desenvolvimento Web, etc.

4.2 A Plataforma Eclipse

A Plataforma Eclipse é um *framework* para construção de IDEs. Ela simplesmente define a estrutura básica de uma IDE. Ferramentas específicas estendem o *framework* e são introduzidas nele para definir uma IDE em particular.

De fato, a arquitetura da plataforma permite que um subconjunto de seus componentes seja usado na construção de qualquer aplicação. A arquitetura da Plataforma Eclipse pode ser visualizada na Figura 1 e é descrita a seguir.

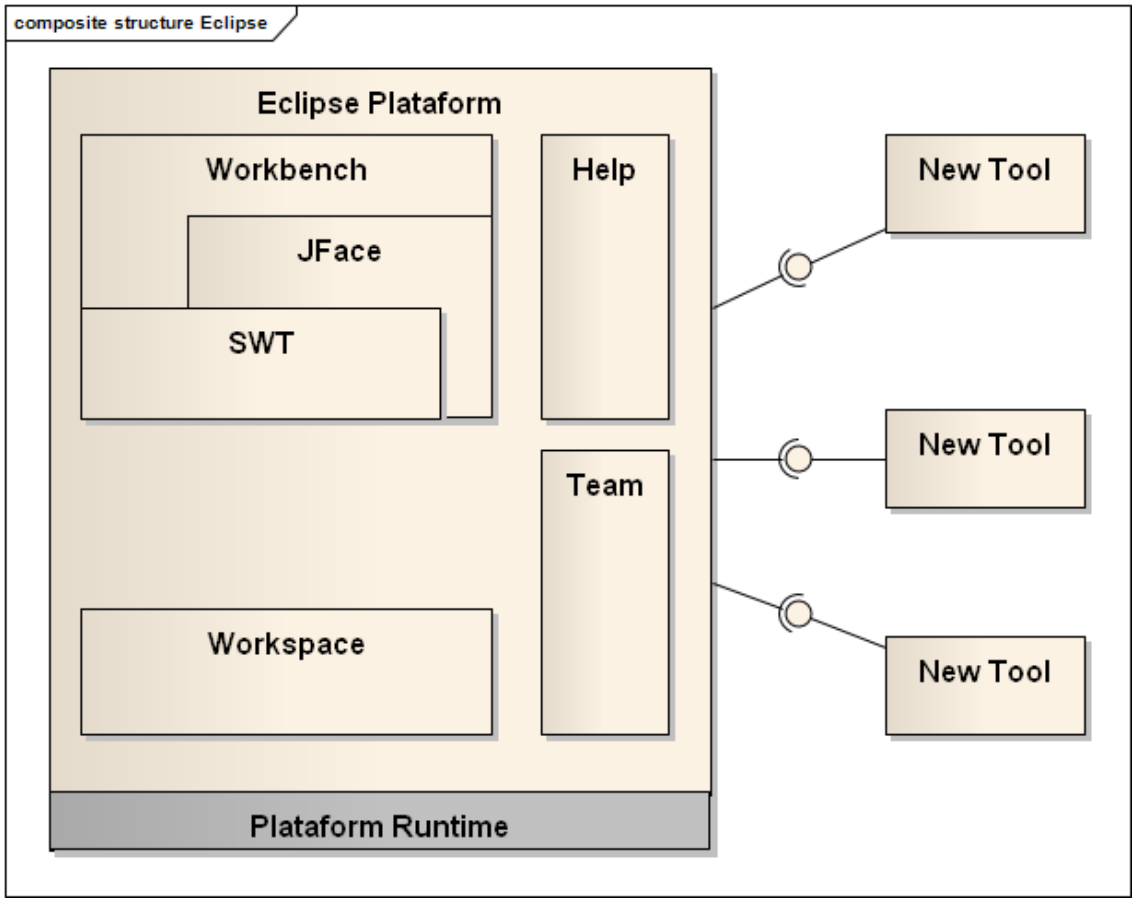

Figura 1. Arquitetura da Plataforma Eclipse (adaptado [7]).

4.2.1 A Arquitetura de *Plug-ins*

Na Plataforma Eclipse, uma unidade básica de funcionalidade, ou um componente, é chamado de *plug-in*. A Plataforma em si e as ferramentas que a estende são ambas compostas de *plug-ins*. Uma ferramenta simples pode ser composta de um *plug-in*, mas ferramentas mais complexas são normalmente divididas em vários *plug-ins*. Cada *plug-in* contribui com funcionalidades que podem ser invocadas pelo usuário ou reusadas e estendidas por outros *plug-ins*.

O motor de execução da plataforma é responsável por achar e executar os *plug-ins*. Ele é desenvolvido sobre a Plataforma de Serviços OSGi, que fornece um padrão flexível de componentes no seu *framework* permitindo que *plug-ins* sejam instalados e removidos sem a necessidade de reiniciar a plataforma.

De uma perspectiva de empacotamento, um *plug-in* inclui tudo necessário para a sua execução, como o código Java, imagens, textos, etc. Ele também inclui dois arquivos de manifesto, um usado pela plataforma antes da adoção do padrão OSGi e outro definido pelo próprio OSGi. No arquivo de manifesto do OSGi, chamado *META-INF/MANIFEST.MF*, identifica o *plug-in* e fornece, entre outras informações, as diretivas de dependência. Ele inclui o seguinte:

- **Componentes requeridos.** Sua dependência com outros *plug-ins*.
- **Pacotes exportados.** Os pacotes que ele faz visível para outros *plug-ins*.

O arquivo de manifesto do *plug-in*, chamado *plugin.xml*, declara as interconexões com outros *plug-ins*. Ele pode conter as seguintes diretivas:

- **Pontos de extensão.** Declaração de funcionalidades que ele disponibiliza para outros *plug-ins*.
- **Extensões.** Uso (desenvolvimento) de pontos de extensão de outros *plug-ins*.

A plataforma de execução gerencia o ciclo de vida dos *plug-ins* e faz a correspondência entre pontos de extensão e as extensões. Ela usa mecanismos de carregamento de classes para garantir a visibilidade declarada nos arquivos de manifesto e fornece um registro que pode ser consultado por *plug-ins* para encontrar as extensões para seus pontos de extensão.

4.2.2 Workspaces

As ferramentas integradas à Plataforma Eclipse trabalham com arquivos e pastas comuns, mas elas usam uma interface (*application programming interface – API*) baseada em recursos, projetos e a área de trabalho. Um recurso é a

representação de um arquivo ou pasta na Plataforma Eclipse que fornece as seguintes funcionalidades adicionais:

- *Listeners* de mudança podem ser registrados para que uma notificação seja enviada cada vez que o recurso mudar. Eles são chamados de *resource deltas*.
- *Markers*, como mensagens de erro ou listas de tarefas, podem ser adicionados aos recursos.
- O conteúdo anterior, ou o histórico, de um recurso pode ser monitorado.

Um projeto é um tipo especial de recurso do tipo pasta que faz o mapeamento entre pastas especificadas pelo usuário e o sistema de arquivos. As subpastas de um projeto têm a mesma representação do sistema de arquivos físico, mas projetos são pastas de alto nível dentro do contêiner virtual de projetos do usuário, chamado de *workspace*. Projetos podem também ser marcados com um comportamento em particular, chamado de natureza do projeto. Por exemplo, um projeto com a natureza Java indica que ele possui código fonte de um programa Java.

Capítulo 5
Os Plug-ins do Analisador de Diagramas

Para que os objetivos deste trabalho sejam obtidos é necessária a construção de uma ferramenta que integre a criação e edição de diagramas UML com a análise de modelos usando *Alloy*. Com este intuito, por ser uma eficaz e de vasto uso, foi adotada a Plataforma Eclipse como provedora de meios de criação, distribuição e disseminação do Analisador.

Por se tratar de uma ferramenta de modelagem e auxilio ao desenvolvimento através de modelos, alguns padrões definidos pelo *Eclipse Modeling Framework* – EMF – foram usados. Por ser também uma ferramenta que se integra com a própria plataforma, foi seguido o modelo de criação de *plug-ins* definido pela própria plataforma.

5.1 A estrutura

A Figura 2 revela a estrutura de *plug-ins* usada para a construção do Analisador. Nela podemos ver a Plataforma Eclipse que fornece a infra-estrutura para a construção e integração dos *plug-ins* e o EMF que fornece classes básicas para a construção de aplicações de modelagem explorando as facilidades fornecidas pela Plataforma Eclipse.

Ainda na Figura 2 podemos ver a representação dos três *plug-ins* criados para o Analisador. A divisão mostrada pela figura segue um padrão de EMF que prevê uma separação entre o modelo, os controladores de edição e o ambiente de edição, concretizada pelos *plug-ins* Modelo, Edição e Editor respectivamente.

5.2 O *Plug-in* do Modelo

Para falar do *plug-in* do modelo é necessário um entendimento básico de como funcionam os padrões e facilidades que o EMF e o modelo usado para representar modelos em EMF, chamado Ecore, fornecem.

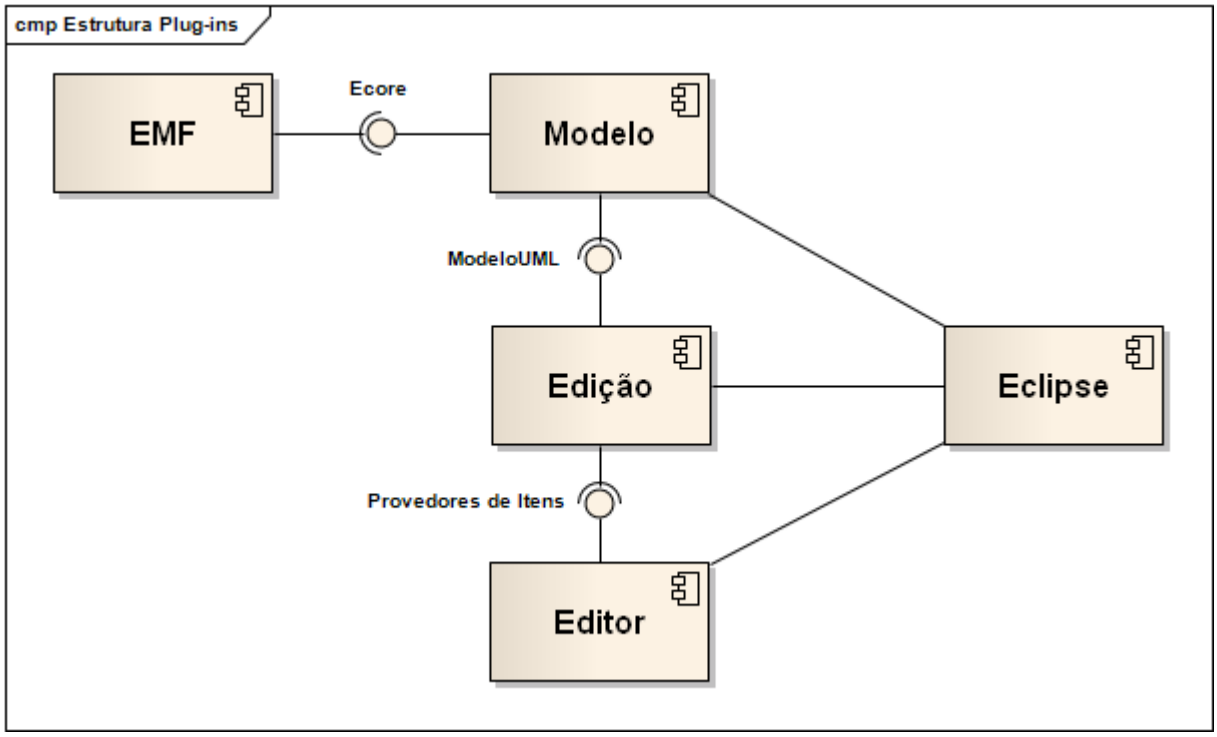

Figura 2. Estrutura de *plug-ins* do Analisador

5.2.1 O Modelo Ecore

O Ecore é um modelo para representar modelos em EMF e é também o seu próprio meta-modelo. Um conjunto simplificado de elementos do modelo Ecore é mostrado na Figura 3. Para as necessidades deste trabalho citaremos apenas os quatro elementos mais importantes.

- **EClass** é usado para representar uma classe modelada. Ela possui um nome, zero ou mais atributos e zero ou mais referencias.
- **EAtribute** é usado para representar um atributo modelado. Atributos têm um nome e um tipo.
- **EReference** é usado para representar um lado de uma associação entre classes. Ela tem um nome, uma *flag* booleana para indicar se ela representa uma agregação e um tipo de referencia (destino da referencia), que deve ser uma classe.
- **EDataType** é usada para representar o tipo de um atributo. Um tipo de dado pode ser um primitivo como ***int*** ou ***float*** ou um tipo de objeto.

Vale notar que os nomes das classes do Ecore correspondem a termos usados em UML. Na realidade, o Ecore foi projetado para ser um subconjunto

simplificado de elementos UML, apenas aqueles elementos necessários para a descrição de outros modelos.

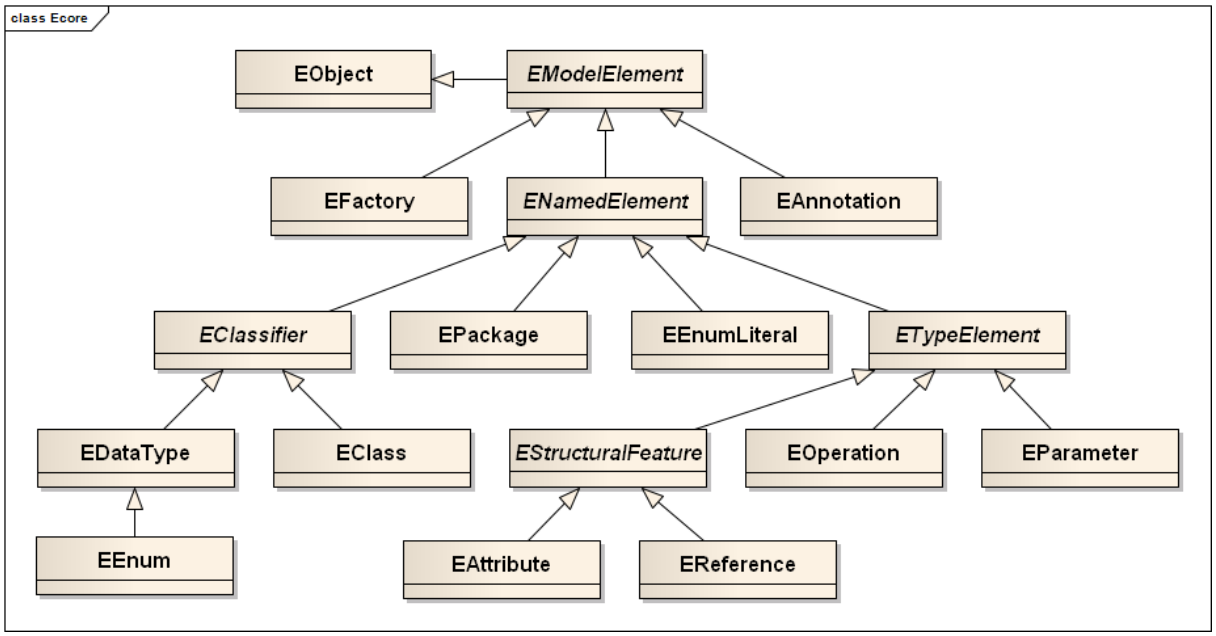

Figura 3. Modelo Ecore

5.2.2 O Modelo do *Plug-in*

Usando as classes Ecore como base para as classes do modelo UML usado no Analisador obtemos algumas vantagens importantes. Podem-se usar as bibliotecas de EMF para manipular as classes do modelo, as classes que herdam das classes Ecore herdam também uma implementação da interface ***Notifier***, que é muito importante para o *plug-in* do Editor, e a serialização em XMI [11] – XML [12] Metadata Interchange – fornecida pelo framework.

A Figura 4 revela o modelo de classes criado para modelar o subconjunto de UML suportado pelo Analisador, alem de algumas classes especiais para a integração com o framework de modelagem. Para deixar o modelo mais simples e claro, as dependências foram omitidas e apenas a hierarquia das classes é mostrada.

No modelo só são mostradas as interfaces, isto por causa do padrão adotado pelo EMF de que todos os itens do modelo são definidos como interfaces que herdam das interfaces do *framework* e as usas respectivas implementações herdam das classes de implementação do *framework* e implementam as interfaces.

Duas relações importantes de ressaltar no modelo. A primeira é a ligação entre a *interface OCL*, item do modelo que abstrai uma invariante OCL, e a *interface Class*. Desta forma toda invariante em OCL é modelada diretamente no seu contexto, ou seja, na Classe onde a restrição se aplica. A segunda é a ligação entre a *interface AlloyCode* e a *interface Package*. Assim podem-se inserir códigos em *Alloy* diretamente no modelo.

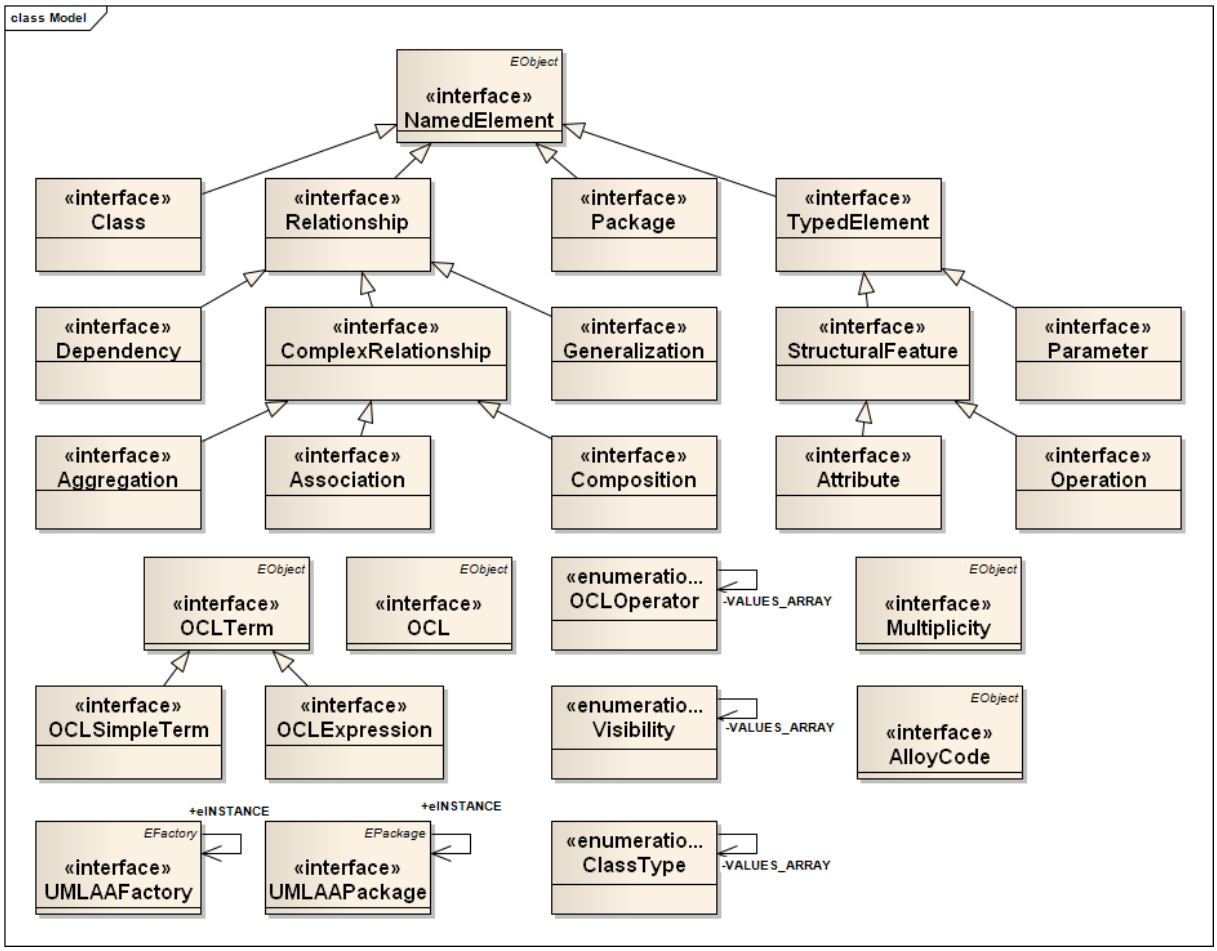

Figura 4. Modelo do *Plug-in* do Modelo

5.2.3 A Implementação do *Plug-in*

A implementação do *plug-in* do Modelo usando EMF é direta e simples. Depois de definidos os elementos de UML necessários usando as classes básicas de EMF no pacote principal, foi criado o pacote de implementações onde as interfaces, mostradas na Figura 4, foram implementadas.

As Interfaces

O código a seguir mostra um exemplo de interface definida usando EMF.

```
package UMLAA;
import org.eclipse.emf.common.util.EList;

public interface Class extends NamedElement {
  EList<Attribute> getAttributes();
  EList<Operation> getOperations();
  EList<Relationship> getRelationships();
  EList<OCL> getOCLInvariant();
  ClassType getType();
  void setType(ClassType value);
}
```

A classe EList é usada para armazenar os elementos de um relacionamento EMF. Ela é usada, por exemplo, no método *getAttributes* que é modelado como um relacionamento EMF entre uma *Class* e os *Attributes* que ela contém. Os atributos EMF são modelados usando seu próprio tipo, porem, este tipo deve ser conhecido pelo *framework* para que os métodos herdados funcionem corretamente. Por isso, no modelo, são definidos todos os tipos usados com base nos tipos do próprio *framework*.

As Implementações

O código da implementação da interface *Class* é mostrada abaixo, com algumas partes omitidas.

```
package UMLAA.impl;
…
import org.eclipse.emf.common.notify.Notification;
public class ClassImpl extends NamedElementImpl implements UMLAA.Class {
        protected EList<Attribute> attributes;
        protected static final ClassType TYPE_EDEFAULT = ClassType.CLASS;
        protected ClassType type = TYPE_EDEFAULT;
        protected ClassImpl() {
                super();
        }
        public EList<Attribute> getAttributes() {
                if (attributes == null) {
                        attributes = new EObjectContainmentEList<Attribute>(Attribute.class,
this, UMLAAPackage.CLASS__ATTRIBUTES);
                }
                return attributes;
        }
        public void setType(ClassType newType) {
                ClassType oldType = type;
                type = newType == null ? TYPE_EDEFAULT : newType;
                if (eNotificationRequired())
                        eNotify(new ENotificationImpl(this, Notification.SET,
UMLAAPackage.CLASS__TYPE, oldType, type));
        }
        …
}
```

As principais características do código do pacote de implementação são:

- O uso da classe *EObjectContainmentEList* que é usada para obter todas as classes pertencentes a um relacionamento.
- A implementação dos métodos *set* que usam os mecanismos de notificação [13] de mudança do *framework*, usado pelo pacote de Edição.
- E o uso da classe *UMLAAPackage*. Esta classe é gerada com a ajuda do *framework* e usando o modelo de interfaces definidos.

As Classes Geradas

O EMF, baseado no modelo de interfaces criado, gera algumas classes que fazem a integração entre o código criado e o *framework*. Estas classes são:

- *UMLAAFactory*: A fabrica [13] de elementos do modelo. Ela fornece métodos de criação para todas as classes não abstratas do modelo.
- *UMLAAPackage*: O pacote de definições EMF para o modelo. Ele contém definições e métodos de acesso às meta-informações de todas as classes, atributos, relacionamentos, etc.
- *UMLAAAdapterFactory* e *UMLAASwitch*: Definem meios de detectar as fabricas e interfaces corretas no modelo para os métodos internos do *framework*.

5.3 O *Plug-in* de Edição

O *plug-in* de Edição tem como objetivo fornecer meios de edição do modelo sem dependência direta com alguma biblioteca gráfica. Segundo o padrão de EMF o *plug-in* de Edição deve conter um conjunto de classes provedoras de itens do modelo, uma para cada item do modelo. Estas classes provedoras são as classes responsáveis pelo acesso e manipulação dos itens do *plug-in do* Modelo pelo *plug-in* do Editor.

5.3.1 As Classes Provedoras

As classes provedoras – *Item Providers* – são as classes mais importantes do *plug-in* de Edição. Elas são usadas para adaptar [13] objetos EMF, fornecendo todas as interfaces necessárias para a visualização e edição de objetos EMF, que são as classes do *plug-in* do Modelo. No *framework*, as classes de adaptação, como as

classes provedoras, são usadas para dar extensão comportamental e para funcionar como observadores [13] de mudança.

Como extensões comportamentais, as classes provedoras podem adaptar os objetos para fornecer qualquer interface que os visualizadores e editores necessitem e ao mesmo tempo, como observadores, elas serão notificadas das mudanças de estado, que elas podem depois passar para os visualizadores aos quais elas estão ligadas.

5.3.2 O Modelo do *Plug-in*

A Figura 5 traz o modelo do *plug-in* de Edição, com alguns itens omitidos.

No modelo estão apenas alguns *Item Providers* e as classes geradas pelo EMF. O nome *Item Provider* vem do fato de que eles fornecem funcionalidades em nome dos itens editáveis, os objetos individuais. Pelo padrão do *framework*, os itens delegam a maior parte de suas funcionalidades aos *Item Providers*. Conseqüentemente, os *Item Providers* precisam exercer quatro papeis principais.

- Fornecer as funcionalidades de acesso a informações de conteúdo e etiquetas.
- Fornecer descritores de propriedades para os objetos EMF.
- Agir como uma fabrica de comandos para os comandos relacionados aos objetos associados.
- Repassar as notificações de mudança EMF para os visualizadores.

Um *Item Provider* pode fornecer todas estas funcionalidades ou apenas um subconjunto delas, dependendo de quais funcionalidades de edição são realmente necessárias. Geralmente, um *Item Provider* simplesmente todos eles herdado de uma classe muito importante do *framework*, a *ItemProviderAdapter*. Ela fornece a maior parte das funções de forma genérica, portanto, uma subclasse precisa apenas implementar alguns métodos para completar o trabalho.

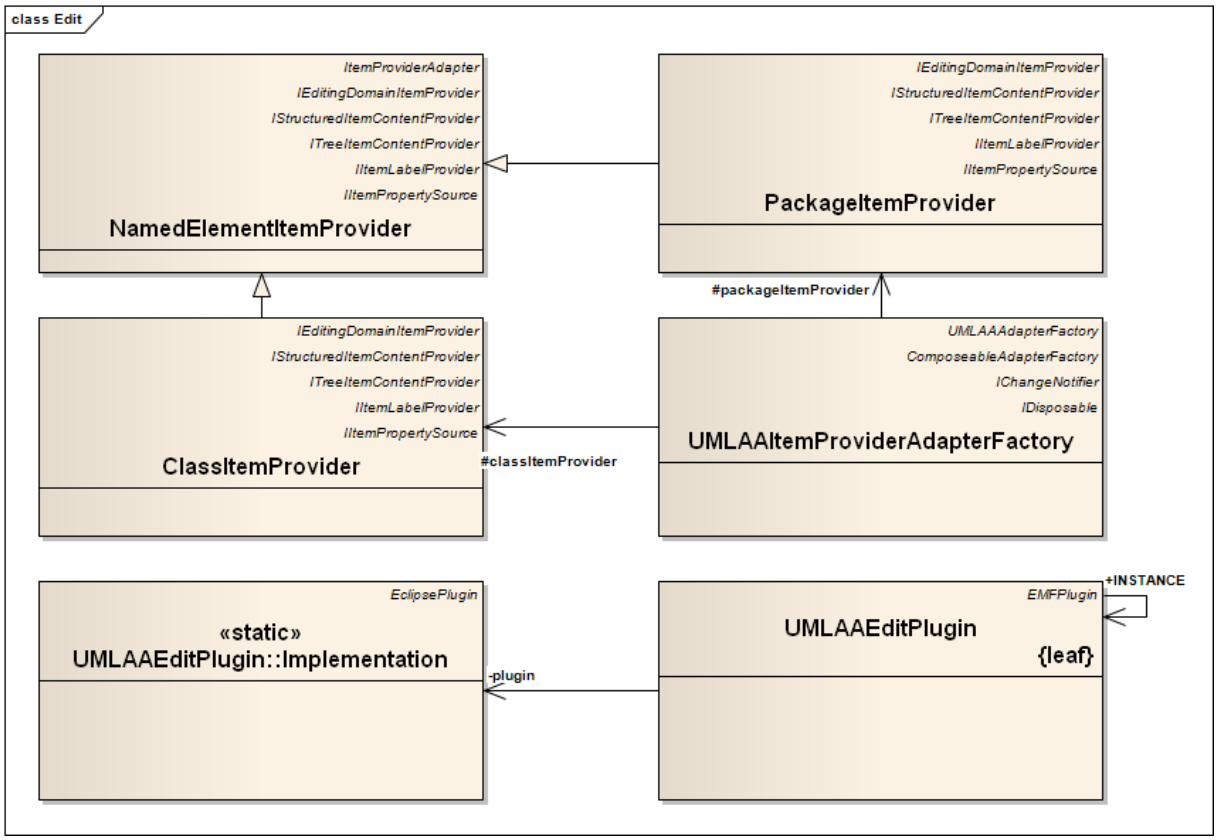

Figura 5. Modelo do *Plug-in* de Edição

5.3.3 A Implementação do *Plug-in*

A seguir são listados alguns trechos de código de um *Item Provider*, o *ClassItemProvider*.

```
package UMLAA.provider;
...
import org.eclipse.emf.common.notify.AdapterFactory;
import org.eclipse.emf.common.notify.Notification;
import org.eclipse.emf.edit.provider.IEditingDomainItemProvider;
import org.eclipse.emf.edit.provider.IItemLabelProvider;
import org.eclipse.emf.edit.provider.IItemPropertySource;
import org.eclipse.emf.edit.provider.IStructuredItemContentProvider;
import org.eclipse.emf.edit.provider.ITreeItemContentProvider;
public class ClassItemProvider
        extends NamedElementItemProvider
        implements
                IEditingDomainItemProvider,
                IStructuredItemContentProvider,
                ITreeItemContentProvider,
                IItemLabelProvider,
                IItemPropertySource {
        public ClassItemProvider(AdapterFactory adapterFactory) {
                super(adapterFactory);
        }
}
```

No trecho acima são mostradas as interfaces que um *Item Provider* implementa. Vale notar que esta classe indiretamente herda da classe *ItemProviderAdapter*. Os *Item Providers* implementam as interfaces *IItemLabelProvider* e *IStructuredItemContentProvider* para fornecer as funcionalidades de acesso a informações de conteúdo e etiqueta.

O trecho a seguir mostra a implementação da função *getChildrenFeatures* da interface *ITreeItemContentProvider*, responsável pela estrutura em arvore do modelo no editor de diagramas do Analisador.

```
public Collection<? extends EStructuralFeature> getChildrenFeatures(Object object) {
        if (childrenFeatures == null) {
                super.getChildrenFeatures(object);
                childrenFeatures.add(UMLAAPackage.Literals.CLASS__ATTRIBUTES);
                childrenFeatures.add(UMLAAPackage.Literals.CLASS__OPERATIONS);
                childrenFeatures.add(UMLAAPackage.Literals.CLASS__RELATIONSHIPS);
                childrenFeatures.add(UMLAAPackage.Literals.CLASS__OCL_INVARIANT);
        }
        return childrenFeatures;
}
```

Em suma, a implementação da interface *ITreeItemContentProvider* visa apenas responder uma pergunta: que itens são filhos da classe associada a este *Item Provider*. Para este fim, as constantes geradas em *UMLAAPackage* são inseridas no atributo *childrenFeatures*. O restante dos métodos declarados na interface são implementados pela classe *ItemProviderAdapter* e são elas: *getParent*, que retorna o objeto que contem o objeto adaptado pelo *Item Provider*; *getChildren*, que itera pelo atributo *childrenFeatures* construindo a coleção de objetos que o objeto adaptado contém; e *hasChildren*, retorna verdadeiro se o atributo *childrenFeatures* não é vazio.

A classe *ItemProviderAdapter* fornece ainda a implementação do método *getElements*, único método da interface *IStructuredItemContentProvider*, ela chama o método *getChildren* e coloca os valores em uma estrutura de tabela. O trecho de código abaixo traz a implementação dos métodos da interface *IItemLabelProvider*.

```
public Object getImage(Object object) {
        return overlayImage(object, getResourceLocator().getImage("full/obj16/Class"));
}
public String getText(Object object) {
        String label = ((UMLAA.Class)object).getName();
        return label == null || label.length() == 0 ?
                getString("_UI_Class_type") :
                getString("_UI_Class_type") + " " + label;
}
```

No método *getImage* a chamada do método *getResourceLocator* retorna a instancia da classe de *plug-in UMLAAEditPlugin*. Esta classe implementa o método *getImage* para retornar uma instancia de uma imagem dentro da pasta *icons* do *plug-in*. O método *overlayImage* é definido na classe *ItemProviderAdapter* e é usado para permitir a criação de camadas que modificam a imagem dependendo do contexto.

O método *getString*, usado no método *getText*, é definido na classe *ItemProviderAdapter* e é responsável por retornar o texto *externalizado* pelo *plug-in* através do *ResourceLocator*, ou seja, uma instancia da classe *UMLAAEditPlugin*. O método *getText* serve para etiquetar o item dentro do modelo. Nesta implementação ele usa um atributo da classe *UMLAA.Class* para formar a etiqueta.

Para que as propriedades e atributos de um objeto EMF sejam visíveis, e se possível, editáveis através de uma folha de propriedades, é necessário que o seu respectivo *Item Provider* implemente os métodos da interface *IPropertyDescriptor*, disponível pela interface *IItemPropertySource*. O trecho de código que implementa esta interface está listado abaixo.

```
public List<IItemPropertyDescriptor> getPropertyDescriptors(Object object) {
        if (itemPropertyDescriptors == null) {
                super.getPropertyDescriptors(object);
                addTypePropertyDescriptor(object);
        }
        return itemPropertyDescriptors;
}
protected void addTypePropertyDescriptor(Object object) {
        itemPropertyDescriptors.add
                (createItemPropertyDescriptor
                ((((ComposeableAdapterFactory)adapterFactory).getRootAdapterFactory(),
                        getResourceLocator(),
                        getString("_UI_Class_type_feature"),
                        getString("_UI_PropertyDescriptor_description",
"_UI_Class_type_feature", "_UI_Class_type"),
                        UMLAAPackage.Literals.CLASS__TYPE,
                        true, false, false,
                        ItemPropertyDescriptor.GENERIC_VALUE_IMAGE,
                        null, null));
}
```

O método *getPropertyDescriptors* constrói uma lista de descritores de propriedades para o objeto EMF associado. Por padrão, ela cria uma lista com os atributos EMF e os relacionamentos que não expressam pertinência. Ele chama o método da superclasse, para adicionar os atributos herdados, e o método *addTypePropertyDescriptor* que chama o método *createItemPropertyDescriptor*,

definido na classe *ItemProviderAdapter*, para adicionar as propriedades do atributo *type*.

O método *createItemPropertyDescriptor* recebe os seguintes parâmetros: a fabrica dos adaptadores, que criou o *Item Provider*; o *ResourceLocator*, usado para recuperar os textos *externalizados* e ícones; a etiqueta da propriedade; a descrição da propriedade, exibida na barra de status; a classe que contem a propriedade; o indicador de edição, verdadeiro se for possível editar a propriedade; o indicador de edição especial, verdadeiro se a propriedade for de texto de múltiplas linhas; um indicador de ordenação, verdadeiro se os valores possíveis para a propriedade devem ser ordenados; o ícone da propriedade; o nome da categoria da propriedade; e um filtro para valores da propriedade.

Outro papel do *Item Provider* é o de repassador de notificação de mudanças sobre os itens do modelo. O código listado a seguir mostra a implementação do método responsável por repassar a notificação de mudança dos objetos a seus filhos e para os observadores.

```
        public void notifyChanged(Notification notification) {
                updateChildren(notification);
                switch (notification.getFeatureID(UMLAA.Class.class)) {
                        case UMLAAPackage.CLASS__TYPE:
                                fireNotifyChanged(new ViewerNotification(notification,
notification.getNotifier(), false, true));
                                return;
                        case UMLAAPackage.CLASS__ATTRIBUTES:
                        case UMLAAPackage.CLASS__OPERATIONS:
                        case UMLAAPackage.CLASS__RELATIONSHIPS:
                        case UMLAAPackage.CLASS__OCL_INVARIANT:
                                fireNotifyChanged(new ViewerNotification(notification,
notification.getNotifier(), true, false));
                                return;
                }
                super.notifyChanged(notification);
        }
```

O método *notifyChanged*, disparado quando o objeto é modificado e o método *eNotify* é chamado, chama *updateChildren*, um método da classe *ItemProviderAdapter* que propaga a notificação aos filhos do objeto. Em seguida ele chama, caso a notificação tenha vindo de uma de suas propriedades, o método fireNotifyChanged, método da classe *ItemProviderAdapter* que envia a notificação às classes do Editor.

A interface *IEditingDomainItemProvider* é importante para a criação de comandos de edição no *plug-in* de Editor. Os quatro métodos desta interface são

getChildren, *getParent*, *getNewChildDescriptors* e *createCommand*. Eles são todos implementados, por padrão, pela classe *ItemProviderAdapter* e são usado na criação dos menus e itens dos menus e comandos associados ao objeto que o *Item Provider* adapta.

Os *Item Providers* determinam quais objetos podem ser criados como filhos dos objetos associados a eles através do método *getNewChildDescriptors*. A implementação deste método na classe *ItemProviderAdapter* cria uma nova lista e passa como parâmetro, junto com o objeto pai, para o método *collectNewChildDescriptors* para ser preenchido.

```
protected void collectNewChildDescriptors(Collection<Object> newChildDescriptors, Object
object) {
        super.collectNewChildDescriptors(newChildDescriptors, object);
        newChildDescriptors.add
                (createChildParameter
                        (UMLAAPackage.Literals.CLASS__ATTRIBUTES,
                         UMLAAFactory.eINSTANCE.createAttribute()));
    ...
    }
```

O método *createChildParameter*, definido pela classe *ItemProviderAdapter*, é usado para criar instancias da classe *CommandParameter* para cada par pai-propriedade filha que é então adicionada à lista de descritores.

As Classes Geradas

O *framework* gera as seguintes classes para o *plug-in* de Edição:

- *UMLAAItemProviderAdapterFactory*: fabrica dos adaptadores *Item Providers*. Cria e gerencia os *Item* Providers do modelo e fornece as interfaces necessárias para suportar visualizadores.
- *UMLAAEditPlugin*: é o singleton [13] central do *plug-in* de Edição. Inclui métodos para recuperar recursos externos como ícones e textos *externalizados*.

5.4 O *Plug-in* do Editor

O *plug-in* do Editor funciona como ponte entre o *plug-in* de Edição e o *framework* de interface com o usuário da Plataforma Eclipse, conhecido como *Platform UI*. O *plug-in* é dependente do *framework* de Interface com o usuário da Plataforma Eclipse. Por padrão, o *plug-in* do Editor contém uma classe *Editor*, que

fornece a interface de edição do modelo, uma classe *ActionBarContributor*, que fornece os menus para a interface, e uma classe *ModelWizard* que representa a tela de dialogo de criação de novos arquivos do modelo.

5.4.1 Plataforma de Interface com o Usuário

O *framework* de interface com o usuário da Plataforma Eclipse, conhecido como *Platform UI*, consiste em dois grupos de ferramentas de propósito geral, SWT e JFace; e uma estrutura personalizável de ambiente gráfico de trabalho, chamado *workbench*. O *framework* do *Platform UI* fornece também uma instancia do *workbench* configurada para ser usada como uma IDE.

SWT

O *Standard Widget Toolkit [18] (SWT)* é um conjunto de componentes e itens gráficos, independentes de plataforma, desenvolvidos usando, sempre que possível, componentes nativos. Diferente do *Abstract Window Toolkit (AWT)* de Java, onde apenas os componentes mais básicos usam componentes nativos, deixando um emulador criar o resto dos componentes. Desta forma, emulando apenas os componentes que não podem ser realizados de forma nativa, SWT fornece uma API portável, mas com a aparência, *look and feel*, mais próxima possível da plataforma nativa.

JFace

JFace [6] é um conjunto de componentes de mais alto nível, desenvolvidos usando SWT. Ele fornece classes para suportar as atividades mais comuns de programação de interfaces com o usuário como gerenciamento de imagens e fontes, caixas de dialogo, monitores de progresso, wizards, etc. A API de JFace não sobrepõe a de SWT, mas trabalha com ela e expande suas funcionalidades.

Uma parte importante de JFaces é o conjunto de classes de visualização. Classes de visualização para listas, árvores e tabelas trabalham com os respectivos componentes de SWT, mas dispõem de conexões de alto nível com repositórios de dados. Estas classes incluem mecanismos para buscar dados em modelos de dados e se manter em sincronia com estes modelos.

Outra parte muito usada de JFaces é o seu *framework* de ações, que é usada para adicionar comandos aos menus e barras de ferramentas. O *framework* permite a criação de ações, para desenvolver um comando do usuário, e usá-la em menus, barras de ferramentas e menus de contexto.

Workbench

O *workbench* [6] é a tela principal que o usuário vê quando executa a IDE Eclipse, ou qualquer aplicação baseada em Eclipse. Ele é desenvolvido usando SWT e JFaces.

Uma janela do *workbench* é composta de editores e visualizadores. Os editores em Eclipse funcionam como um editor qualquer, mas são integrados a uma janela do *workbench*. Visualizadores são utilizados para fornecer mais, ou diferentes informações sobre o conteúdo do editor ativo ou sobre um objeto selecionado no editor ou em outro visualizador. Normalmente, somente uma instancia de um visualizador pode existir em um determinado instante, e ela é atualizada imediatamente baseada no estado do workbench. Da mesma forma, qualquer mudança feita em um visualizador acontece imediatamente, sem a necessidade de um comando para salvar as mudanças. Quando ativos, editores e visualizadores podem adicionar ações no menu e barra de ferramentas do *workbench*.

O arranjo dos editores e visualizadores na janela do *workbench* pode ser personalizado para um determinado papel ou tarefa. Um arranjo padrão é chamado de perspectiva em Eclipse. O usuário pode modificar o arranjo de uma perspectiva de salvá-la para posterior uso.

A principal maneira de estender a Plataforma Eclipse é usando os pontos de extensão disponíveis no *workbench*. Estes pontos de extensão permitem que ferramentas adicionem novos editores, visualizadores ou perspectivas ao *workbench*. Ferramentas podem modificar editores, visualizadores e perspectivas já existentes.

5.4.2 O Modelo do *Plug-in*

A Figura 6 mostra o modelo de classes do *plug-in* do Editor. Os itens que merecem maior atenção são as classes *UMLAAModelWizard*, *UMLAAActionBarListener* e *UMLAAEditor*.

5.4.3 A Implementação do *Plug-in*

A classe *UMLAAEditor* implementa a parte do *workbench* através da qual o usuário pode editar instancias dos modelos. A classe *UMLAAEditor* é parcialmente gerada pelo *framework*. A seguir segue o código que deve ser alterado pelo usuário para que o editor possa trabalhar com arquivos do modelo criado. A modificação

necessária é a inclusão da fabrica de *Item Providers* do *plug-in* de Edição, *UMLAAItemProviderAdapterFactory*, na fabrica *ComposedAdapterFactory*.

```
protected void initializeEditingDomain() {
        BasicCommandStack commandStack = new BasicCommandStack();
        adapterFactory = new ComposedAdapterFactory(
    ComposedAdapterFactory.Descriptor.Registry.INSTANCE);
        adapterFactory.addAdapterFactory(new UMLAAItemProviderAdapterFactory());
        ...
        editingDomain = new AdapterFactoryEditingDomain(adapterFactory, commandStack, new
HashMap<Resource, Boolean>());
    }
```

A classe *UMLAAModelWizard* é completamente gerada, mas foi modificada para criar modelos que aceitem apenas um elemento do tipo *Package* como raiz. O método alterado foi o *getInitialObjectNames*.

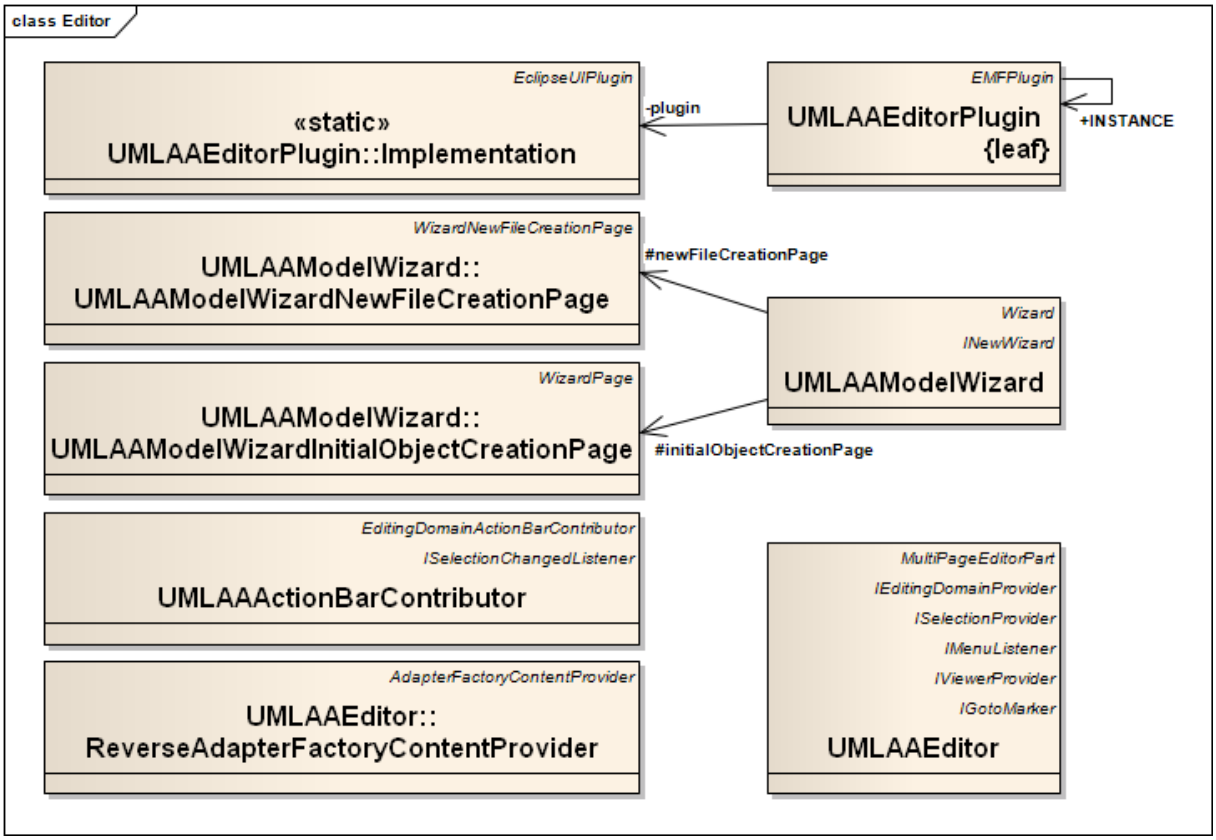

Figura 6. Modelo do *Plug-in* do Editor

Assim como a classe anterior, a classe UMLAAActionBarContributor também é completamente gerada, mas foi modificada para incluir um item no menu de

contexto. O método *addGlobalActions* foi modificado para incluir a ação *JFace alloyAnalyzerAction*, que será mostrado em mais detalhes no próximo capítulo.

Outras Classes Geradas

- Classes Internas: as classes internas são usadas pelas outras classes do *plug-in*.
- *UMLAAEditorPlugin*: é o singleton central do *plug-in* de Edição. Inclui métodos para recuperar recursos externos como ícones e textos *externalizados*.

5.5 Outros Artefatos Gerados

Os arquivos abaixo são gerados pelo *framework* para os três *plug-ins*.

- Os arquivos de manifesto: META-INF/MANIFEST.MF e plugin.xml, que especificam as dependências e extensões dos pontos de extensão do *workbench*.
- O arquivo build.properties, que contém informações para guiar a construção dos *plug-ins*.
- O arquivo plugin.properties, que contém os textos *externalizados* pelas classes dos *plug-ins*.

5.6 Interface do Analisador

A seguir são mostradas algumas das telas do Analisador. Na Figura 7 é mostrada a tela do wizard de criação de novo arquivo do modelo.

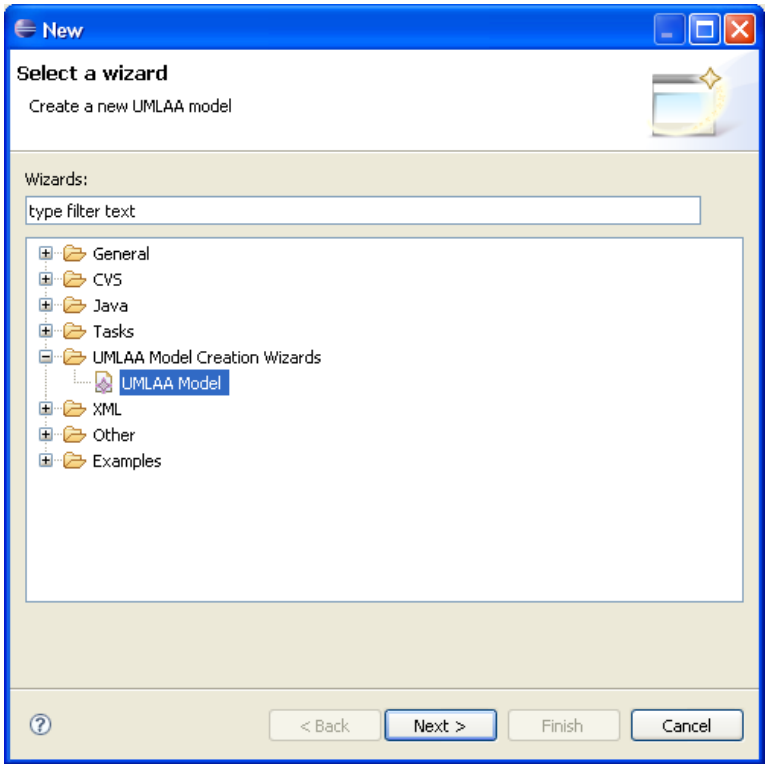

Figura 7. Wizard de Criação de Modelo

Na Figura 8 mostra a tela principal do Analisador dentro da IDE do Eclipse. Nela pode ser vista também o menu de contexto dos elementos do modelo, onde se destacam os itens *New Child* e *New Sibling*, que permitem a inclusão de novos itens, e a opção *Alloy Analyzer*, que permite a geração automática do código correspondente *Alloy* e a sua análise usando *Alloy*.

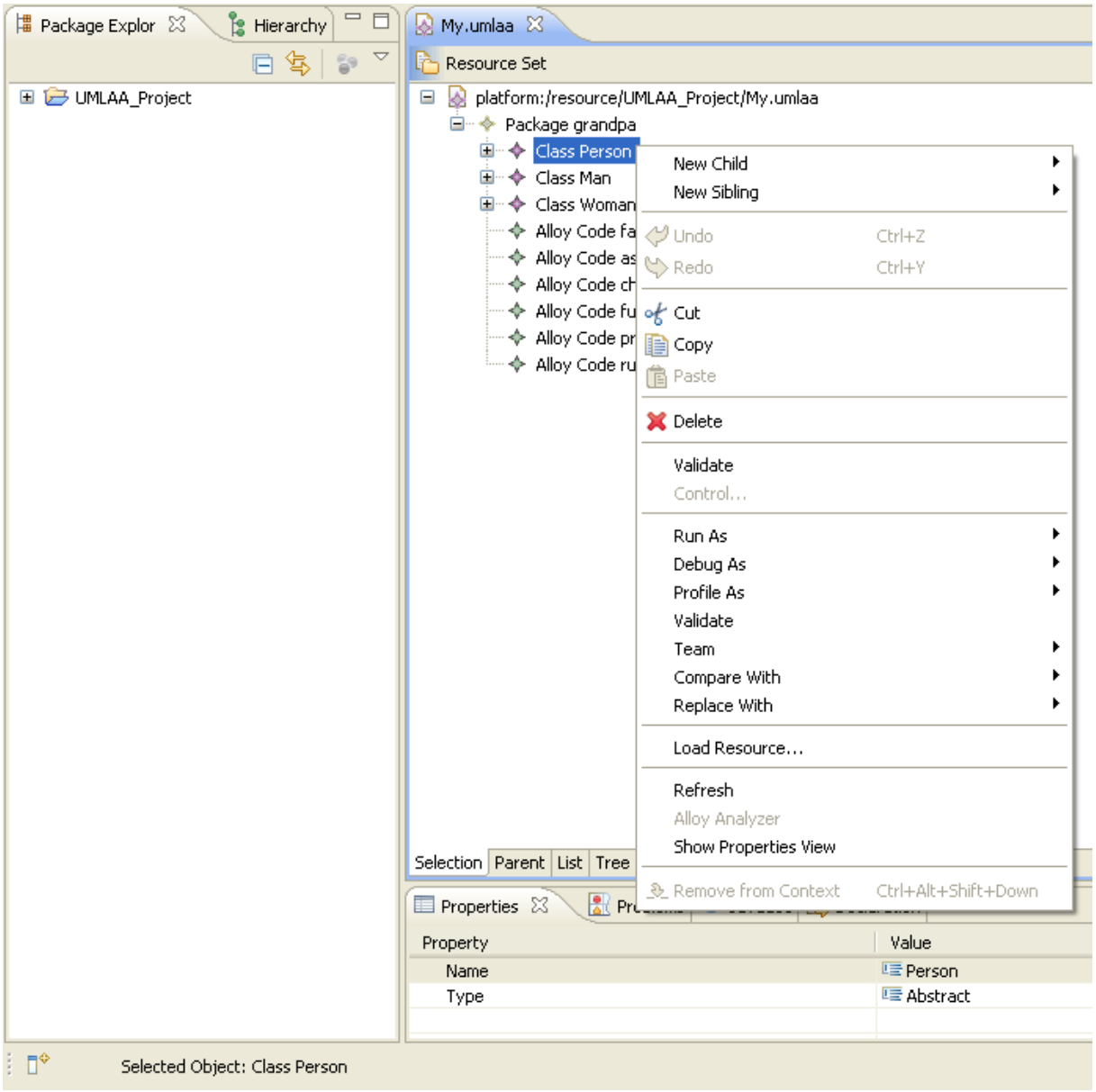

Figura 8. Tela Principal do Analisador

Capítulo 6

Analisando UML com *Alloy*

Ate então foram mostrados os mecanismos utilizados para modelar a linguagem de modelagem UML e o ambiente e infra-estrutura de criação e edição de modelos usando o *framework* EMF e a Plataforma Eclipse. Neste capitulo serão descritas as funcionalidades que possibilitam a transformação de modelos em UML em modelos em *Alloy* e a sua posterior análise através da API do *Alloy Analyzer*.

Para que a transformação entre modelos seja possível, um mapeamento entre as construções de uma linguagem para a outra linguagem foi proposto e implementado dentro do *plug-in* do Modelo.

6.1 Transformando UML em *Alloy*

Para realizar a transformação entre UML e *Alloy* foi criada uma interface no *plug-in* de Edição e todas as classes provedoras que possuem um mapeamento para algum elemento de *Alloy* implementa esta interface. O trecho de código que segue traz a interface *AlloyTranslatable*.

```
package UMLAA.Edit;
public interface AlloyTranslatable {
        public String getAlloyTranslation();
}
```

A interface possui apenas o método *getAlloyTranslation*. Este método é implementado de uma forma que, começando pelo elemento *Package*, todos os elementos mapeados sejam traduzidos em uma única *String*.

6.1.1 O Mapeamento

Nesta seção, as construções da linguagem *Alloy* são mostradas junto com os trechos de código que, a partir de um modelo UML, geram os resultantes em *Alloy* a ser analisado.

Module

Os módulos em *Alloy* são usados para organizar os modelos. Os módulos em *Alloy* funcionam como pacotes de Java. Eles contêm todas as outras estruturas da

linguagem. Assim como em Java, o nome dos módulos corresponde ao nome do arquivo em um sistema de arquivos. Cada módulo representa um arquivo no sistema de arquivos e possuem a extensão ".als", por padrão.

No mapeamento proposto, cada *Package* de um diagrama de classes UML é traduzido em um *Module* em *Alloy*. O código abaixo mostra a implementação do método *getAlloyTranslation* da classe *PackageImpl*.

```
public String getAlloyTranslation() {
        StringBuilder alloyCode = new StringBuilder();
        alloyCode.append("module " + this.name + "\r\n");
        for (UMLAA.Class aClass : this.classes)
                if(aClass instanceof UMLAA.impl.ClassImpl)
                        alloyCode.append(((UMLAA.impl.ClassImpl)
aClass).getAlloyTranslation());
        for (UMLAA.AlloyCode alloy : this.alloyCode)
                if(alloy instanceof UMLAA.impl.AlloyCodeImpl)
                        alloyCode.append(((UMLAA.impl.AlloyCodeImpl)
alloy).getAlloyTranslation());
        return alloyCode.toString();
    }
```

códigos em *Alloy* que não possuem representação em UML, como os comandos *Alloy*.

Signature

Marcadas pela palavra-chave *sig*, cada assinatura representa um conjunto de átomos. A declaração de assinaturas pode ser usada para criar uma hierarquia de conjuntos. As assinaturas podem possuir campos, cada um representando uma relação.

No mapeamento deste trabalho, cada *Class* de um diagrama UML é traduzido em um Signature de *Alloy*. O código no próximo quadro traz a implementação da interface *AlloyTranslatable* na classe *ClassImpl*, com algumas omissões.

Nota-se que, como em *Alloy* herança múltipla não é suportada através da declaração simples, o código tradutor não suporta múltipla herança.

Outras Construções

O restante dos elementos do modelo segue o mesmo padrão dos mostrados ate aqui, cada um fornece a sua tradução ao elemento que o contém ou referencia. A Tabela 6 mostra uma simplificação do mapeamento proposto entre elementos de UML e construções de *Alloy*.

```
public String getAlloyTranslation() {
        StringBuilder alloyCode = new StringBuilder();
        if(this.type != ClassType.CLASS)
                alloyCode.append("abstract ");
        alloyCode.append("sig " + this.name);
        for (UMLAA.Relationship relationship: this.relationships)
                if(relationship instanceof UMLAA.impl.GeneralizationImpl) {
                        alloyCode.append("extends " + relationship.getRelatesTo().getName()+ "{
\r\n");
                        break;}
        for (UMLAA.Attribute attribute: this.attributes)
                if(attribute instanceof UMLAA.impl.AttributeImpl)
                        alloyCode.append(((UMLAA.impl.AttributeImpl)
attribute).getAlloyTranslation());
        ...
        if(!this.oclInvariant.isEmpty()){
                alloyCode.append("}\r\n{");
                for (UMLAA.OCL ocl: this.oclInvariant)
                        if(ocl instanceof UMLAA.impl.OCLImpl)
                                alloyCode.append(((UMLAA.impl.OCLImpl)
ocl).getAlloyTranslation());
        }else
                alloyCode.append("}\r\n");
        return alloyCode.toString();
    }
```

Tabela 6. Mapeamento UML/*Alloy*.

UML	*Alloy*
Package	*Module*
Class	*Signature*
Generalization	*Extends*
Attributes/Relationships	*Signature Fields*
OCL	*Signature Facts*

6.2 O Analisador *Alloy*

Tendo a tradução do modelo UML em *Alloy* o passo seguinte é, usando a API *Alloy4*, realizar a análise do modelo. Através do *Alloy Analyzer*, disponível pela API *Alloy4* é possível compilar e analisar modelos escritos em *Alloy*. O *Alloy Analyzer* é um solucionador de modelos, ou seja, ele procura por modelos para os quais as formulas, o código *Alloy*, sejam verdadeiras.

Na pratica, o *Alloy Analyzer* é um compilador que gera formulas booleanas que são então passadas para um resolvedor de expressões booleanas. As soluções encontradas pelo solucionador são então traduzidas para o contexto de *Alloy* e

retornadas para o usuário. Todas as soluções são procuradas em um escopo definido, o que torna o problema finito.

A seguir são mostrados trechos de código criados para realizar a análise dos modelos UML. A classe modificada para realizar as chamadas à API *Alloy4* foi a *UMLAAActionBarContributor*. No primeiro trecho são definidos os novos atributos da classe *UMLAAActionBarContributor*.

```
        Object aPackage = null;
        VizGUI viz = null;
        protected IAction alloyAnalyzerAction = new Action(UMLAAEditorPlugin.
INSTANCE.getString("_UI_AlloyAnalyzer_menu_item")) {
```

O atributo *aPackage* é usado pra manter uma referencia para o objeto selecionado pelo usuário no modelo. O atributo *viz* é uma instancia da interface gráfica do *Alloy Analyzer*. O atributo *alloyAnalyzerAction* é um *IAction*, uma ação da biblioteca JFace. Para que um IAction possa ser usado em uma interface com o usuário é necessário a definição do método *run*. O método *run* do *alloyAnalyzerAction* é responsável por criar o arquivo em *Alloy* com o código gerado a partir do modelo e chamar a API *Alloy4* para que a análise seja feita.

```
        public void run() {
                try{
                        if (aPackage != null && aPackage instanceof UMLAA.impl.PackageImpl){
                                String alloyFileContent = ((UMLAA.impl.PackageImpl)aPackage).
getAlloyTranslation();
                                EditingDomain domain = ((IEditingDomainProvider)activeEditorPart).
getEditingDomain();
                                String alloyFilePath = domain.getResourceSet().getResources().get(0).
getURI().path();
                                IPath path = ResourcesPlugin.getWorkspace().getRoot().getLocation();
                                path = path.append(alloyFilePath.substring(alloyFilePath.indexOf('/',1),
alloyFilePath.length()-MODEL_FILE_EXTENTION.length()));
                                path = path.addFileExtension(ALLOY_FILE_EXTENTION);
                                alloyFilePath = path.makeAbsolute().toOSString();
                                File alloyFile = new File(alloyFilePath);
                                if(alloyFile.exists())
                                        alloyFile.delete();
                                alloyFile.createNewFile();
                                FileWriter alloyFileWriter = new FileWriter(alloyFile);
                                alloyFileWriter.write(alloyFileContent);
                                alloyFileWriter.close();
```

O trecho acima cria o arquivo *Alloy*, com extensão ".als" com o mesmo nome do arquivo de modelo aberto no editor. A classe *EditingDomain* é uma classe do *framework* EMF que gerencia arquivos e ambiente de edição d arquivos de modelo.

As classes *IPath* e *ResourcesPlugin* são classes do *framework* básico da Plataforma Eclipse.

No trecho abaixo esta o código responsável pela tradução e análise do arquivo criado no trecho anterior. As classes *A4Reporter*, *Module*, *CompUtil*, *A4Options*, *Command* e *A4Solution* são classes da API *Alloy4*.

```
A4Reporter rep = new A4Reporter() {
        @Override
        public void warning(ErrorWarning msg) {
                System.out.print("Relevance Warning:\n"+(msg.toString().trim())+"\n\n");
                System.out.flush();} ... };
edu.mit.csail.sdg.alloy4compiler.parser.Module world = CompUtil.parseEverything_fromFile(rep,
null, alloyFilePath);
A4Options options = new A4Options();
options.solver = A4Options.SatSolver.SAT4J;
for (edu.mit.csail.sdg.alloy4compiler.ast.Command command: world.getAllCommands()) {
        A4Solution ans = TranslateAlloyToKodkod.execute_command(rep,
world.getAllReachableSigs(), command, options);
        if (ans.satisfiable()) {
                ans.writeXML(alloyFilePath+"_output.xml");
                if (viz==null) {
                        viz = new VizGUI(false, alloyFilePath+"_output.xml", null);
                } else {
                        viz.loadXML(alloyFilePath+"_output.xml", true);
                }
        }
}
}}catch(Exception e){String m = e.getMessage(); System.out.print(m);}}
```

A lista a seguir descreve brevemente as classes utilizadas no trecho acima.

- *A4Reporter*: recebem mensagens de diagnostico, de progresso e de avisos das classes da API *Alloy4* os métodos *debug*, *parse* e *warning* foram sobrescritos com o mesmo corpo.
- *Module*: representa um modulo *Alloy*.
- *A4Options*: encapsula as opções personalizáveis do tradutor *Alloy.*
- *Command*: representa comandos *run* ou *check*.
- *A4Solution*: armazena uma solução que satisfaz ou não o comando executado.

Capítulo 7

Conclusão e Trabalhos Futuros

Como já foi citado em capítulos anteriores, a linguagem UML é a linguagem mais usada pela indústria de software para a criação de modelos. Juntamente com OCL, ela é usada para modelagem de sistemas de todos os tamanhos. No entanto, mesmo com a inclusão de OCL à sua especificação, a UML é genérica o suficiente para gerar confusão na análise de modelos de classe. Neste ponto é que entram as linguagens formais de análise de modelos como *Alloy*. *Alloy* fornece meios de especificação de modelos que podem ser analisadas automaticamente.

Nenhuma dessas linguagens, porem, são muito úteis sem uma ferramenta para auxiliar a criação e manutenção de modelo. Desta forma, este trabalho focou na criação de uma ferramenta que unisse esses dois mundos: a facilidade de modelagem usando UML e a análise automática usando *Alloy*.

7.1 Contribuições

Usando os modelos propostos neste trabalho foram criados *plug-ins* para a Plataforma Eclipse que fornecem meios de criação e manutenção de modelos usando UML, alem da tradução destes modelos em modelos *Alloy* e ainda a análise dos modelos *Alloy* usando a API *Alloy4*.

Inicialmente foram criados, usando a Plataforma Eclipse o *framework* EMF, três *plug-ins* para Eclipse que forneceram a infra-estrutura necessária para a criação e manutenção de modelos em UML/OCL. Estes *plug-ins* permitem que o usuário crie e modifique modelos UML visualmente e salve-os em formato XMI.

Em seguida foi criado um mapeamento entre os elementos UML e as construções em *Alloy*. Então foi criada uma interface que fornece um método que retorna a tradução de um elemento UML em *Alloy*. Todos os elementos UML do *plug-in* do Modelo UML que possuem uma tradução em *Alloy* passaram a implementar esta interface.

Por fim, foi usada a API *Alloy4*, fornecida pelos criadores de *Alloy*, para fazer a compilação e análise de modelos. O *plug-in* do Editor foi modificado para incluir a

ação de análise, que além de gravar o arquivo *Alloy* resultante mostra o resultado da análise.

Como a ferramenta pode ser utilizada para criar modelos superficiais ou entrar em mais detalhes da modelagem, usando os mecanismos de ambas linguagens, ela pode ser usada tanto por analistas e engenheiros como por desenvolvedores, *SQEs* e testadores. Cada um desses papeis contribuindo com uma visão particular do modelo.

Além de poder ser usada no mercado de desenvolvimento de software ela pode ainda ser usada no ensino de modelagem e oferece abordagens diferentes da modelagem convencional.

Esta ferramenta é disponível sob a licença Expat [19] – *MIT License* – e é distribuída através do site *SourceForge* [20].

7.2 Trabalhos Futuros

Trabalhos futuros poderiam fazer a inclusão de mais elementos UML no modelo. Com o uso de EMF esta atividade não é difícil já que o próprio *framework* se encarrega da integração dos novos elementos com os já existentes.

A validação da ferramenta em um ambiente educacional ou no mercado de desenvolvimento de software e avaliar a usabilidade da ferramenta.

Outras adições possíveis na parte gráfica do Analisador seriam a criação de um editor visual de modelos que use a notação usual de UML, no lugar de um editor em formato de arvore; uma maior integração entre o Analisador e a API *Alloy4*, com, por exemplo, uma implementação mais sofisticada do *A4Reporter*; uma maneira mais integrada de exibir as soluções *Alloy*, como a criação de uma *view* especifica de *Alloy*.

Outros trabalhos interessantes seriam a criação de mecanismos, usando a biblioteca *KodKod* [14], para fazer engenharia reversa de códigos *Alloy* que gerem diagramas UML e eventual sincronização entre modelos; incluir, usando o EMF, formas de validação de modelos e geração de código a partir do modelo.

Bibliografia

[1] BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The Unified Modeling Language User Guide.* Second Edition. Massachusetts: Addison Wesley Professional, 2005. 496 p.

[2] JACKSON, Daniel. *Software Abstractions: Logic, Language and Analysis.* First Edition. Massachusetts: MIT Press, 2006. 350 p.

[3] PILONE, Dan; PITMAN, Neil. *UML 2.0 in a Nutshell*. First Edition. Sebastopol: O'Reilly, 2005. 216 p.

[4] *Alloy Homepage*. Disponível em: < http://alloy.mit.edu/community/ >. Acesso em: 20 de março de 2009.

[5] *Object Management Group Homepage.* Disponível em: < http://www.omg.org/ >. Acesso em: 20 de março de 2009.

[6] *Eclipse Platform Homepage.* Disponível em: < http://www.eclipse.org/ >. Acesso em: 20 de março de 2009.

[7] *Eclipse Platform – Technical Overview.* Disponível em: < http://www.eclipse.org/whitepapers/eclipse-overview.pdf >. Acesso em: 20 de março de 2009.

[8] CLAYBERG, Eric; RUBEL, Dan. *Eclipse Plug-ins*. Third Edition. Massachusetts: Addison Wesley Professional, 2009. 878 p.

[9] STEINBERG, Dave; BUDINSKY, Frank; PATERNOSTRO, Marcelo; MERKS, Ed. *EMF – Eclipse Modeling Framework*. Second Edition. Massachusetts: Addison Wesley Professional, 2009. 704 p.

[10] DENNIS, Greg; SEATER, Rob. *Alloy Analyzer 4 Tutorial*. Disponível em: <http://alloy.mit.edu/alloy4/tutorial/s1_logic.pdf>. Acesso em: 01 de maio de 2009.

[11] *Object Management Group Homepage*. Disponível em: < http://www.omg.org/>. Acesso em: 01 de maio de 2009.

[12] *World Wide Web Consortium Homepage*. Disponível em: < http://www.w3.org/>. Acesso em: 01 de maio de 2009.

[13] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Illustrated Edition. Massachusetts: Addison Wesley Professional, 1994. 416 p.

[14] KodKod Homepage. Disponível em: <http://alloy.mit.edu/kodkod/>. Acesso em 05 de maio de 2009.

[15] Eclipse Public License - v 1.0 Homepage. Disponível em: <http://www.eclipse.org/legal/epl-v10.html>. Acesso em 05 de maio de 2009.

[16] UML2 Homepage. Disponível em: <www.eclipse.org/uml2>. Acesso em 05 de maio de 2009.

[17] Graphical Editing Framework Homepage. Disponível em: <www.eclipse.org/gef>. Acesso em 05 de maio de 2009.

[18] Standard Widget Toolkit Homepage. Disponível em: <www.eclipse.org/swt>. Acesso em 05 de maio de 2009.

[19] Free Software Foundation licenses Homepage. Disponível em: <http://www.fsf.org/licensing/licenses/>. Acesso em 05 de maio de 2009.

[20] UMLAA Project Homepage. Disponível em: <http://sourceforge.net/projects/umlaa>. Acesso em 05 de maio de 2009.