ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Análise Automática de Diagramas de Classes UML

Trabalho de Conclusão de Curso

Engenharia da Computação

Tássia de Sousa Lima
Orientador: Prof. MSc. Tiago Lima Massoni

Recife, junho de 2007

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Análise Automática de Diagramas de Classes UML

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Tássia de Sousa Lima
Orientador: Prof. MSc. Tiago Lima Massoni

Recife, junho de 2007

Tássia de Sousa Lima

Análise Automática de Diagramas de Classes UML

Resumo

Realizar análises é de fundamental importância para a validação de modelos de *software*. UML é a linguagem mais utilizada por engenheiros de *software* para modelagem e junto com a linguagem OCL, que é utilizada para adicionar restrições aos diagramas, podem construir diagramas de classe que possuam regras mais fiéis ao domínio modelado. No entanto, sua análise vem sendo feita de maneira visual e algumas propriedades podem confundir o analista, como é o caso das multiplicidades dos diagramas de classes e as restrições do OCL. Nesta presente monografia, é apresentado um protótipo de ferramenta que realiza a análise automática de diagramas de classes UML. O protótipo foi desenvolvimento através do mapeamento de construções UML/OCL para construções da linguagem de modelagem *Alloy,* que já possui uma ferramenta de análise bastante eficiente para encontrar erros de modelagem.

Abstract

Automatic analysis of software models is critical for verification and validation. UML is the most widely adopted language for modeling software, and along with the OCL language, which adds constraints to class diagrams, can be used to build models showing more precise domain rules. Nevertheless, these diagrams are usually informally evaluated; issues are often seen in multiplicities and OCL constraint specification. In this work, we present a tool prototype for performing automatic analyses on UML class diagrams. The prototype was developed based on a mapping from UML/OCL constructs to Alloy constructs, a modeling language created along with an analysis tool, which has showed efficiency in finding modeling errors.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

AA – *Alloy Analyzer*
API - *Application Programming Interface*
AST – Árvore Sintática Abstrata
DTD - Definição de Tipos de Documentos
DOM - *Document Object Model*
GUI - *Graphical User Interface*
OCL - *Object Constraint Language*
UML - *Unified Modeling Language*
XML - *EXtensive Markup Language*

Agradecimentos

Primeiramente a Deus que tem me sido fiel nos momentos em que mais precisei e que sem a sua presença seguramente eu não teria chegado onde estou.

A minha mãe Dacilda que é a pessoa mais importante na minha vida, que sempre foi meu porto seguro e fez tudo que podia para proporcionar o meu bem-estar geral.

Ao meu pai Reginaldo e aos meus seis irmãos, pessoas especiais na minha vida, que apesar da distância, sempre que nos encontramos temos momentos muito prazerosos.

Aos meus amigos do tempo de colégio, amigos que sempre estiveram presentes em todos os momentos e que espero levar para o resto da vida. Em especial a Mina, Cinthia, Priscila, Paulinho e Ana.

Aos amigos que a faculdade me proporcionou conhecer e que foram grandes companheiros durante esses cinco anos. Amigos como Emanuel, Juliane, Robson e Fred e também, Gabriel, Renata e Fernando que além de companheiros não me deixaram desistir da monografia.

A Gabi e Lúcia, pessoas que conheci através de Gabriel e que como ele, são muito importantes pra mim.

Aos meus familiares, que sempre estiveram ao meu lado me dando apoio principalmente no último ano, quando eu mais precisei.

Finalizando, aos professores do departamento de computação, principalmente ao professor Tiago Massoni que teve a paciência e disposição para me orientar e que na reta final me deu total apoio no momento em que passei por dificuldades pessoais.

Capítulo 1

Introdução

O principal elemento de um sistema orientado a objetos é a classe. Esta é uma abstração de um conjunto de objetos similares do mundo real que possuem as mesmas características. Diagramas de classes da linguagem *Unified Modeling Language*(UML) [1] são responsáveis por modelar a estrutura do sistema. Todas as classes que o sistema necessita, bem como suas associações são definidas nesses diagramas podendo ser abstratos, onde as classes definem o domínio do problema, ou concretos, onde as classes definem uma implementação em uma linguagem orientada a objetos [2].

Object Constraint Language(OCL) [3] é uma linguagem de especificação formal que não contém qualquer efeito colateral, ou seja, não pode alterar o estado do presente sistema em execução. Sendo assim, os diagramas de classes UML que possuem restrições OCL não serão afetados, mas informações importantes serão adicionadas a eles [3].

UML é a linguagem mais utilizada para modelagem de dados no mercado. Algumas propriedades dos modelos UML/OCL podem confundir um analista no momento dele realizar a análise do mesmo. É o caso, por exemplo, das multiplicidades e restrições do OCL que caso sejam analisadas visualmente, dificulta a busca por erros.

No entanto, apesar do aumento da riqueza de informações, esses modelos UML/OCL são ainda mais difíceis de analisar sem ferramentas como a desenvolvida nessa monografia, pois o esquecimento de restrições ou ainda o uso delas de maneira incorreta podem passar despercebidos se forem realizados de forma não-automática.

Uma outra linguagem também utilizada para criar modelos, conhecida como *Alloy* [4], utiliza-se de código para desenvolver seus modelos e possui uma ferramenta de análise automática, o *Alloy Analyzer*(AA) [4]. Através do AA é possível gerar instâncias e contra-exemplos. As instâncias são geradas para os predicados, produzindo exemplos que podem ser analisados para garantir a corretude do seu modelo, já os contra-exemplos são gerados para provar que o modelo possui algo de errado de acordo com as afirmações que estão sendo checadas [5].

Tendo em vista automatizar a análise de diagramas de classes UML/OCL utilizando as funcionalidades do AA, neste trabalho foi desenvolvida uma estrutura para a correspondência com a linguagem *Alloy*. Essa estrutura foi realizada na linguagem Java [6], utilizando a plataforma eclipse [7] e tornando mais eficiente a procura por erros e conseqüentemente a validação do diagrama.

1.1 Objetivos

Para conseguir realizar a correspondência entre um subconjunto dos diagramas de classes UML/OCL e *Alloy* é necessário atingir alguns objetivos, dentre eles pode-se citar:

- Definição de uma forma representativa dos diagramas de classes UML e restrições do OCL para servir de entrada para o compilador.
- Elaboração das regras de tradução entre as linguagens, para assim poder mapear os elementos de uma linguagem em elementos equivalentes da outra.
- Criação de uma estrutura Java equivalente ao modelo de entrada em formato hierárquico que é montada de acordo com a leitura dos dados de entrada e que representa a Árvore Sintática Abstrata (AST) [8] do compilador.
- Tradução do modelo Java para o código *Alloy* de acordo com a entrada e as regras de tradução.
- Construção do analisador do código *Alloy* fazendo uso das funções da *Application Programming Interface* (API) do *Alloy Analyzer* [4].
- Integração do analisador construído com o compilador, a fim de realizar a análise automática do código *Alloy* gerado pelo compilador.

1.2 Estrutura

Os capítulos da monografia estão organizados da maneira como segue:

O **Capítulo 2** define os conceitos básicos da linguagem UML, descreve rapidamente suas principais características e enfatiza os diagramas de classes e seus componentes por serem eles os elementos mais importantes do presente trabalho. Definem-se também conceitos da linguagem OCL, mostrando a sua estrutura e algumas funções.

O **Capítulo 3** traz uma visão geral da linguagem *Alloy*, definindo suas características principais que a diferenciam das demais linguagens e mostra ainda a estrutura de um código *Alloy* e a lógica do mesmo.

O **Capítulo 4** inicialmente define o mapeamento entre as linguagens. Em seguida, mostra o modelo representativo utilizado como entrada e os passos para o desenvolvimento do compilador. Por fim, traz o modelo do código *Alloy* resultante do compilador.

O **Capítulo 5** define a estrutura da API do *Alloy Analyzer*, seus pacotes e descrição das funcionalidades de algumas de suas classes. Mostra os passos da construção do analisador e como ele foi integrado ao compilador. E ainda, traz exemplos de diagramas de classes UML e geração de contra-exemplos ou instâncias.

O **Capítulo 6** retoma os objetivos, mostra as conclusões e contribuições deste trabalho e ainda os trabalhos futuros que podem ser realizados a partir do protótipo desenvolvido nesta monografia.

O **Apêndice A** mostra o XML gerado pelo contra-exemplo da seção 5.3.1 do Capítulo 5.

O **Apêndice B** mostra o XML gerado pela instância da seção 5.3.2 do Capítulo 5.

Capítulo 2

Unified Modeling Language

A *Unified Modeling Language* (UML) [9] é uma linguagem gráfica universal para visualização, especificação, construção e documentação de artefatos de um sistema intensivo de *software*. A existência de um modelo visual facilita a comunicação e faz com que os membros de um grupo tenham a mesma idéia do sistema. Já a importância da especificação se dá à necessidade de construção de modelos precisos, não ambíguos e completos.

UML não é uma linguagem de programação visual, mas ela pode estar ligada com uma variedade de linguagens de programação. Isto significa que se pode mapear um modelo em UML para uma linguagem de programação como Java. Esse mapeamento permite a geração de código de um modelo UML em uma linguagem de programação ou reconstruir um modelo de uma implementação, voltando para o UML [1].

Além dessas van*tag*ens, UML pode ainda incluir artefatos como *deliverables*, que são documentos como especificação de requisitos, especificações funcionais, planos de testes. E também, materiais que são importantes para controlar, medir, e refletir sobre um sistema durante o seu desenvolvimento e implantação.

2.1 Diagramas de UML

Na versão 2.0 do UML existem 13 tipos de diagramas que são responsáveis por uma parcial representação do sistema. Eles representam graficamente um conjunto de elementos com seus respectivos relacionamentos usados para visualizar o sistema de perspectivas diferentes. Existem dois tipos de diagramas: os diagramas estáticos e os diagramas dinâmicos. Os estáticos são utilizados para visualizar as partes estáticas do sistema e os dinâmicos para modelar o seu comportamento. No projeto em questão é utilizado apenas um tipo de diagrama estático UML, o diagrama de classes [10].

2.2 Diagramas de Classes

Diagramas de classes UML são os diagramas mais comuns na modelagem de sistemas orientados a objetos. Eles possuem um conjunto de classes, interfaces, colaborações e relações. São responsáveis por modelar a visão do design estático do sistema e formam a base para os demais diagramas. Além disso, eles podem ser abstratos, onde as classes definem o domínio do problema, ou concretos, definindo uma implementação em uma linguagem orientada a objetos.

2.3 Classes

Uma classe é uma abstração de um conjunto de objetos que possuem os mesmos atributos, operações, relações e semântica. As classes têm uma função muito importante na modelagem orientada a objetos, pois elas dividem o problema, modularizam a aplicação e baixam o nível de acoplamento do *software* [1].

Representa-se uma classe por um retângulo geralmente dividido em três partes compostas por nome, atributos e operações, respectivamente. Na Figura 2-1 tem-se a ilustração da notação de classe mais geral onde na primeira divisão está presente o nome, na segunda os atributos e na terceira as operações**.**

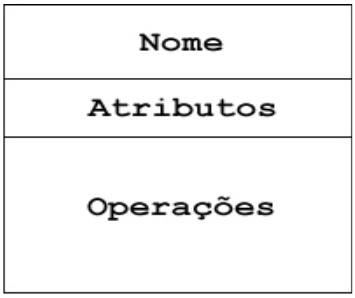

Figura 2-1 Notação de Classe

2.3.1 Nome

Toda classe deve ter um nome para distingui-la das outras. Este nome é uma string textual que pode aparecer sozinho ou acompanhado de um caminho representado pelo nome da classe prefixada pelo nome do pacote em que a mesma está contida.

2.3.2 Atributos

Um atributo representa alguma propriedade que é compartilhada por todos os objetos de uma classe e descreve os dados contidos nas suas instâncias. Cada objeto terá valores particulares para seus atributos, podendo estes, mudar com o tempo.

Sintaxe de Atributos

Em sua forma mais completa, a sintaxe de um atributo em UML é representada por:

```
[visibilidade]nome[:tipo][= valor inicial]
```

A visibilidade pode ser pública, protegida ou privada. Na pública, representa-se pelo símbolo "+" e o atributo é acessível a qualquer outro objeto ou classe. Já na protegida, representa-se pelo símbolo "#" e o atributo é acessível apenas na classe e nas subclasses. Por fim, na privada, representa-se pelo símbolo "-" e o atributo só é acessível dentro da classe em que foi definido.

2.3.3 Operações

Uma operação é a implementação de um serviço que pode ser requisitado por algum objeto de uma classe para afetar seu comportamento. Em outras palavras, uma operação é uma abstração de algo que se pode fazer para um objeto e que é compartilhado por todos os objetos desta classe.

Sintaxe para Operações

Em sua forma mais completa, a sintaxe de uma operação em UML é representada por:

```
[visibilidade]nome[(lista-de-parâmetros)][:tipo de retorno]
```

2.4 Relacionamentos

Um relacionamento é uma conexão entre classes. Em modelagem orientada a objetos, os três relacionamentos mais importantes são dependência, generalização e associação. Graficamente, um relacionamento é representado por um caminho, com diferentes tipos de linhas usadas para distinguir os tipos de relações.

2.4.1 Dependência

Uma dependência é um relacionamento onde uma mudança em uma especificação de um elemento pode afetar outro que o use, mas não necessariamente o reverso. Uma dependência é representada por uma linha tracejada, direcionada para a classe sobre a qual se depende como ilustra a Figura 2-2. Nesta figura tem-se uma dependência da classe **Cliente** em relação à classe **Fornecedor**, em que caso esta seja alterada, pode-se ter mudanças também na classe **Cliente**.

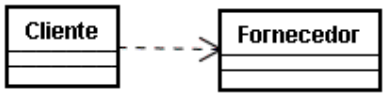

Figura 2-2 Exemplo de Dependência

2.4.2 Generalização

Uma generalização é um relacionamento entre uma classe geral e um tipo mais específico desta classe. Generalização significa que objetos de uma subclasse podem ser usados em qualquer classe mais abstrata, mas não o reverso. O seguinte exemplo da Figura 2-3 ilustra uma classe mais geral de nome **Pessoa** e, duas classes mais específicas nomeadas por **Mulher** e **Homem** que são tipos de **Pessoa**.

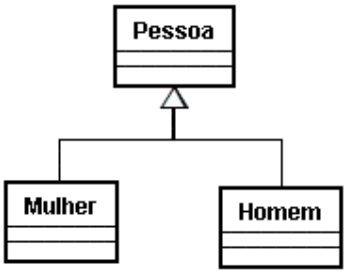

Figura 2-3 Exemplo de Generalização

2.4.3 Associação

Uma associação é um relacionamento estrutural que especifica que objetos de um elemento estão conectados a objetos de outro elemento. Dada uma associação conectando duas classes, pode-se navegar de um objeto de uma classe para outro objeto de outra classe, e vice versa. Uma associação que conecta exatamente duas classes é chamada de associação binária e não é comum ter associações que conectem mais do que essa quantidade, mas quando acontecem, elas são chamadas de associações n-ária.

Representa-se uma associação por uma linha sólida conectando as classes relacionadas. No caso da Figura 2-4, essa representação é dada pela linha que une as classes **Professor** e **Universidade.**

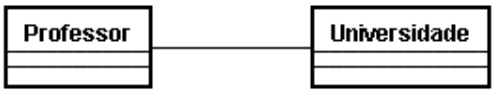

Figura 2-4 Exemplo de Associação

Navegabilidade

A navegabilidade entre as classes de uma associação é bi-direcional porém, pode-se limitá-la a apenas uma direção. Isso significa uma forma de mostrar acesso direto a objetos. No caso da Figura 2-5, **Estudante** tem acesso direto a objetos de **Disciplina.**

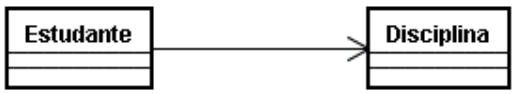

Figura 2-5 Exemplo de Navegabilidade

Multiplicidade

A multiplicidade define quantos elementos participam do relacionamento, ou seja, o número de instâncias de uma classe relacionada a uma instância de outra. Ela é especificada em cada extremidade da associação. Na Figura 2-6 estão representados os principais tipos de multiplicidade e o que cada um representa. Quando a multiplicidade possui **".."** entre os valores, significa que a quantidade de elementos que participarão do relacionamento está dentro do intervalo delimitado por esses números. No caso do uso de **","** ou do uso de um único valor, a quantidade dos elementos deverá ser exatamente um desses valores. Por fim, se a multiplicidade

possui o valor **"0..*"** ou **"*"**, qualquer quantidade de elementos da classe poderá participar da relação e caso seja **"1..*"**, poderá participar qualquer quantidade maior do que 1.

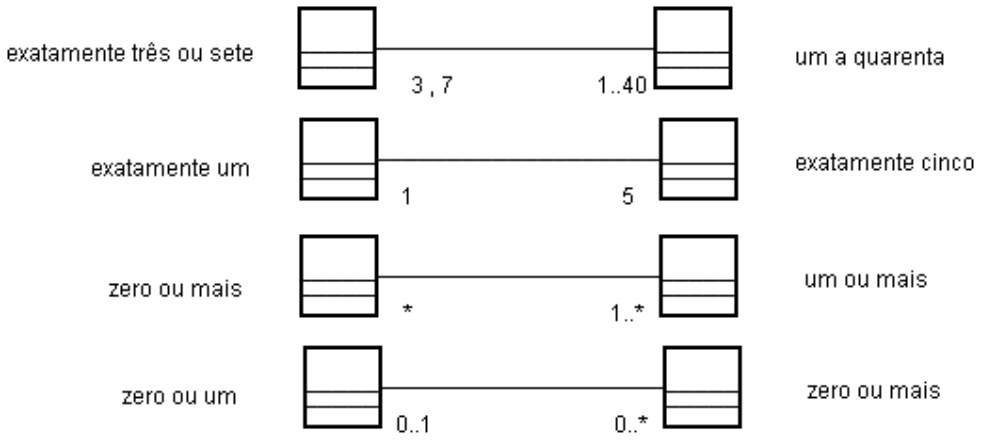

Figura 2-6 Multiplicidades

Papéis

Quando uma classe participa de uma associação, ela tem um papel específico no relacionamento. Normalmente coloca-se explicitamente o nome do papel quando a associação relaciona dois elementos da mesma classe, quando há mais de uma associação entre as classes ou para facilitar o entendimento geral. Na Figura 2-7 os papéis **chefe** e **subordinado** são utilizados para diferenciar qual papel os elementos da classe **Pessoa** assumem. Já o papel **empregador**, assumido pelos elementos da classe **Empresa,** é utilizado para facilitar o entendimento geral do diagrama.

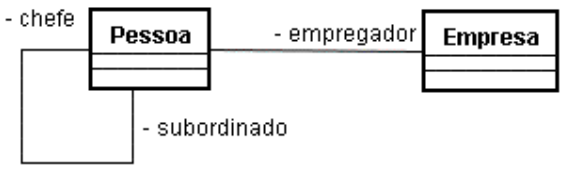

Figura 2-7 Exemplo de Papéis

2.4.4 Agregação e Composição

Às vezes é preciso modelar um relacionamento "todo/parte", em que uma classe represente um elemento maior e que consiste de elementos menores. Esse é um tipo especial de associação chamado de **agregação** e é especificado adornando-se uma associação plena com um diamante aberto na extremidade do todo. A Figura 2-8 traz um exemplo de agregação em que o todo é representado pela classe **Curso** e a parte pela classe **Aluno**.

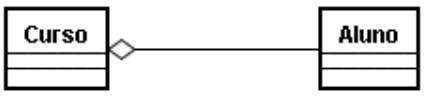

Figura 2-8 Exemplo de Agregação

A **composição** é uma variação da agregação simples onde uma vez criada a parte, ela irá viver e morrer com o todo. Em uma agregação composta, um objeto só pode ser parte de uma composição por vez. Isto é um contraste com a agregação simples, em que uma parte pode ser compartilhada por diversos todos. O todo é o responsável pelo gerenciamento da criação e destruição das partes. Representa-se esse tipo especial de associação adornando a associação plena com um diamante preenchido na extremidade do todo. A classe **Banco** da Figura 2-9 representa o todo da relação e sem ela, não existiria a parte, ou seja, a classe **Conta**.

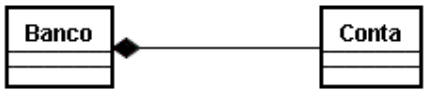

Figura 2-9 Exemplo de Composição

2.5 *Object Constraint Language*

A *Object Constraint Language* (OCL) [3] é uma linguagem formal usada para descrever expressões em modelos UML. Expressões escritas em OCL adicionam vitais informações que não podem ser expressas em um diagrama para modelos OO [11], e outros artefatos de modelagem de objetos. OCL é uma linguagem universal em que as expressões podem ser escritas de maneira clara e não ambígua.

Expressões escritas em uma precisa linguagem baseada na matemática como OCL não podem ser interpretadas diferentemente por pessoas distintas, como um analista e um programador. Por não serem ambíguas, fazem o modelo mais preciso e detalhado, além de poderem ser verificadas por ferramentas automáticas para assegurar que estão corretas e consistentes com outros elementos do modelo. A geração de código torna-se assim, muito mais poderosa [12].

Entretanto, modelos escritos em linguagens que usem uma representação de expressão sozinha não são facilmente compreendidos, ou seja, muitas pessoas preferem um modelo diagramático, pois facilita o entendimento do sistema. Para obter um modelo completo, diagramas e expressões OCL são necessários. Sem expressões OCL o modelo seria pouco especificado; sem os diagramas de UML, as expressões do OCL consultariam aos elementos do modelo não-existentes, porque não há nenhuma maneira de OCL especificar classes e associações.

2.5.1 Contexto

A definição de contexto especifica uma quantificação universal da entidade do modelo para a qual a expressão do OCL é definida. Geralmente, esta é uma classe, uma relação, um tipo de dado ou um componente. Às vezes é uma operação, e raramente é uma instância. É sempre um elemento específico definido em um diagrama de UML.

As expressões do OCL podem ser incorporadas ao modelo diretamente nos diagramas, mas podem também ser fornecidas em um arquivo de texto separado. Ambos os casos possuem uma definição de contexto. No diagrama, a definição do contexto é mostrada por uma linha pontilhada que ligue o elemento do modelo com a expressão do OCL. Quando a expressão do OCL é dada em um arquivo separado, a definição do contexto é dada em um formato textual. É denotada pela palavra-chave **`context`** seguido pelo nome do tipo.

```
context Pessoa
```

2.5.2 Invariantes

Mais informações podem ser adicionadas ao modelo na forma de invariantes. Um invariante é uma restrição que deva ser verdadeira para um objeto durante toda sua vida. Invariantes representam freqüentemente as regras que devem vincular os objetos da vida real depois que os objetos de *software* são modelados.

Invariantes sobre atributos

Invariantes em um ou em mais atributos de uma classe podem ser expressos de uma maneira muito simples. Inicialmente tem-se a classe a que o invariante consulta que é o contexto do mesmo, seguida pela palavra-chave **`inv`** acompanhada opcionalmente por um nome, e por fim uma expressão booleana que indique seu invariante. Todos os atributos da classe do contexto podem ser usados neste invariante. No exemplo seguinte, tem-se como contexto a classe **`Cliente`**, **`maioridade`** como nome do invariante e **`idade >=18`** como a expressão booleana.

```
context Cliente
inv maioridade: idade >= 18
```

Invariantes sobre objetos associados

Invariantes podem também indicar regras para objetos associados. Isto é feito usando os nomes dos papéis da associação que irão consultar o objeto na outra extremidade. Se não possuir o nome do papel, deve-se usar o nome da classe. Tendo-se agora o contexto **`ClienteCartao`**, e o nome do invariante **`maioridade`**, através do nome do papel **`proprietário`** consegue-se consultar os objetos da classe a que ele corresponde e assim aplicar a expressão booleana.

```
context ClienteCartao
inv maioridade: proprietario.idade >= 18
```

2.5.3 Self

Às vezes é necessário referenciar explicitamente a instância contextual em uma expressão do OCL. A palavra-chave **`self`** é utilizada para esse propósito. Se, por exemplo, pretende-se especificar que em uma determinada instância do contexto **`Pessoa`** o atributo **`nome`** receberia o valor **Maria**, a seguinte expressão do OCL seria utilizada para essa definição:

```
context Pessoa
inv: self.nome = 'Maria'
```

Enumerações

Em um modelo de UML, os tipos da enumeração podem ser definidos. Os valores de um tipo da enumeração são indicados em uma expressão do OCL pelo nome do tipo da enumeração, seguido por um duplo sinal de dois pontos e pelo valor. Por exemplo, pode-se querer distinguir associados pelo nível atual de cor que eles possuem onde a cor do atributo pode ter dois valores, prata ou

ouro. Os seguintes invariantes mostram que a cor dos cartões deve combinar com o nível de serviço dos associados.

```
context Associado
inv nivelCor:
  nivelAtual.nome = 'Prata' implies cartao.cor = Cor::prata
  and
  nivelAtual.nome = 'Ouro' implies cartao.cor = Cor::ouro
```

2.5.4 Coleções de objetos

Sempre que a navegação resulta em uma coleção de objetos, pode-se usar uma das operações de coleção para manipulá-las. Para indicar o uso de uma das operações predefinidas da coleção, coloca-se uma seta entre o nome do papel e a operação. Quando se usa uma operação definida no modelo de UML, utiliza-se um ponto. Alguns dos operadores e exemplos deles são indicados a seguir.

Size

No contexto **Pessoa**, o invariante garante que a quantidade de empregadores será menor do que três.

```
context Pessoa
inv : self.empregador->size()<3
```

Select e *reject*

No contexto **Pessoa**, o primeiro invariante seleciona os empregados com idade maior que 50 e analogamente o segundo rejeita-os.

```
context Pessoa
inv: self.empregado->select(idade>50)

context Pessoa
inv: self.empregado->reject(idade>50)
```

ForAll e *exists*

No contexto **Companhia**, o primeiro invariante avalia se a expressão **idade<=65** é verdadeira para todos os elementos do conjunto **empregado** e o segundo se existe algum elemento que satisfaça a mesma expressão booleana.

```
context Companhia
inv:  empregado->forAll(idade<=65)

context Companhia
inv:  empregado->exists(idade<=65)
```

Collect

No contexto **Companhia**, ambos invariantes irão coletar os salários dos empregados, sendo a segunda forma mais comum e tendo as duas o mesmo significado.

```
context Companhia
inv: self.empregado->collect(salario)

context Companhia
inv: self.empregado.salario
```

NotEmpty e *isEmpty*

No contexto **Pessoa**, avalia-se se o conjunto de empregadores não é vazio e analogamente se o mesmo é vazio.

```
context Pessoa
inv: self.empregador->notEmpty()

context Pessoa
inv: self.empregador->isEmpty()
```

2.5.5 *Sets*, *Bags*, *OrderedSets* e *Sequences*

Ao trabalhar com coleções de objetos, deve-se estar ciente da diferença entre um *Set*, um *Bag*, um *OrderedSet*, e uma *Sequence*. Em um *Set*, cada elemento pode ocorrer somente uma vez. Em um *Bag*, os elementos podem aparecer mais de uma vez. Uma *Sequence* é uma coleção em que os elementos são ordenados e pode haver repetição. Um *OrderedSet* é um conjunto em que os elementos são ordenados.

```
Set {1, 2, 3, 4, 5, 6}
Bag {1, 1, 2, 2, 4, 5, 6}
Sequence {2, 1, 2, 3, 5, 6, 4}
OrderedSet{12, 9, 6, 3}
```

2.5.6 Expressão *let*

Às vezes escrevem-se expressões grandes em que uma expressão secundária é usada mais de uma vez. A expressão **let** permite definir uma variável que possa ser usada em vez da expressão secundária. No seguinte exemplo, cria-se a variável **renda** do tipo **Integer**, para armazenar a soma dos salários de uma pessoa. Essa variável é então utilizada duas vezes no decorrer do invariante.

```
context Pessoa
inv: let renda : Integer = self.trabalho.salario->sum in
if self.ehDesempregado then renda < 100
else
     renda >= 100
end if
```

Capítulo 3

Alloy

Alloy [5] é uma linguagem para modelagem de projeto de *software* relacionada com a linguagem Z [13], que combina lógica de predicados com álgebra relacional. Ela possui algumas características únicas como *signature* e a noção de escopo, bem como características universais das linguagens de programação. Além disso, faz uso da análise inteiramente automática, através do *Alloy Analyzer*(AA) [4], que possui um visualizador para exibição das soluções e contra-exemplos gerados. Nessa monografia foi utilizado o *Alloy* na versão 4.

Existem alguns pontos chaves que diferenciam *Alloy* das demais linguagens e técnicas de modelagem. Dentre eles pode-se destacar [4]:

- Verificação de um espaço finito – uma vez que se vai analisar realmente o modelo, deve-se especificar um escopo para ele. A análise é correta por nunca retornar falsos positivos, mas incompleta por verificar até um certo espaço especificado. Entretanto essa análise está correta até esse espaço, pois nunca falta um contra-exemplo que seja menor que o mesmo.
- Modelo infinito – os modelos que se escrevem em *Alloy* não refletem o fato que a análise é finita. Isto é, descrevem-se os componentes do sistema e como eles interagem, mas não se especifica quantos componentes devem ser (como é feito no tradicional "modelo de verificação").
- Declarativo – um modelo declarativo responde à pergunta "como eu reconheceria que X aconteceu" ao contrário de um modelo "operacional" ou "imperativo", modelo que pergunta "como eu posso realizar X".
- Análise automática – ao contrário de outras linguagens declarativas de especificação, a *Alloy* pode ser analisada automaticamente. Podem-se gerar exemplos automaticamente de seus sistemas e contra-exemplos as reivindicações feitas sobre eles.
- Dados estruturados – *Alloy* suporta estruturas de dados complexas tais como árvores, e é assim uma maneira rica de descrever um estado.

3.1 Estrutura

A Figura 3-1 ilustra um exemplo de código *Alloy* com o intuito de conferir se uma pessoa é o seu próprio avô através de um predicado, gerando assim um exemplo, e checar se a pessoa não é o seu próprio pai gerando um contra-exemplo caso essa afirmativa seja falsa.

```
module avo

abstract sig Pessoa{
     pai: lone Homem,
     mãe: lone Mulher
}

sig Homem extends Pessoa {esposa: lone Mulher}
sig Mulher extends Pessoa {marido: lone Homem}

fact Terminologia {esposa = ~marido}
fact ConvencaoSocial {
     no esposa&(mae+pai).mae
     no marido&(mãe+pai).pai
}

fun avos [p: Pessoa] : set Pessoa {
     let pais = mãe+pai+pai.esposa+mãe.marido|
          p.pais.pais&Homem
}

pred proprioAvo [h: Homem] { h in avos[h]}

assert proprioPai{
     all p: Pessoa| no p.pai = p
}

run proprioAvo for 4 Pessoa

check proprioPai for 5
```

Figura 3-1 Código *Alloy*

A estrutura de um modelo descrito em *Alloy* consiste de um módulo principal, declarações de assinaturas, parágrafos de restrições, afirmações e comandos, que serão descritos nesta presente seção.

3.1.1 Módulo

A primeira linha de um modelo é a declaração do módulo constituída da palavra **`module`** seguida pelo seu nome. Eles são nomeados como em Java: o nome completo do mesmo corresponde ao caminho e nome do arquivo no sistema de arquivo.

```
module avo
```

Módulos *Alloy* têm a extensão de arquivo ".als" por *default*, então esse módulo é armazenado no arquivo "avo.als", relativamente ao diretório de trabalho do AA.

3.1.2 Assinaturas

Uma assinatura representa um conjunto de átomos e pode também introduzir alguns campos, cada um representando uma relação. A forma mais simples que se tem de declaração da assinatura é feita através da palavra-chave **`sig`** seguida pelo nome da assinatura e por um par de chaves.

```
sig Pessoa {}
```

Dentro do par de chaves de cada declaração da assinatura tem-se o corpo da assinatura. Nele pode-se definir uma série de relações para o qual o conjunto definido na declaração de assinatura seja o domínio. Por exemplo, pode-se criar uma assinatura **`Homem`** que possua em seu corpo uma relação **`esposa`** relacionando homens com mulheres. A palavra-chave **`lone`** indica simplesmente que existe 0 ou 1 objetos **`esposa`** para cada **`Homem`**. Já a palavra **`extends`** indica que tanto **`Homem`** quanto **`Mulher`** são subtipos de **`Pessoa`**. Isso significa duas coisas: o conjunto de todos os homens é um subconjunto de todas as pessoas e **`Homem`** é disjunto de outro subtipo de **`Pessoa`** (i.e. **`Homem`** é disjunto de **`Mulher`**).

```
sig Homem extends Pessoa { esposa: lone Mulher }
sig Mulher extends Pessoa { marido: lone Homem }
```

Outra palavra-chave importante na declaração de assinaturas é a palavra **`abstract`**. Usando esta palavra na declaração de **`Pessoa`** garante-se que as assinaturas que as estendem herdarão suas propriedades. Assim, não haverá nenhuma pessoa que não pertença a suas extensões, se omite-se a declaração, poderá ter uma pessoa que nem seja homem nem mulher.

```
abstract sig Pessoa {
     pai: lone Homem,
     mae: lone Mulher
}
```

3.1.3 Restrições

Não é tão simples definir as assinaturas como foi feito anteriormente. É necessário também adicionar restrições básicas para certificar que elas se comportem da maneira como se espera intuitivamente. As restrições são representadas pelas palavras chaves *fact*, *fun* e *pred*, e são responsáveis por vários tipos de restrições e expressões.

Fato

Um fato registra uma restrição que assume ser sempre possível. Uma indicação **`fact`** em *Alloy* coloca uma restrição explicita (ou uma lista de restrições) sobre o modelo. Quando *Alloy* procura por exemplos, ele descarta algum que viole o **`fact`**. Assim, se o fato for trivialmente falso, então simplesmente não irá gerar exemplos, no entanto não será dito que o modelo é inconsistente. Um

exemplo de fato com uma única restrição seria dizer que **marido** é o inverso de **esposa**, em termos de valores de relação seria dizer que **marido** tem a imagem refletida de **esposa**.

```
fact Terminologia { esposa = ~marido }
```

Outro exemplo de fato, agora com mais de uma restrição, seria dizer que **esposa** não pode estar no conjunto das mães dos seus pais, ou seja, suas avós e, respectivamente, que **marido** não pode estar no conjunto de pais dos seus pais, ou seja, seus avôs.

```
fact ConvencaoSocial {
     no esposa & *(mae+pai).mae
     no marido & *(mae+pai).pai
}
```

Funções e Predicados

Uma função define uma expressão reusável e um predicado define uma restrição reusável. Funções e predicados podem servir como "fatos opcionais" permitindo que se tenham ocorrências como "se o construtor A acontece então o construtor B acontece". Uma função avalia um valor. Uma construção similar é um predicado onde, se todas as entradas satisfazem todos os construtores listados no corpo, então o predicado é avaliado como verdadeiro. Caso contrário é avaliado como falso.

No exemplo seguinte, o desafio é encontrar um homem que seja seu próprio avô. Para isso é criada uma função **avos** que recebe uma pessoa e retorna um conjunto de pessoas formado pelos pais dos pais da mesma, e que sejam homens. Essa função é utilizada no predicado proprioAvo que recebe um homem e avalia se o mesmo está contido no conjunto resultante dos avôs dele.

```
fun avos [p: Pessoa]: set Pessoa {
     let pais = mae + pai + pai.esposa + mae.marido |
          p.pais.pais & Homem
}

pred proprioAvo [m: Homem] { m in avos[m]  }
```

3.1.4 Afirmações

As afirmações registram as propriedades que se esperam atingir. São representadas pela palavra **assert** e, ao contrário do fato que força algo a ser verdadeiro no modelo, elas reivindicam que algo deve ser verdadeiro devido ao comportamento do modelo. Pode-se querer conferir, por exemplo, se uma pessoa não é o seu próprio pai. Em palavras, afirma-se que "para todo **p** do tipo pessoa, o pai de **p** não deve ser igual a **p**".

```
assert proprioPai {
     all p:Pessoa| no p.pai = p
}
```

3.1.5 Comandos

Existem dois tipos de comandos em *Alloy*, representados pelas palavras-chaves **`run`** e **`check`**, e que são utilizados pelo AA para avaliar a performance do modelo, através da geração de soluções e contra-exemplos.

Run

O comando **`run`** instrui o AA a procurar uma solução para a restrição. Esse comando é importante, pois é através dele que se podem ver exemplos de que o modelo está correto. Também no caso de uma modelagem errada, modelos fora da normalidade serão gerados, ajudando numa melhor percepção de onde deve haver modificações em busca da corretude do mesmo. No seguinte exemplo o comando **`run`** executará o predicado **`proprioAvo`** em busca de possíveis soluções em um universo de quatro pessoas.

```
run proprioAvo for 4 Pessoa
```

Check

O comando *check* chama o AA para procurar contra-exemplos, que é uma instância que prove que a afirmação é falsa. Quando ele é executado têm-se dois possíveis resultados: *no solution found* – não há contra exemplos para a afirmação com o especificado escopo ou menor e, *solution found* – o AA encontrou um contra-exemplo. O ideal é que não sejam encontrados contra-exemplos mostrando que a afirmação está correta no escopo especificado. O comando *check* que segue, buscará contra-exemplos a afirmação *proprioPai* para um universo de tamanho cinco.

```
check proprioPai for 5
```

3.2 Lógica

Na lógica de *Alloy*, só objetos são átomos indivisíveis. Embora eles sejam imutáveis, pode-se modelar mutações em que o valor de um objeto mude todo o tempo para separar o identificador do objeto e seu valor em átomos diferentes, e relatar identificador, valores e tempos. Quando um modelo concede só um simples objeto e o valor é mudado, um conjunto de átomos pode ser usado pelos objetos para representar seu valor em diferentes tempos.

Embora átomos sejam uninterpretados, é possível pegar algumas propriedades para introduzir relações entre elas. Em *Alloy* não existe representação para relações contendo outras relações, mas pode-se trabalhar com átomos interpretando-os como inteiros. As relações não podem ter infinitos tamanhos e larguras o que não é um grande problema, pois normalmente os tamanhos são finitos.

Uma relação binária que mapeia cada átomo em no máximo um outro átomo é chamada de funcional e a que mapeia no máximo um átomo em cada átomo é injetiva. Também multirelações são usadas na prática, pois a execução dos modelos que necessitam de duas relações é muito comum e para sua modelagem utilizam-se relações ternárias.

3.2.1 Constantes

Existem três tipos de constantes em *Alloy* que são ***none***, conjunto vazio, ***univ***, conjunto universo e ***iden***, conjunto identidade. Se por exemplo, têm-se os conjuntos **n = {(N0),(N1),(N2)}** e **a = {(D0),(D1)}**, e se quer os valores das três constantes relacionadas a eles. O resultado seria **none = {}, univ = {(N0),(N1),(N2), (D0),(D1)}, iden = {(N0,N0),(N1,N1),(N2,N2),(D0,D0),(D1,D1)}.**

3.2.2 Quantificadores

Um quantificador é da forma **Q x:e|F**. A Tabela 3-1 mostra as possíveis formas de quantificação em *Alloy*.

Tabela 3-1 Quantificadores

all x:e|F	F é válido para todo x em e
some x:e|F	F é válido para algum x em e
no x:e|F	F não é válido para x em e
lone x:e|F	F é válido para no máximo um x em e
one x:e|F	F é válido para exatamente um x em e

3.2.3 Operadores

Nesta seção estão descritos os principais operadores presentes na linguagem *Alloy*.

Operadores lógicos

Os operadores lógicos que fazem parte de *Alloy* são utilizados na construção de expressões booleanas assim como na lógica de programação. Seus símbolos e significados estão presentes na Tabela 3-2.

Tabela 3-2 Operadores lógicos

not	!	negação
and	&&	conjunção
or	||	disjunção
implies	=>	implicação
else	,	alternativo
if	<=>	bi-implicação

Operadores de conjuntos

Os operadores de conjuntos utilizados em *Alloy* possuem mesma semântica dos utilizados pela matemática e seus símbolos estão representados na Tabela 3-3.

Tabela 3-3 Operadores de conjuntos

+	união
&	intersecção
-	diferença
in	subconjunto
=	igual

Operadores relacionais

Os operadores relacionais são os responsáveis por relacionar os elementos de *Alloy*. Uma lista desses operadores está presente na Tabela 3-4. Quando se quer relacionar dois elementos **p** e **q**, sendo ambos conjuntos, a expressão **p->q** constitui uma relação binária. No caso de nessa mesma expressão, um dos elementos ter tamanho maior que dois, então essa expressão será uma multirelação. Ainda para a mesma expressão, se ambos os elementos forem tuplas então a relação entre eles será também uma tupla e se eles forem escalares, **p->q** será um par.

A operação **p.q** de relações **p** e **q** é a relação onde se tem todas as combinações de uma tupla em **p** e uma tupla em **q** e se pega sua junção, se ela existe. Se por exemplo têm-se as relações {(N0,A0)} e {(A0,DO)}, sua junção com o operador "**.**" traria como resultado {(N0,D0)}.

O operador "[]" também usado para junção é semanticamente equivalente ao ".", mas os argumentos são dispostos em ordem diferente, e possuem diferentes precedências. A expressão $\mathbf{e_1[e_2]}$ tem mesmo significado de $\mathbf{e_2.e_1}$.

O operador "**~**" de uma relação binária reflete a imagem da relação. Uma relação binária **r** é simétrica se contém a tupla **a->b** e **b->a**, portanto o fechamento simétrico de **r** é **r** + **~r**. Se por exemplo tem-se uma relação **r** = **{(N0,D0),(N1,D1),(N2,D2)}** sua transposta será **~r** = **{(D0,N0),(D1,N1),(D2,N2)}.**

O fechamento transitivo "**^**" de uma relação **r**, é a relação que contém **r** e é transitiva. Intuitivamente, o fechamento transitivo de **r** é o que ocorre quando se mantém navegação através de **r** até que não se possa ir mais adiante. Já o fechamento transitivo reflexivo é o mesmo do fechamento transitivo, com o acréscimo das relações reflexivas, ou seja ***r = ^r + iden.**

O operador de restrição é utilizado para filtrar um domínio ou imagem. A expressão **s<:r** formada por um conjunto **s** e uma relação **r**, contém as tuplas de **r** que começam com o elemento em **s**. Similarmente, **r:>s** contém as tuplas de **r** que terminam com **s**.

Por fim, a operação **p++q** funciona como uma união mais sem as tuplas que contém o domínio de **q** como primeiro elemento, ou seja, **p++q = p – (domain(q)<:p) +q.**

Tabela 3-4 Operadores relacionais

->	produto
.	junção
[]	junção
~	transposta
^	fechamento transitivo
*	fechamento transitivo reflexivo
<:	restrição de domínio
:>	restrição de imagem
++	*override*

3.2.4 Expressão *let*

Uma expressão *let* é definida por **let x = e | A.** Isso significa que uma variável **x** está sendo definida com o valor da expressão **e,** e será utilizada dentro do contexto de **A**

3.2.5 Compreensões

Compreensões fazem relações de propriedades. A expressão de compreensão **{x_1:e_1, x_2:e_2, x_3:e_3, ... , x_n:e_n|F}** faz uma relação com todas as tuplas do tipo **x_1->x_2->x_3-> ... -> x_n** com o construtor **F** ser possível.

Uma declaração introduz um nome a relação. Um construtor da forma **nome-da-relação:expressão** é uma declaração. Uma mesma relação pode ser declarada de diferentes formas, dependendo de quanto de informação precisa se colocar na declaração.

3.2.6 Multirelações

Supondo a declaração **r**: A**m** -> **n**B onde **m** e **n** são palavras-chave de multiplicidade e A e B são conjuntos. Então a relação **r** é construída para mapear cada membro de A para **n** membros de B, e para mapear **m** membros de A para cada membro de B.

3.2.7 Construtor de cardinalidade

O operador "#" aplicado a uma relação, pega a quantidade de tuplas que ela possui. O valor resultante do uso desse operador é um número inteiro. A Tabela 3-5 traz os possíveis operadores que podem ser usados para combinar e comparar inteiros.

Tabela 3-5 Operadores de inteiros

+	mais
-	menos
=	igual
<	menor que
>	maior que
=<	menor igual
>=	maior igual

Capítulo 4

UML/*Alloy*

Com a finalidade de realizar a análise automática dos diagramas de classes UML abstratos foi desenvolvida uma relação entre subconjuntos destes diagramas, com suas restrições em OCL, e *Alloy*. Para isso, foi criado um modelo *EXtensible Markup Language* (XML) [14] representativo desses diagramas, com as anotações em OCL e com algumas instruções de *Alloy*.

Também foi criado um compilador, responsável por fazer a tradução de XML para código Java e de código Java para um código *Alloy* resultante. Sendo esse código *Alloy* resultante o utilizado para realizar as análises. A transformação para uma estrutura Java tem a van*ta*gem de deixar o código mais modular e assim, poder ser transformado não apenar para código *Alloy*, mas também para outras linguagens.

4.1 Mapeamento

Para poder construir o compilador entre os diagramas de classes UML com restrições em OCL e a linguagem *Alloy* é necessário definir regras de tradução entre elas para assim poder realizar o mapeamento. Esse mapeamento se faz necessário para definir como cada elemento do UML será representado no *Alloy*.

Algumas construções foram deixadas de fora do mapeamento por não terem correspondentes em *Alloy* ou por não possuírem grande importância em nível de análise. É o caso, por exemplo, do relacionamento de dependência que não tem correspondente em *Alloy*.

As classes dos diagramas de classes serão mapeadas em assinaturas *Alloy*, já os atributos presentes nela serão campos da assinatura. Se a classe for abstrata, a assinatura também será, e caso algum dos relacionamentos seja do tipo generalização, a assinatura possuirá a palavra **"extends"** após o nome.

Relacionamentos do tipo associação binária, são mapeados como relações diretas de *Alloy*. A multiplicidade dessas relações pode ser mapeada por palavras reservadas de *Alloy* como **lone**, **one** e **Set** ou, em caso de multiplicidades que não possuam palavras reservadas correspondentes, como relações diretas e o valor delimitado em um fato.

Como as expressões de OCL são definidas dentro de um contexto que define uma quantificação universal, as mesmas estarão presentes no fato anexado da assinatura, que é um fato que aparece acoplado a assinatura. De acordo com a função do OCL, uma regra foi criada para corresponder no *Alloy*. E no caso das funções **asSet**, **allInstances** e **self**, nada se altera na

estrutura do *Alloy*. A Tabela 4-1 mostra o mapeamento de alguns dos elementos principais das linguagens em questão.

Tabela 4-1 Mapeamento UML/OCL para *Alloy*

UML/OCL	Alloy
classe	assinatura
associação binária	relação direta
multiplicidade	cardinalidade
generalização	extends
invariante OCL	fato anexado
implies	=>
<>	!=
X->includes(b)	b in X
X->isEmpty()	no X
X->size()	#X
X.allInstances	X
X->select(exp_logic)	b: X| exp_logic

A Figura 4-1 ilustra um exemplo simples de diagrama de classes composto pelas classes **Pessoa**, **Mulher** e **Homem**. A classe **Pessoa** é abstrata e mais geral e as classes **Homem** e **Mulher** são mais específicas. **Pessoa** tem **"idade"** como atributo e restrições OCL para informar que idade não pode ser nula. **Homem** e **Mulher** possuem uma associação binária com multiplicidade **"0..1"** em ambos os lados e papéis "**marido**"e "**esposa**", respectivamente.

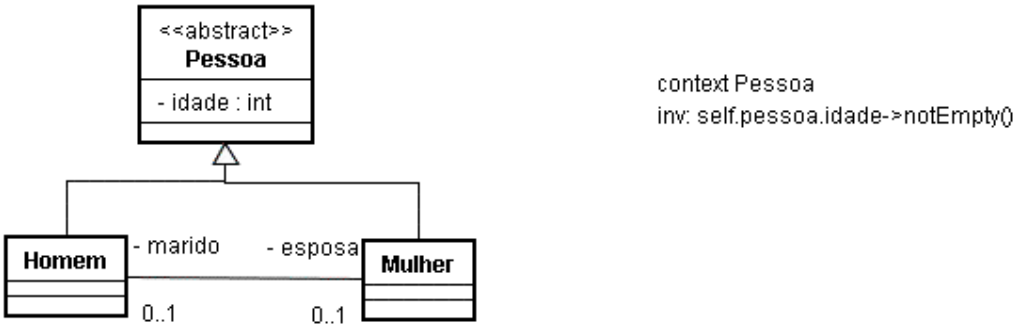

Figura 4-1 Exemplo de Diagrama de Classes

O correspondente código *Alloy* do diagrama de classes da Figura 4-1 é mostrado a seguir:

```
abstract sig Pessoa{
     idade: Int
}{
     some idade
 }

sig Mulher extends Pessoa{
     marido: lone Homem
}
sig Homem extends Pessoa{
     esposa: lone Mulher
}
```

4.2 UML

Nessa seção será descrito o modelo XML criado para representar um diagrama de classes UML, com suas restrições em OCL e algumas instruções de *Alloy* necessárias para a análise automática deste modelo, através da criação da Definição de Tipos de Documentos (DTD) [15]. O DTD define a estrutura de um documento, onde são especificados quais os elementos e atributos são permitidos no mesmo.

Inicialmente tem-se a *tag* <uml> que é a *tag* raiz do XML. Dentro de sua estrutura podem existir as *tags* <diagram>, <ocl>,<assert>,<pred>,<fun> e <command>. A freqüência com que essas *tags* ocorrem é definida de acordo com a seguinte declaração do DTD:

```
<!ELEMENT uml (diagram?, ocl?,(assert|pred|fun|command)*)>
```

Como a *tag* <diagram> e a *tag* <ocl> possuem o sinal **"?"** após seu nome, elas só poderão ocorrer 0 ou 1 vez. Já as *tags* <assert>, <pred>, <fun> e <command> que possuem o sinal **"*"**, poderão ocorrer 0 ou mais vezes. A ordem que essas *tags* devem aparecer é definida pelas **","** e **"|"** utilizadas entre elas. Nesse caso seria <diagram>, seguida por <ocl>, seguida por <assert> ou <pred> ou <fun> ou <command>.

Dentro da *tag* <diagram> só poderá existir a *tag* <class>, com freqüência **"*"**.

```
<!ELEMENT diagram (class*)>
```

Uma classe pode possuir 0 ou mais atributos e relacionamentos e possui um único nome, obrigatório. Assim, a *tag* <class> terá *tags* <attribute> e <relation> internas a ela, e terá *name* como atributo obrigatório. Além deste, essa *tag* possuirá também um outro atributo obrigatório *abstract,* para diferenciar se a classe é abstrata ou não.

```
<!ELEMENT class (attribute*, relation*)>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class abstract CDATA #REQUIRED>
```

Os atributos de uma classe possuem um nome e um tipo que no XML serão representados por atributos obrigatórios da *tag* <attribute>. Essa *tag* não possui *tags* internas assim como a *tag* <relation> que terá como atributos obrigatórios o tipo e a classe da relação, e como atributos opcionais a multiplicidade e o nome da relação.

A *tag* <ocl> é responsável por guardar as informações relacionadas aos invariantes OCL. Dentro desta *tag*, estão presentes 0 ou mais *tags* <context>.

```
<!ELEMENT ocl (context*)>
```

O contexto por sua vez possui internamente 0 ou 1 *tag* <invariant>, que representa o invariante OCL. A *tag* <context> contém como atributos obrigatórios um *id* e um *name* que guardam respectivamente um identificador para diferenciar dos demais contextos e o nome da classe a que o invariante está relacionado.

Um invariante pode opcionalmente possuir um nome como atributo. Internamente ele pode conter as *tags* <exp>, <self> e as *tags* correspondentes as funções do OCL que podem aparecer 0 ou mais vezes em qualquer ordem, dependendo do invariante que se está mapeando.

```
<!ELEMENT invariant (self| exp| asSet| includes| excludes|
select| reject| implies| union| intersection| collect|
forAll| isUnique| exists| one| let| in| if| then| else|
allInstances| size| isEmpty| notEmpty| and| or| xor| less|
dif| not)*>
<!ATTLIST invariant name CDATA #IMPLIED>
```

A *tag* <self> é utilizada quando uma ocorrência de *self* aparece no invariante e a *tag* <exp> para representar algo que nem seja uma função e nem *self*. A *tag* <exp> tem como atributo obrigatório *attribute* que guarda o conteúdo de uma expressão.

```
<!ELEMENT exp EMPTY>
<!ATTLIST exp attribute CDATA #REQUIRED>
```

Algumas das funções do OCL não possuem argumentos e portanto, não possuem *tags* internas, como por exemplo a *tag* <asSet> e a *tag* <isEmpty>. Já outras, como a *tag* <select>, podem possuir internamente combinações dessas funções similarmente ao invariante.

As afirmativas, predicados, funções e comandos de *Alloy* estão também inseridos no XML, pois é necessário passar instruções ao analisador para que ele gere as devidas soluções. Trata-se de afirmações que se pretende checar se são verdadeiras e, predicados e funções que buscam soluções para o modelo especificado até este ponto. As afirmativas são representadas pela *tag* <assert> e possuem um nome como atributo interno e a afirmativa em si como conteúdo entre as *tags*.

```
<!ELEMENT assert (#PCDATA)>
<!ATTLIST assert name CDATA #REQUIRED>
```

A *tag* <pred> e a *tag* <fun> são semelhantes por possuírem atributos internos *name* e *input* relativos ao nome e a entrada do predicado ou função. Além disso, possuem em seu conteúdo o predicado ou função em si. O que diferencia essas duas *tags* é que a *tag* <fun> além dos atributos citados anteriormente, possui ainda um atributo *output* relativo à saída da função.

Por fim, a *tag* <command> possui o atributo *type* que diferencia se o comando é *run* ou *check* e o atributo *name* que representa o nome do comando. Internamente as suas *tags*, ele pode receber as instruções do comando, caso elas existam.

A Figura 4-2 ilustra um modelo UML/OCL de um sistema de arquivos composto por objetos, que podem ser arquivos ou diretórios, uma única raiz, subtipo de diretório, e onde todos os diretórios possuem um conjunto de objetos e, com exceção da raiz, um diretório pai.

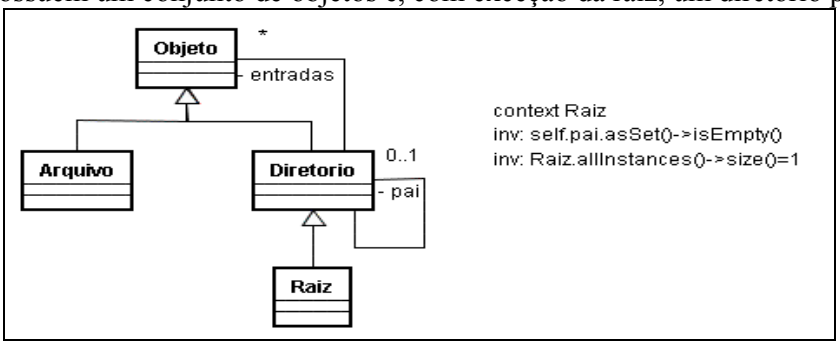

Figura 4-2 Exemplo de Diagrama de Classes

A representação do mesmo diagrama da Figura 4-2 no formato representativo XML acrescida de uma afirmativa e um comando é representada a seguir:

```
<uml>
     <diagram>
          <class name="Objeto" abstract="yes"></class>
          <class name="Arquivo" abstract="no">
               <relation type="generalization" class="Objeto"/>
          </class>
          <class name="Diretorio" abstract="no">
               <relation type="generalization" class="Objeto"/>
               <relation name="entradas" type="association"
                    class="Objeto" multiplicity="*"/>
               <relation name="pai" type="association"
                    class="Diretorio" multiplicity="0..1"/>
          </class>
          <class name="Raiz" abstract="no">
               <relation type="generalization"
                    class="Diretorio"/>
          </class>
     </diagram>
     <ocl>
          <context id="c1" name="Raiz">
               <invariant>
                    <exp attribute="Raiz"/>
                    <allInstances/>
                    <size/>
                    <exp attribute="=1"/>
               </invariant>
          </context>
          <context id="c2" name="Raiz">
               <invariant>
                    <self/>
                    <exp attribute="pai"/>
                    <asSet/>
                    <isEmpty/>
               </invariant>
          </context>
     </ocl>
     <assert name="Test">all b: Raiz| b =Diretorio</assert>
     <command type="check" name="Test">for 4</command>
</uml>
```

4.3 Compilador

Buscando estabelecer uma relação entre os elementos das linguagens UML e *Alloy*, foi desenvolvido um compilador entre elas na linguagem Java. Como mostra a Figura 4-3, esse compilador tem como entrada o XML explicado na seção anterior, e a partir dele gera uma estrutura Java correspondente e, por fim, monta um arquivo com o código *Alloy* resultante.

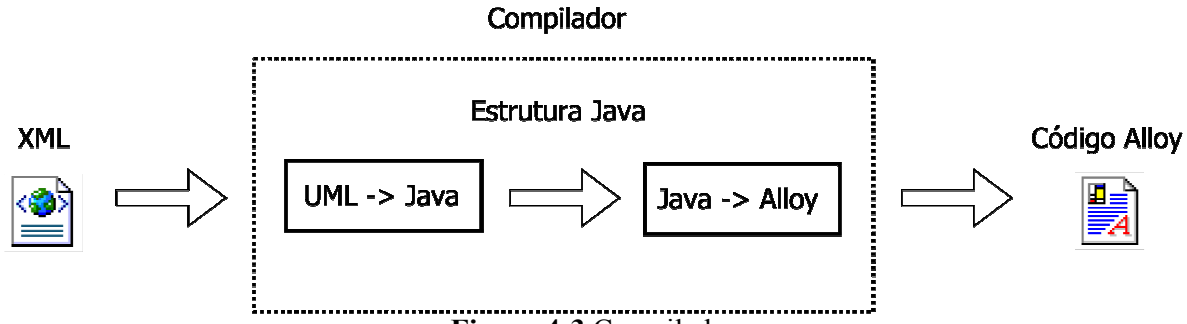

Figura 4-3 Compilador

Dentro do projeto Java foram criados dois pacotes denominados ***ast*** e ***compiler***. Para cada *tag* do XML uma classe foi criada no pacote *ast*, referente a Árvore Sintática Abstrata (AST) [8] e de acordo com seus atributos e campos no XML foram criadas variáveis correspondentes nas classes Java. Já no pacote *compiler* estão presentes as classes responsáveis por ler o XML e transformar em uma correspondente estrutura Java e por pegar esta estrutura e transformá-la em código *Alloy*.

4.3.1 Pacote ast

As classes do pacote *ast* foram criadas com o intuito de armazenar o conteúdo do XML em Java e por isso, cada *tag* do XML possui uma classe correspondente a ela como é mostrado no exemplo do diagrama de classes da Figura 4-4. Ela ilustra um exemplo de relacionamento entre algumas classes presentes do pacote *ast*. Cada classe possui os métodos *get* e *set* para retornar e informar o valor dos atributos respectivamente.

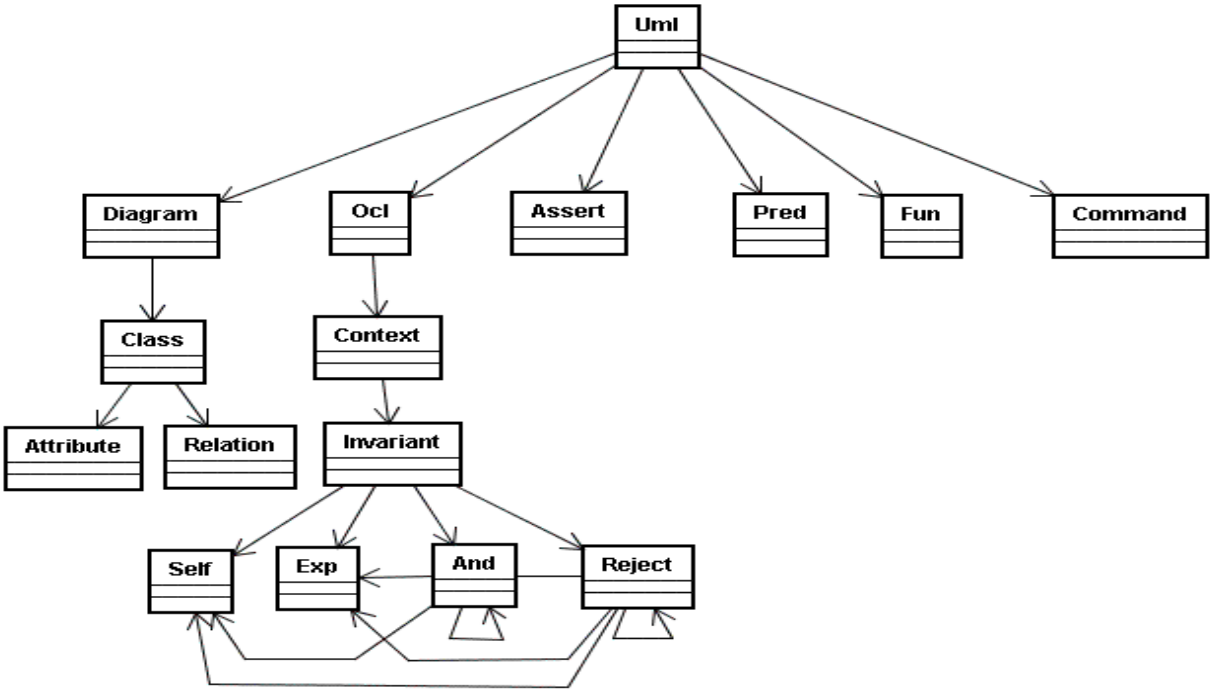

Figura 4-4 Exemplo de Diagrama de Classes do pacote *ast*

A classe correspondente a *tag* <uml> é a **Uml.java** e seu construtor é formado por um **`Diagram`**, um **`Ocl`**, um vetor de afirmações, um vetor de predicados e um vetor de comandos.

```
public Uml(Diagram diagram, Ocl ocl, vetor assertions,
     vetor preds, vetor functions, vetor commands){
          this.diagram = diagram;
          this.ocl = ocl;
          this.assertions = assertions;
          this.preds = preds;
          this.functions = functions;
          this.commands = commands;
}
```

A seguir estão descritas as classes correspondentes as *tags* que podem estar presentes dentro da *tag* <uml>, bem como as internas a elas.

Diagram

No arquivo XML, as possíveis *tags* presentes dentro de <diagram> são as *tags* <class>, representando as classes do diagrama. Sendo assim, a classe **Diagram.java** possui um vetor responsável por armazenar os objetos da classe **Class.java**.

```
public Diagram(vetor classes){...}
```

A classe **Class.java** possui no seu construtor duas Strings que recebem o valor vindo dos atributos *name* e *abstract* do XML com o nome da mesma e o valor do *abstract* respectivamente.

```
public Class(String name, String abs){...}
```

Essa classe possui ainda dois vetores para guardar os atributos e as relações, caso eles existam, e os métodos **`addAttribute`** e **`addRelation`** para adicioná-los no vetor.

```
public void addAttribute(Attribute attribute){
     this.attributes.addElement(attribute);
}

public void addRelation(Relation relation){
     this.relations.addElement(relation);
}
```

A classe **Attribute.java** tem como parâmetros para seu construtor as Strings *name* e *type*, que recebem os valores dos atributos da *tag* <attribute>. Essa classe não possui métodos além dos *gets* e *sets* das variáveis.

```
public Attribute(String name,String type){...}
```

A classe **Relation.java** por sua vez possui três construtores que podem ser de três tipos diferentes, de acordo com os atributos obrigatórios e opcionais. O primeiro deles possui como

parâmetros as Strings *type* e *classe*, o segundo as Strings *name*, *type* e *classe* e o terceiro as Strings *name*, *type*, *multiplicity* e *classe*.

```
public Relation(String name, String type, String
    multiplicity, String classe) {...}
```

Ocl

No caso da *tag* <ocl>, as *tags* que aparecem no seu interior são *tags* <context>. Por poder possuir mais de um contexto, a classe **Ocl.java** possui um vetor de contextos que irá guardar os objetos da classe **Context.java**. O construtor da classe **`Ocl`** recebe apenas esse vetor de contextos e não possui outros métodos, além dos *get* e *set* do vetor.

```
public Ocl(vetor context) {...}
```

Por sua vez, a *tag* <context> possui os atributos *id* e *name* e tem no seu interior a *tag* <invariant>. Assim, a classe correspondente **Context.java** tem em seu construtor uma String com o *id*, uma String com o *name* e um objeto da classe **Invariant.java**.

```
public Context(String id, String name, Invariant inv) {...}
```

A *tag* <invariant> é responsável por guardar o invariante OCL e nela pode ter diversos tipos diferentes de funções do OCL, cada uma representada por uma *tag*. Além disso, cada função pode ter contida nela outros tipos de funções. Sendo assim, o construtor da classe **Invariant.java** possui uma String com o nome do invariante, caso ele possua, e um vetor que guarda os objetos relacionados com as *tags* das funções ou expressões vindas do XML.

```
public Invariant(String name, vetor constraint) {...}
```

Se tem-se o seguinte XML:

```
<context id="c2" name="Raiz">
     <invariant>
          <exp attribute="pai"/>
          <asSet/>
          <isEmpty/>
     </invariant>
</context>
```

O vetor formado por esse invariante terá um objeto da classe **Exp.java**, seguido por um objeto da classe **AsSet.java** e por fim um objeto da classe **IsEmpty.java.** Se as *tags* de funções OCL não possuem *tags* internas, as classes possuem construtor vazio, caso contrário, elas irão possuir um vetor semelhante ao utilizado pela classe **Invariant.java.**

Assert, Pred, Fun e Command

A *tag* <assert> tem como atributo *name* e uma afirmativa como conteúdo. Dessa forma, o construtor da classe **Assert.java** é formado pelas Strings *name* e *ass* que terão os valores recebidos do XML.

```
public Assert(String name, String ass) {...}
```

As *tags* <pred> e <fun> possuem como atributos em comum *name* e *input*. No caso de <pred> além desses atributos ela possui ainda um predicado como o seu conteúdo, já a <fun> possui o atributo *output* e uma função como o seu conteúdo. Os construtores das classes **Pred.java** e **Fun.java** são formados por Strings que representam cada um desses atributos e os seus conteúdos.

```
public Pred(String name, String input, String pred) {...}

public Fun(String name, String input, String output, String
     fun){...}
```

Por fim, a *tag* <command> tem como atributos *type* e *name* e com isso, o construtor da classe **Command.java** é formado por Strings desses atributos e pelo conteúdo da mesma, caso exista instruções para os comandos como conteúdo.

```
public Command(String type, String name, String cmd){...}
```

4.3.2 Pacote compiler

O pacote *compiler* possui duas classes com a finalidade de transformar do XML para Java e de Java para código *Alloy*. Na classe **UmlJava.java**, através do uso do *Document Object Model* (DOM) [16], é feita a leitura do XML e o armazenamento no formato de árvore de nodos. À partir dessa árvore, os métodos **`readDiagram`**, **`readOcl`**, **`readAssert`**, **`readPred`**, **`readFun`** e **`readCommand`** percorrerão a árvore em busca dos elementos correspondentes.

Com o uso do DOM é possível montar uma lista dos nodos da árvore e assim percorrer a estrutura interna da mesma. No método **`readDigram`**, por exemplo, pretende-se buscar as possíveis *tags* <diagram> e para isso, é criada uma lista de nodos através da busca por *tags* com o nome **"diagram"** em **`elem`**, que é um elemento formado pelo documento XML.

```
NodeList d = elem.getElementsBy<i>Tag</i>Name("diagram");
```

No XML de entrada desse compilador é possível a ocorrência de no máximo um diagrama. Assim, é criado um **`Element`** que pegará a primeira ocorrência da **`NodeList`** e a partir dela, criará uma nova **`NodeList`** formada pelas classes do diagrama.

```
Element <i>tag</i>Diagram =(Element)d.item(0);
NodeList nl = <i>tag</i>Diagram.getElementsBy<i>Tag</i>Name( "class" );
```

Tendo essa **`NodeList`**, um vetor de classes é criado para armazenar os dados presentes nas *tags* <class> e após percorrer toda **`NodeList`** o vetor será adicionado no objeto **`Diagram`**. Para cada <class> do XML um objeto **`Class`** será criado com seus atributos armazenados em Strings e as *tags* <attribute> e <relation> em vetores.

```
String nameClass = <i>tag</i>Class.getAttribute( "name" );
String abs = <i>tag</i>Class.getAttribute("abstract");
Class classe = new Class(nameClass,abs);
```

```
NodeList n2 = tagClass.getElementsByTagName("attribute");
NodeList n3 = tagClass.getElementsByTagName("relation");
```

De maneira semelhante a como se adquire os atributos para <class>, ocorrerá para os atributos de <attribute> e <relation>, e a cada novo atributo ou relacionamento um objeto **Attibute** ou **Relation** é criado e armazenado no vetor correspondente.

```
classe.addAttribute(new Attribute(nameAttr,type));
classe.addRelation(new Relation(name, nameRel, multiplicity,
     associationClass));
```

Para o método **readOcl** as NodeLists para as *tags* <ocl>, <context> e <invariant> são criadas de maneira semelhante a da *tag* <diagram>. No entanto, a partir da *tag* <invariant>, não se sabe ao certo quais *tags* estarão internas a ela, nem em que ordem estarão dispostas. Portanto, a **NodeList** formada pelos elementos de <invariant> conterá todos os elementos presentes internamente as suas *tags*, independente de seus tipos e um vetor é criado para guardar os objetos que serão criados de acordo com essas *tags*.

```
NodeList list=tagInvariant.getElementsByTagName("*");
vetor exps = new vetor();
```

No entanto, essa forma de adquirir os elementos irá pegar todas as *tags* internas a *tag* <invariant>, independente de serem *tags* filhas ou internas a estas. Assim, ao percorrer essa **NodeList** é necessário se certificar que o elemento que está se trabalhando tem como pai a *tag* <invariant> para poder montar a estrutura apenas com elementos filhos.

```
Element element = (Element)list.item(j);
          if(element.getParentNode()==tagInvariant )
```

A partir deste ponto, existem três possíveis tipos de *tags* internas: as *tags* que não possuem argumentos, as *tags* de funções que possuem conteúdo interno, e a *tag* <exp> que guarda as expressões que não são funções e nem *self*. Assim, para cada *tag* filha de <invariant> será analisado, através do método **verifyTypes**, se trata-se de uma *tag* <exp> ou de uma *tag* que não possui argumentos. Caso seja uma *tag* <exp> o método **isExp** irá criar um objeto do tipo **Exp** e caso seja uma *tag* que não possui argumentos, o método **isType** chamará o método responsável por criar o objeto do tipo especificado, de acordo com a String com o nome da *tag*.

```
obj = verifyTypes(element,exps);
```

Se **obj** não for nulo, o objeto retornado por **verifyTypes** será inserido no vetor **exps**. Em caso contrário, o objeto a ser criado é do tipo função com conteúdo interno e com isso, o método **isOthers** será invocado a fim de descobrir a qual função do OCL a *tag* em questão corresponde.

```
obj = verifyTypes(element,exps);
    if(obj!=null){
        exps.addElement(obj);
    }else
```

```
exps.addElement(isOthers(element));
```

O método **isOthers** irá proceder da mesma forma como se a *tag* fosse <invariant>, pois também não se sabe ao certo que conteúdo estará presente. Assim, como internamente as *tags* de funções pode haver os mesmos tipos de *tags* internas a <invariant>, o mesmo procedimento será adotado e **isOthers** será invocado recursivamente. Ao fim desse método, o método **isTypeFull** será invocado passando o nome da *tag* e o vetor montado com as *tags* internas a ela e de acordo com o nome da *tag* o método correspondente a ela criará o objeto a ser retornado.

```
ob = isTypeFull(name,elements);
return ob;
```

Ao fim da leitura de cada *tag* <invariant> um objeto da classe **Invariant.java** será criado e irá armazenar o vetor com os objetos que compõem esse invariante. Por sua vez, esse objeto irá ser armazenado em um objeto **Context** e guardado no vetor de contextos. Terminada a leitura de todos os contextos, um objeto **Ocl** será criado e irá guardar o vetor de contextos.

Além da leitura do **Diagram** e do **Ocl**, ainda é possível a dos comandos, afirmativas, predicados e funções. A leitura das *tags* correspondentes a eles é feita semelhantemente a das classes. Para cada atributo ou conteúdo interno as *tags*, existe uma String correspondente em Java que irá armazenar os valores adquiridos do XML. Ao fim da leitura de cada *tag* o objeto correspondente a ela será guardado no vetor que o representa, ou seja, se, por exemplo, se tem um comando, um objeto da classe **Command.java** será criado e armazenado no vetor **commands**.

```
Command cmd = new Command(type,name,command);
commands.addElement(cmd);
```

Para criar um objeto da classe **Uml.java**, é necessário a leitura de todos os componentes do XML. O construtor dessa classe é formado por um **Diagram**, um **Ocl**, um vetor de afirmativas, um vetor de predicados, um vetor de funções e um vetor de comandos. Assim, o método **compiler** cria um objeto **Uml** formado pelos leitores dos componentes do XML.

```
public Uml compiler() throws Exception{
     uml =new Uml(readDiagram() ,readOcl(), readAssert(),
          readPred(), readFun(), readCommand());
     return uml;
}
```

A classe **JavaAlloy.java** é a responsável por receber um objeto **Uml** e construir o arquivo com o código *Alloy* resultante. Esse mapeamento entre as linguagens UML/OCL e *Alloy* foi desenvolvido de acordo com as regras de tradução definidas entre elas. Inicialmente é criado o arquivo **"test.als"** onde será escrito o código. O método **readUml** tem como argumento um vetor de classes e percorre esse vetor montando o correspondente *Alloy*.

Para cada classe, além de percorrer as informações de se é abstrata, o nome da mesma e os atributos e relacionamentos, também será percorrido o vetor de contextos em busca de algum que tenha mesmo nome que o nome da classe. Caso isso ocorra, esse contexto será traduzido de acordo com as regras para *Alloy* e será guardado em um fato anexado.

O método **test** é o responsável pelas regras de tradução das funções do OCL, bem como de **Exp** e **Self**. Ele tem como parâmetros uma String que corresponde ao código resultante do

invariante OCL traduzido para *Alloy* até o momento, e o objeto a que se está buscando a regra a ser aplicada. Se as entradas para **`test`** forem, por exemplo, a String "pai" e o objeto **`IsEmpty`**, o código resultante seria:

```
no pai
```

4.4 Resultante *Alloy*

O código resultante do compilador é montado na classe **JavaAlloy.java** e para se chegar a ele é necessário percorrer toda a estrutura do **`Uml`** e ir escrevendo no arquivo de acordo com as regras de tradução. No método **`readUml`** inicialmente define-se o módulo que se chama **test**.

```
out.write("module test");
```

Uma classe do diagrama de classes é mapeada como uma assinatura do *Alloy*. Para cada classe do vetor de classes que entra como parâmetro do método **`readUml`**, a primeira coisa a ser observada é se ela é abstrata para assim poder definir a assinatura.

```
if(abs.equals("yes")){
     out.write("abstract sig "+ name);
     }
else
     out.write("sig "+ name );
```

Em seguida é necessário observar se a classe tem algum relacionamento do tipo **"generalization"** e se é a primeira vez que essa classe tem um relacionamento desse tipo, pois caso isso seja verdadeiro a palavra **"extends"** aparecerá ainda na definição da assinatura, seguida pela chave que indica o início do corpo da mesma.

```
if(rel.getType().equals("generalization")&&f==0){
     f++;
     out.write(" extends "+rel.getClasse());
     out.write("{");
}
```

Caso não exista nenhuma relação de generalização, é aberta a chave simbolizando o inicio do corpo da assinatura.

```
if(f==0){
     out.write("{");
}
```

O corpo da assinatura é formado pelas relações de associações e pelos atributos das classes vindos do XML. No caso das associações, escreve-se o nome da mesma e caso a multiplicidade possa ser mapeada como uma palavra reservada do *Alloy*, como é o caso de **"0..1", "*", "1"**, escreve-se uma definição de relação utilizando essa palavra.

```
if(rel.getType().equals("association")){
```

```
        out.write(rel.getName());
        if(rel.getMultiplicity().equals("0..1")){
             out.write(":lone "+rel.getClasse());
}
```

No caso de não haver palavra reservada que represente a quantidade desejada, escreve-se apenas a relação entre as classes e a definição da quantidade possível é definida em um fato anexado.

```
else
        out.write(":"+rel.getClasse());
```

Os atributos são escritos no arquivo simplesmente pegando o seu nome seguido por **":"** e o tipo do mesmo, de acordo com o tipo que vem do XML.

```
if(type.equals("int"))
        out.write(attr.getName()+":Int");
```

Terminada a leitura dos atributos o corpo da assinatura é fechado e é iniciado o fato anexado. Para cada assinatura um fato anexado será criado com a finalidade de definir as multiplicidades que não possuem palavra reservada e guardar o código *Alloy* correspondente aos invariantes do OCL.

As multiplicidades serão definidas de acordo com os valores e com o que os separam, se **","** ou **".."**. Foram criados dois **`StringTokenizer`** com a finalidade de diferenciar esses separadores e de acordo com eles é criado um novo campo no fato anexado.

```
str = new StringTokenizer(rel.getMultiplicity(),"..");
str1= new StringTokenizer(rel.getMultiplicity(),",");
```

No caso dos valores serem separados por **".."** significa que a multiplicidade está entre esses dois valores e então se escrevem duas regras para delimitar o valor.

```
if(str.countTokens()==2){
        out.write("#"+rel.getClasse()+">"+str.nextToken());
        out.write("#"+rel.getClasse()+"<"+str.nextToken());
}
```

Já no caso de serem separados por **","** existirá uma relação **"||"** entre os possíveis valores, se esse valor for diferente de 1.

```
else if(str1.countTokens()!=0){
        for(int e=0;e<str1.countTokens()-1;e++){
             out.write("#"+rel.getClasse()+"="
                  +str1.nextToken() +"||");
        }
}
        out.write("#"+rel.getClasse()+"="+ str1.nextToken());
}
```

Terminadas as multiplicidades os seguintes campos a serem escritos são os do mapeamento dos invariantes, de acordo com as regras de tradução. Para cada contexto é criada uma String que guardará o valor do invariante até o presente momento e será percorrido o vetor do invariante enviando para o método **`test`** a String e o objeto referente ao elemento do vetor.

```
String str="";
for(int j=0;j<context.getInv().getConstraint().size(); j++){
Object obj = context.getInv().getConstraint().get(j);
str=test(str,obj);
}
```

Para cada objeto, uma regra de tradução diferente é aplicada, de acordo com o mapeamento entre as duas linguagens. Ao terminar de percorrer o vetor, o corpo do fato anexado é fechado e serão escritos as afirmativas, os predicados, as funções e os comandos a partir apenas dos dados guardados na estrutura **`Uml`** e montando de acordo com as construções da linguagem.

Capítulo 5

Análise Automática

A fim de se obter a análise automática do código *Alloy* gerado pelo compilador, um estudo da API do *Alloy Analyzer* foi necessário para assim poder integrá-la ao projeto em questão. A integração da API com o compilador é feita utilizando o arquivo ".als" gerado pelo compilador.

A Figura 5-1 mostra a integração do analisador com o compilador explicado na seção anterior. O código *Alloy* resultante do compilador é utilizado como entrada para o analisador que, ao fim das análises, gera um XML para cada comando satisfatório.

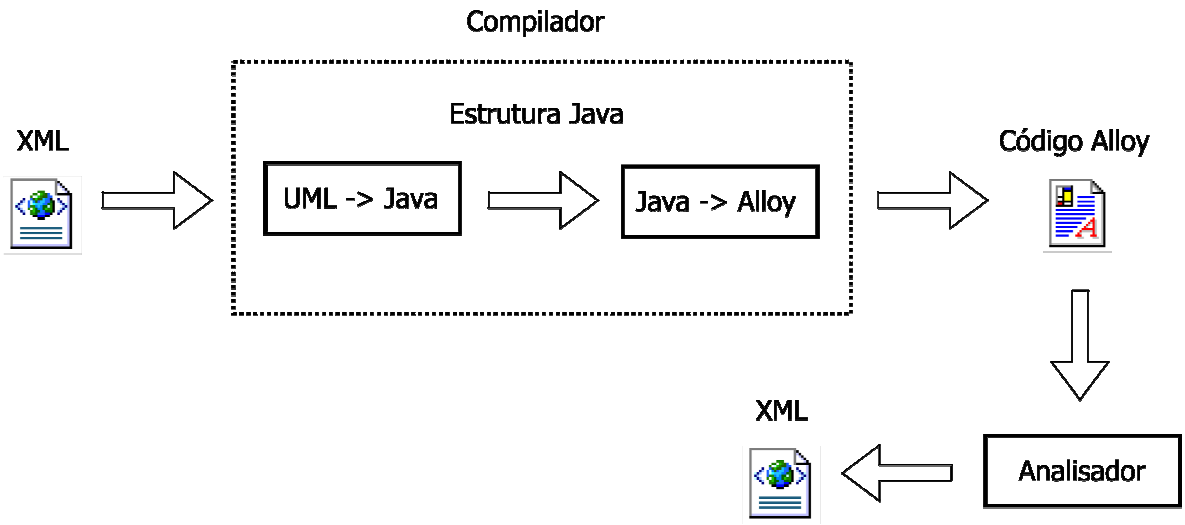

Figura 5-1 Integração do Compilador com o Analisador

5.1 API do *Alloy Analyzer*

A nova versão do *Alloy*, a versão 4, foi reescrita e trouxe novidades que possibilitam a integração e expansão dessa linguagem. A API do AA agora faz parte do *alloy4.jar*, é composta por componentes individuais com mínima dependência e seu acesso é público.

Como todas as partes da API podem ser acessadas independentemente, a maneira mais simples de se trabalhar com os pacotes é incluindo o *alloy4.jar* diretamente no projeto e importar as classes necessárias ao mesmo. Seis pacotes fazem parte da API do AA:

1) alloy4 – contém as estruturas de dados fundamentais e classes de ajuda
2) alloy4compiler.ast – contém a definição dos nodos da AST
3) alloy4compiler.parser – contém o compilador
4) alloy4compiler.translator – contém o tradutor do *Alloy4* para CNF
5) alloy4viz – lê e mostra os exemplos do *Alloy4*.
6) alloy4whole – contém um cliente simples *Graphical User Interface* (GUI), e alguns exemplos de como utilizar a API.

No pacote *alloy4* é possível criar novas mensagens para eventos ou alterar as que aparecem quando se está à procura de exemplos através da classe **`A4Reporter`**. Caso pretenda-se alterar algo na estrutura da linguagem, o pacote *alloy4compiler.ast* deve ser alterado, pois nele estão as definições das estruturas da linguagem.

A classe **`CompUtil`** de *alloy4compiler.parser* é responsável por pegar uma estrutura de entrada e reconhecer os *tokens* que estão corretos de acordo com a linguagem. Já no pacote *alloy4compiler.translator* existe a classe **`A4Solution`** que representa a solução, que pode ser satisfatória ou não.

O pacote *alloy4viz* recebe as soluções, faz a leitura e monta uma janela de visualização com os diferentes tipos possíveis de resultados, como por exemplo, o formato gráfico, XML ou em pastas. Por fim, no pacote *alloy4whole*, existem alguns exemplos do uso do compilador, da API e também um exemplo na classe **`SimpleGUI`**, do uso geral do AA.

5.2 Integração

Sem a integração do AA com o compilador, não seria possível a análise automática de um diagrama de classes UML utilizando *Alloy*. Assim, um pacote *analyzer* foi criado para fazer a integração entre o compilador e o AA e gerar um XML para cada solução encontrada para os comandos de *Alloy*.

5.2.1 Pacote *analyzer*

O pacote *analyzer* é formado apenas pela classe **Analyzer.java** que foi desenvolvida baseada na classe **ExampleUsingTheCompiler.java**. Essa classe mostra um exemplo de como utilizar algumas funções do compilador do AA. Inicialmente ela mostra como criar e imprimir mensagens de diagnóstico de acordo com os eventos que estão ocorrendo no decorrer da compilação. Em seguida, um exemplo de como criar um visualizador, responsável por mostrar os resultados nos diferentes tipos de visualização, a partir do XML criado como solução.

No próximo passo ele percorre as Strings que contêm o caminho do arquivo ".als" e chama o método da classe **`CompUtil`** que monta a AST do arquivo, checa os tipos e cria o XML para cada solução satisfatória para os comandos encontrados no arquivo. Caso se pretenda observar os resultados no visualizador, é chamado o método **`run`** do mesmo.

No caso da classe **Analyzer.java** o intuito principal é gerar o XML de solução para os comandos encontrados no arquivo. Esse XML é o resultado da análise automática do modelo UML que entra em formato de XML no compilador. Para isso, foi construído o método **`analyzer`** que irá utilizar partes da API do AA e assim, realizará a análise do código *Alloy*. Esse método é o responsável por produzir as soluções resultantes da análise. Dentro dele está,

inicialmente a criação de um objeto **A4Reporter** que irá criar as mensagens de diagnóstico seguido por uma String que contém o nome do arquivo gerado pelo compilador.

```
A4Reporter rep = new A4Reporter() {
     @Override public void warning(ErrorWarning msg) {
          System.out.print("Relevance
               Warning:\n"+(msg.toString().trim())+"\n\n");
          System.out.flush();
        }
};

String filename = "test.als";
```

Esse nome de arquivo será genérico, pois para cada arquivo XML de entrada, o compilador irá gerar o arquivo **"test.als"** que será utilizado pelo analisador. Após o nome, dentro do método **analyzer** virá a criação de um objeto **World** que é responsável por armazenar os elementos do *Alloy* como sig, fun e pred. Esse objeto será criado utilizando o método **parseEverything_fromFile** da classe **CompUtil** que lê o arquivo e monta uma estrutura de árvore, sendo **World** a raiz.

```
World world = CompUtil.parseEverything_fromFile(null, null,
     filename, rep);
```

Após essa criação os tipos serão checados e se tem a possibilidade de criar algumas opções para como se quer executar o comando, nesse caso será escolhido que deve ser em Java, através da opção **A4Options.SatSolver.SAT4J**.

```
A4Options options = new A4Options();
options.setReporter(rep);
options.solver = A4Options.SatSolver.SAT4J;
```

Por fim, a lista de comandos será percorrida e uma **A4Solution** será criada para armazenar o resultado de cada comando. Caso essa solução seja satisfatória, um XML com nome igual ao nome do comando é criado com o resultado da análise do mesmo.

```
A4Solution ans =
     TranslateAlloyToKodkod.execute_command(world, cmd,
     options, null, null);

if (ans.satisfiable()) {
     ans.writeXML(cmd.name+".xml", false);
}
```

Nessa ferramenta os comandos são inseridos juntamente com as estruturas do diagrama de classes no XML de entrada. No entanto, futuramente os comandos deverão ser inseridos pelo usuário a cada análise a ser realizada, tornado esta ferramenta mais interativa.

5.3 Exemplos

Nesta seção estão descritos dois exemplos de análises realizadas com a ferramenta desenvolvida. Sendo o primeiro deles uma demonstração de um contra-exemplo para uma afirmação dada e o segundo, uma instância para um predicado. No resultado *Alloy* gerado além da representação no formato de conjunto dos átomos e relacionamentos, também irá aparecer o conjunto dos possíveis números inteiros utilizados pela linguagem *Alloy*.

5.3.1 Contra-exemplo

No diagrama de classes apresentado na Figura 5-2 a classe **Pessoa** é uma classe abstrata que possui como atributo um inteiro nomeado por **idade**. Relacionadas a ela estão as classes **Aluno** e Professor, através de relacionamentos de generalização. O diagrama possui ainda a classe **Disciplina** com **horasDiarias** como atributo do tipo inteiro e se relacionando através de associação com as classes **Aluno** e **Professor.**

Um invariante OCL foi adicionado ao diagrama. Esse invariante indica apenas que as horas diárias que uma disciplina pode possuir tem que ser maior do que 1.

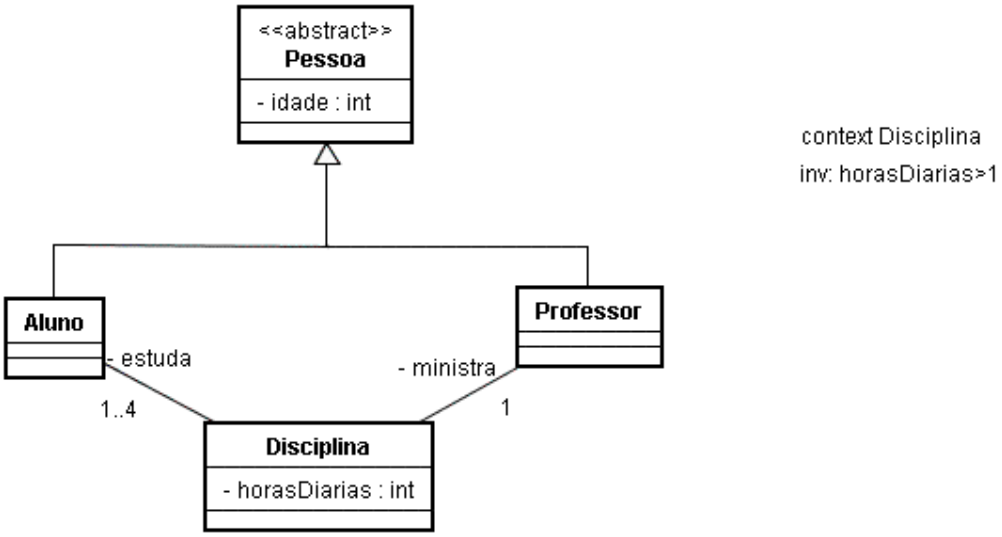

Figura 5-2 Diagrama de Classes para Geração de Contra-Exemplo

Para poder encontrar um contra-exemplo é necessário que uma afirmativa não esteja correta. Para o diagrama em questão é ainda necessária a criação dessa afirmativa que se deseja checar e do comando correspondente a ela. Se afirma-se que existe alguma disciplina que as horas diárias terão que ser necessariamente iguais a 1, um contra exemplo será gerado pois, a única restrição que foi imposta através do invariante OCL, não garante a veracidade dessa afirmação.

```
assert horas{
     some a: Disciplina| a.horasDiarias=1
}

check horas for 3
```

O contra-exemplo gerado pela ferramenta é mostrada no Apêndice A e o resultado em *Alloy* é representado a seguir:

```
integers={-8=-8, -7=-7, -6=-6, -5=-5, -4=-4, -3=-3, -2=-2, -1=-1,
0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7}
Aluno={Aluno[0], Aluno[1]}
Professor={Professor[0]}
Disciplina={Disciplina[0], Disciplina[1], Disciplina[2]}
Pessoa.idade={Professor[0]->-8, Aluno[0]->-7, Aluno[1]->-8}
Disciplina.ministra={Disciplina[0]->Professor[0],
     Disciplina[1]->Professor[0], Disciplina[2]->Professor[0]}
Disciplina.estuda={Disciplina[0]->Aluno[1],
     Disciplina[1]->Aluno[1], Disciplina[2]->Aluno[0]}
Disciplina.horasDiarias={Disciplina[0]->2, Disciplina[1]->2,
     Disciplina[2]->5}
```

Nesse contra-exemplo gerado, as disciplinas **`Disciplina[0]`** e **`Disciplina[1]`** e **`Disciplina[2]`** possuem valores maiores do que 1, como é definido no invariante mas, como nenhum deles possui valor igual a 1 como é afirmado. Dessa forma, a afirmação feita torna-se então incorreta.

5.3.2 Instância

O diagrama da Figura 5-3 ilustra um relacionamento do tipo associação binária entre as classes **Companhia e Empregado**. O **Empregado** possui um identificador inteiro como atributo e só poderá existir uma única **Companhia** com papel denominado **empregador.**

Um invariante OCL adicionou ainda uma restrição sobre o diagrama para certificar que o identificador do **Empregado** não será nulo.

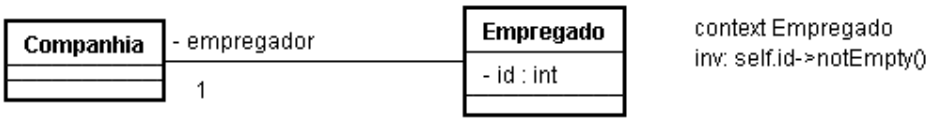

Figura 5-3 Diagrama de Classes para Geração de Instância

O predicado de *Alloy* que se faz necessário para poder existir a geração de uma instância para o diagrama, procura por algum exemplo que mostre que o identificador de **Empregado** é único. O comando **`run`** irá procurar para um conjunto de até no máximo 5 elementos de cada tipo se é possível acontecer de não existirem identificadores repetidos.

```
pred idUnico{
     all a, a':Empregado| a!=a' => a.id!= a'.id
}

run idUnico for 5
```

A instância gerada pela ferramenta é mostrada no Apêndice B e seu resultado em *Alloy* está representada a seguir:

```
integers={-8=-8, -7=-7, -6=-6, -5=-5, -4=-4, -3=-3, -2=-2, -
     1=-1, 0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7}
Companhia={Companhia[0]}
Empregado={Empregado[0], Empregado[1], Empregado[2],
     Empregado[3], Empregado[4]}
Empregado.empregador={Empregado[0]->Companhia[0],
     Empregado[1]->Companhia[0], Empregado[2]->Companhia[0],
     Empregado[3]->Companhia[0], Empregado[4]->Companhia[0]}
Empregado.id={Empregado[0]->7, Empregado[1]->6,
     Empregado[2]->5, Empregado[3]->3, Empregado[4]->1}
```

Nessa instância gerada, os identificadores dos empregados **`Empregado[0], Empregado[1], Empregado[2], Empregado[3], Empregado[4]`** foram 7, 6, 5, 3 e 1, respectivamente. Assim, por serem os identificadores distintos, o predicado se torna valido para o universo de 5 empregados.

Capítulo 6

Conclusão

A linguagem UML é, sem dúvida, a mais utilizada na área de engenharia de *software* para modelagem de dados devido à facilidade encontrada no seu manejo e também por utilizar-se de representação gráfica.

Realizar a análise de diagramas de classes UML é sem dúvida tarefa importante. A busca pela corretude para assim poder validar o diagrama feita de forma visual, sem ferramentas, pode deixar passar erros de construção da linguagem como o esquecimento de elementos importantes e até mesmo de restrições necessárias e erros de inconsistência de projeto, como restrições conflitantes.

Dessa forma, nesta monografia foi desenvolvida uma ferramenta responsável por realizar as análises dos diagramas de classes UML de maneira automática. Essa ferramenta recebe um modelo representativo do diagrama com restrições em OCL e produz um XML resultante da análise.

6.1 Contribuições

A fim de realizar essa análise de forma automática, uma integração de componentes das linguagens UML e OCL com a linguagem *Alloy* foi desenvolvida. *Alloy* faz uso de código para realizar suas análises e possui uma ferramenta, o *Alloy Analyzer*, que gera instancias e contra-exemplos de acordo com o comando escolhido.

De início, foi criado um modelo representativo do diagrama de classes UML com restrições em OCL no formato XML visando as construções importantes da linguagem para a análise e que serve de entrada para a ferramenta. Nesse modelo foram também incluídos os predicados, funções, afirmações e comandos *Alloy* referentes ao diagrama, para assim poder analisar o mesmo utilizando as funcionalidades dessa linguagem.

Em seguida, foram elaboradas as regras de tradução com a finalidade de realizar o mapeamento entre os elementos das linguagens UML/OCL e a linguagem *Alloy*. Além disso, foi realizado o desenvolvimento de uma estrutura Java equivalente ao modelo de entrada, montada de acordo com a leitura dos dados do XML através do DOM, representando a AST do compilador.

Foi realizada ainda a tradução desse modelo Java para um código *Alloy* fazendo uso das regras de tradução entre os elementos das linguagens envolvidas. E ainda, a criação de uma

relação com o analisador de *Alloy*, responsável por realizar as análises do código gerado de acordo com os comandos recebidos no modelo representativo dado como entrada para o compilador e produzindo um XML com o resultado da presente análise para cada comando.

Para a integração do analisador com o compilador foi necessária a utilização das funcionalidades da API do *Alloy Analyzer*. Através do estudo da mesma foi possível a utilização de funções responsáveis por ler o código *Alloy* de entrada, verificar a existência de possíveis contra-exemplos ou instancias e gerar o XML para cada comando que possua uma solução satisfatória.

6.2 Trabalhos Futuros

Como dito anteriormente, a presente monografia demonstra o desenvolvimento de uma ferramenta de análise automática de diagramas de classes UML e restrições em OCL. No entanto essa análise é realizada para construções presentes no mapeamento desenvolvido entre os elementos mais importantes dos diagramas e que possuam um possível elemento que possa o corresponder em *Alloy*. Um trabalho futuro seria a busca por novas construções que não estejam presentes e que tenham a possibilidade de ser mapeadas, como o mapeamento da parte dinâmica dos diagramas.

Por ser o compilador desenvolvido de maneira modular, o acréscimo de alguma nova construção deverá ser realizado de maneira fácil, sem muitas complicações, e a partir da geração do código *Alloy*, também a análise poderá ser realizada.

Existem outros trabalhos futuros que poderão ser desenvolvidos em cima da ferramenta construída nesta monografia com a finalidade de melhorar de alguma forma a utilização da mesma. O primeiro deles seria a criação de uma forma de se adquirir os dados do diagrama de classes UML no seu formato gráfico original e assim traduzir para o modelo XML representativo dos diagramas.

Outro trabalho importante seria a criação de uma interface gráfica responsável por, após a geração do código *Alloy* resultante do compilador, solicitar ao usuário que informe os predicados, funções e afirmativas, bem como os seus comandos relacionados. Isso retiraria essas construções do XML representativo dado como entrada para o mesmo, dando mais sentido à ferramenta.

Bibliografia

[1] BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The Unified Modeling Language User Guide*. First Edition. Massachusetts: Addison Wesley, 1998. 482 p.
[2] PENDER, Tom. UML: A Bíblia. Editora Campus, 2004, 700p.
[3] WARMER, Jos; KLEPPE, Anneke. *The Object Constraint Language: Getting Your Models Ready for MDA*. Second Edition. Massachusetts: Addison Wesley, 2003. 240 p.
[4] *Alloy Homepage*. Disponível em: <http://alloy.mit.edu/>. Acesso em: 26 de março de 2007.
[5] JACKSON, Daniel. *Software Abstractions: Logic, Language, and Analysis*. First Edition. Massachusetts: Mit Press, 2006. 350p.
[6] *Java Technology*. Disponível em: <http://java.sun.com/>. Acesso em: 27 de março de 2007.
[7] *Eclipse – An Open Development Platform*. Disponível em: < http://www.eclipse.org/>. Acesso em: 20 de maio de 2007.
[8] DUESTERWALD, Evelyn. *Compiler Construction*. In: Joint European Conferences on Theory and Practice of Software, 2004, Barcelona, Spain.
[9] RITTGEN, Peter. *Enterprise Modeling and Computing with UML*. IGI Global, 2006, 314p.
[10] BLAHA, Michael; RUMBAUGH, James. *Modelagem e Projetos Baseados em Objetos com UML*. Editora Campus, 2006, 510p.
[11] DEBONI, José E. Z. *Modelagem Orientada a Objetos com UML*. 1 edition. Futura, 2003.
[12] *UML 2.0 OCL Specification*. Disponível em: <http://www.omg.org/docs/ptc/03-10-14.pdf>. Acesso em: 26 de março de 2007.
[13] WOODCOCK, Jim; DAVIES, Jim. Using Z: Specification, Refinement, and Proof. Upper Saddle River: Prentice Hall, 1996.
[14] RAY, Erik T. *Learning XML*. 2nd Edition. O'Reilly, 2003.
[15] DICK, Kevin. *XML: A Manager's Guide*. Second Edition. Addison Wesley, 2002.
[16] MARINI, Joe. *Document Object Model: Processing Structured Documents*. 1st edition. California: McGraw-Hill/Osborne, 2002.

Apêndice A

XML resultante do contra-exemplo

Este apêndice traz o XML resultante do contra-exemplo da subseção 5.3.1 do capítulo 5.

```
<alloy builddate="2007/Apr/04 12:54 EDT">
<instance filename="" bitwidth="4" command="Check horas for 3">
<sig name="Pessoa">
     <atom name="Professor[0]"/>
     <atom name="Aluno[0]"/>
     <atom name="Aluno[1]"/>
</sig>
<field name="idade">
     <type><sig name="Pessoa"/> <sig name="Int"/></type>
     <tuple><atom name="Professor[0]"/><atom name="7"/></tuple>
     <tuple><atom name="Aluno[0]"/><atom name="7"/></tuple>
     <tuple><atom name="Aluno[1]"/><atom name="5"/></tuple>
</field>
<sig name="Aluno" extends="Pessoa">
     <atom name="Aluno[0]"/>
     <atom name="Aluno[1]"/>
</sig>
<sig name="Professor" extends="Pessoa">
     <atom name="Professor[0]"/>
</sig>
<sig name="Disciplina">
     <atom name="Disciplina[0]"/>
     <atom name="Disciplina[1]"/>
</sig>
<field name="ministra">
     <type><sig name="Disciplina"/><sig name="Professor"/></type>
     <tuple><atom name="Disciplina[0]"/><atom
          name="Professor[0]"/></tuple>
     <tuple><atom name="Disciplina[1]"/><atom
          name="Professor[0]"/></tuple>
```

```
</field>
<field name="estuda">
     <type><sig name="Disciplina"/><sig name="Aluno"/></type>
     <tuple><atom name="Disciplina[0]"/><atom
          name="Aluno[1]"/></tuple>
     <tuple><atom name="Disciplina[1]"/><atom name="Aluno[1]"/>
     </tuple>
</field>
<field name="horasDiarias">
     <type><sig name="Disciplina"/><sig name="Int"/></type>
     <tuple><atom name="Disciplina[0]"/><atom name="4"/></tuple>
     <tuple><atom name="Disciplina[1]"/><atom name="4"/></tuple>
</field>
<sig name="Int">
     <atom name="-8"/>
     <atom name="-7"/>
     <atom name="-6"/>
     <atom name="-5"/>
     <atom name="-4"/>
     <atom name="-3"/>
     <atom name="-2"/>
     <atom name="-1"/>
     <atom name="0"/>
     <atom name="1"/>
     <atom name="2"/>
     <atom name="3"/>
     <atom name="4"/>
     <atom name="5"/>
     <atom name="6"/>
     <atom name="7"/>
</sig>
<sig name="seq/Int" extends="Int">
     <atom name="0"/>
     <atom name="1"/>
     <atom name="2"/>
</sig>
<set name="$a" type="Disciplina">
     <atom name="Disciplina[1]"/>
</set>
</instance>
</alloy>
```

Apêndice B

XML resultante da instância

Este apêndice traz o XML resultante da instância da subseção 5.3.2 do capítulo 5.

```
<alloy builddate="2007/Apr/04 12:54 EDT">
<instance filename="" bitwidth="4" command="Run idUnico for 5">
<sig name="Companhia">
     <atom name="Companhia[0]"/>
</sig>
<sig name="Empregado">
     <atom name="Empregado[0]"/>
     <atom name="Empregado[1]"/>
     <atom name="Empregado[2]"/>
     <atom name="Empregado[3]"/>
     <atom name="Empregado[4]"/>
</sig>
<field name="empregador">
     <type><sig name="Empregado"/><sig name="Companhia"/></type>
     <tuple><atom name="Empregado[0]"/><atom
          name="Companhia[0]"/></tuple>
     <tuple><atom name="Empregado[1]"/><atom
          name="Companhia[0]"/></tuple>
     <tuple> <atom name="Empregado[2]"/> <atom
          name="Companhia[0]"/></tuple>
     <tuple> <atom name="Empregado[3]"/> <atom
          name="Companhia[0]"/></tuple>
     <tuple> <atom name="Empregado[4]"/> <atom
          name="Companhia[0]"/></tuple>
</field>
<field name="id">
     <type><sig name="Empregado"/><sig name="Int"/></type>
     <tuple><atom name="Empregado[0]"/><atom name="7"/></tuple>
     <tuple><atom name="Empregado[1]"/><atom name="6"/></tuple>
     <tuple><atom name="Empregado[2]"/><atom name="5"/></tuple>
     <tuple><atom name="Empregado[3]"/><atom name="3"/></tuple>
```

```
      <tuple><atom name="Empregado[4]"/><atom name="1"/></tuple>
</field>
<sig name="Int">
      <atom name="-8"/>
      <atom name="-7"/>
      <atom name="-6"/>
      <atom name="-5"/>
      <atom name="-4"/>
      <atom name="-3"/>
      <atom name="-2"/>
      <atom name="-1"/>
      <atom name="0"/>
      <atom name="1"/>
      <atom name="2"/>
      <atom name="3"/>
      <atom name="4"/>
      <atom name="5"/>
      <atom name="6"/>
      <atom name="7"/>
</sig>
<sig name="seq/Int" extends="Int">
      <atom name="0"/>
      <atom name="1"/>
      <atom name="2"/>
      <atom name="3"/>
      <atom name="4"/>
</sig>
</instance>
</alloy>
```