UNIVERSIDADE da MADEIRA

Departamento de Matemática e Engenharias

MetaSketch OCL Interpreter

João Marcos Duarte Matos

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador: Professor Doutor Leonel Domingos Telo Nóbrega

28 de Novembro de 2008

Resumo

No contexto das tecnologias propostas pela OMG, o MOF é utilizado para definir a sintaxe de linguagens de modelação, contudo, os aspectos semânticos não podem ser capturados usando esta linguagem. A descrição dos aspectos não sintácticos é realizada com recurso à linguagem OCL. Consequentemente, para uma completa definição de uma linguagem de modelação é necessário incorporar o OCL no MOF, criando uma infra-estrutura que possui a expressividade necessária para realizar esta função.

Este projecto visa complementar a ferramenta de metamodelação MetaSketch Editor, introduzindo a capacidade de executar expressões em OCL e permitindo, desta forma, a verificação semântica dos modelos construídos usando o MetaSketch Editor. A gramática da linguagem OCL adoptada está de acordo com a especificação elaborada pela OMG (2006-05-01), juntando-se algumas contribuições de trabalhos existentes sobre esta linguagem. O projecto envolveu a implementação de um *parser* com recurso ao sistema GOLD Parser, a implementação da *standard library* do OCL em C# e, por último, a implementação de uma estratégia de execução das expressões em OCL.

Abstract

MOF is utilized for describing the modeling language's syntax; however semantic aspects cannot be described with this language. In the context of the technologies coming from OMG, the non-syntactic aspect description it's performed using OCL.

Consequently, for a complete modeling language definition it's necessary to incorporate the OCL into the MOF, forming an infrastructure with the needed expressiveness to perform this task. This project seeks to complement the metamodeling tool MetaSketch Editor. The adopted OCL grammar is based on the specification developed by OMG (2006-05-01) and on some existing work. The project involved a parser development using the GOLD Parser system, the OCL's standard library is implemented in C#, at last, an OCL expression execution strategy implementation.

Palavras Chave

OCL

MOF

MDA

MDD

Interpretador

Parser

Metamodelo

MetaSketch Editor

Metamodelação

Keywords

OCL

MOF

MDA

MDD

Interpreter

Parser

Metamodel

MetaSketch Editor

Metamodeling

Agradecimentos

Agradeço em primeiro lugar ao meu orientador, o Professor Doutor Leonel Domingos Telo Nóbrega, pela forma incansável como acompanhou a minha tese mestrado e me orientou. Mostrou uma invulgar disponibilidade para esclarecer dúvidas e me apontar na direcção correcta.

Dedico um agradecimento especial aos meus pais por terem financiado o curso que agora concluo. Agradeço também a todos os meus familiares por todo o apoio que me deram, especialmente nas alturas mais complicadas ao longo deste trabalho.

Sendo esta dissertação um projecto individual, não quero deixar de agradecer aos meus colegas que me acompanharam ao longo destes anos de curso.

Por fim agradeço a compreensão, apoio e carinho da minha namorada.

Índice

Índice de ilustrações

Índice de tabelas

Glossário

ASM – Abstract Syntax Model

BNF – Backus-Naur Form

CST – Concrete Syntax Tree

EBNF – Extended Backus-Naur Form

LALR – Algoritmo Look Ahead Left to Right

MDA – Model-Driven Architecture

MDD – Model-Driven Development

MOF – Meta-Object Facility

UML – Unified Modeling Language

OCL – Object Constraint Language

OMG – Object Management Group

1. Introdução

A Historia do desenvolvimento de software está pautada por sucessivos avanços no nível de abstracção. Já tivemos de programar com "zeros e uns" também conhecido por código binário, depois surgiu o assembly que se assemelha a um conjunto de mnemónicas para facilitar a programação. A dada altura surgem as primeiras linguagens de programação como o FORTRAN e que são traduzidas para linguagem máquina por compiladores. Actualmente utilizamos linguagens de programação, como por exemplo o Java, C#, python e php, resultantes da evolução contínua das linguagens de programação. Um exemplo dessa evolução é a introdução do paradigma Orientado a Objectos. Contudo estas linguagens continuam a ser, na sua essência, linguagens de programação e não acrescentam nenhum avanço significativo no nível de abstracção, pelo menos, não tão significativo como o salto do código binário para o assembly ou deste ultimo para as linguagens de programação.

O Desenvolvimento Dirigido Por Modelos ou MDD é uma abordagem ao desenvolvimento de software que assenta na utilização dos Modelos como artefacto principal, assim a actividade de modelação passa a ser a actividade principal. Deste modo os modelos passam a desempenhar o papel principal no processo de desenvolvimento de software. A Arquitectura Dirigida Por Modelos da OMG ou MDA é o principal padrão desta abordagem embora seja muitas vezes acusada de não alcançar a plenitude da abordagem por apenas se preocupar em criar um nível de dependência entre o código e os modelos utilizados para criar esse código. Num entanto persiste um problema, o facto dos modelos definidos não conterem toda a semântica necessária para que possam ser executados ou transformados automaticamente em programas, leva a que os modelos apenas sejam utilizados como suporte e como documentação no desenvolvimento de software, ficando o trabalho de tradução e transformação de modelos para código a cargo dos programadores. Tarefa cara, tediosa e imprecisa.

1.1. Motivação

Como qualquer outra linguagem, as Linguagens de Modelação têm de obedecer a um conjunto de regras. A esse conjunto de regras chamamos Metamodelo. Ou seja, os Metamodelos expressam as regras estruturais e semânticas das linguagens de modelação.

O MetaSketch cobre os aspectos de definição da Sintaxe Abstracta e Concreta de uma linguagem de modelação. Todavia, algumas definições importantes não são de natureza sintáctica e consequentemente não podem ser expressas usando o MOF. A linguagem OCL cobre esta lacuna. Sem a linguagem OCL seria impossível definir, por exemplo, que uma classe não pode ser superclasse de si mesma. Igualmente impossível seria a verificação do cumprimento de tal regra.

A linguagem de restrições OCL foi escolhida pela OMG como sendo a linguagem que introduz a possibilidade de definição de regras semânticas sobre os modelos que são

construídos usando a linguagem UML. No contexto da definição de linguagens de modelação usando a infra-estrutura criada para a definição do UML 2.0, a linguagem OCL possui um papel determinante na criação de ferramentas que permitam dar o adequado suporte na tarefa de definição de linguagens de modelação segundo esta abordagem. A inclusão do OCL no MetaSketch irá permitir um maior suporte à definição e utilização das Linguagens de Modeleção.

1.2. Descrição do Problema/Objectivos

Como já vimos na secção anterior, o OCL é utilizado para descrever a semântica das linguagens de modelação. A ferramenta MetaSketch Editor já suporta o MOF para a descrição e verificação da estrutura sintáctica, contudo o OCL ainda não é suportado, logo não é possível a descrição nem a verificação dos aspectos semânticos de uma linguagem de modelação.

O problema está em implementar um interpretador de OCL e integra-lo no MetaSketch Editor quer ao nível do MOF, quer ao nível das linguagens definidas utilizando o MOF. Esta integração irá permitir ao MetaSketch assistir na tarefa de definição sintáctica e semântica de novas linguagens de modelação.

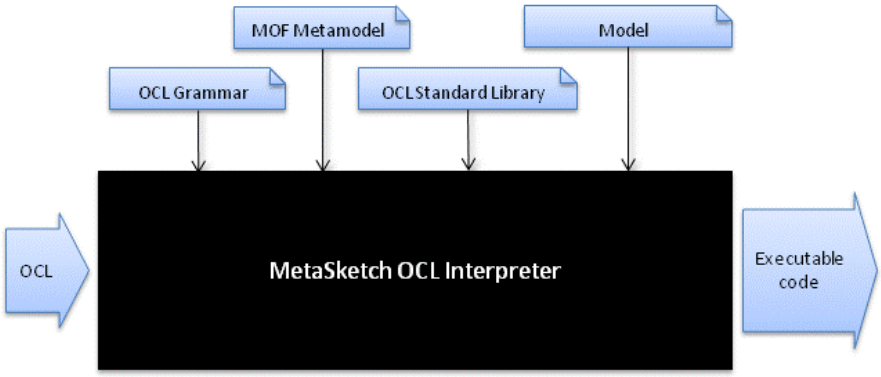


Ilustração 1 - Conceito principal do interpretador de OCL

O interpretador receberá informação sob a forma de expressões OCL e o resultado do processamento dessas regras deverá ser código executável que lhes é equivalente. Para efectuar este processamento o MetaSketch OCL Intrepreter tem ao seu dispor a gramática da linguagem OCL, o Metamodelo do MOF, a OCL Standard Library e o Modelo. Este último recurso, o Modelo, é o que iremos chamar de Modelo alvo, contem a informação sobre a qual as expressões OCL irão exercer as restrições.

1.3. Estrutura da Dissertação

No capítulo 2. «A Linguagem OCL» serão abordados alguns aspectos gerais sobre a linguagem, também vamos falar sobre a adopção do OCL pela OMG e a sua relação com o UML. Apresentarei também a versão da gramática presente no standard em que o trabalho é baseado.

Ainda no mesmo capítulo, mas já na secção 2.3. «OCL para Metamodelos», irei discutir a utilização da linguagem OCL na definição de metamodelos.

No capítulo 3. «Que ferramentas já existem» falo um pouco das ferramentas que actualmente suportam de alguma forma o OCL

No capítulo 4 «GOLD Parsing System» introduzo a ferramenta utilizada para definir a gramática LALR.

No capítulo 5. «MeatSketch OCL Interpreter», descrevo a arquitectura adoptada. É já na secção 5.1. «Tratamento da Gramática» que falo do trabalho realizado durante a transformação da sintaxe concreta existente na especificação numa gramática LALR equivalente. Detalho também as componentes da arquitectura adoptada que estão relacionadas com a sintaxe concreta.

Descrevo na secção 5.2. a implementação da *Standard Library* do OCL, apresentando e justificando as opções tomadas.

Já na secção 5.3. «Interpretação/Execução das Expressões em OCL» faço uma apresentação da estratégia de transformação das expressões OCL em funções C#.

Na secção 5.4. «Integração no MetaSketch Editor» descrevo a integração do interpretador no MetaSketch Editor.

Para concluir a minha tese termino com o capítulo 6. «Conclusão»

Em apêndice existem alguns documentos que pretendem complementar ou apoiar alguns conceitos abordados neste documento. Sempre que se mostrar pertinente indicarei qual o apêndice que deve ser consultado.

Neste documento todos os exemplos em OCL, BNF, ou C# utilizados são identificados por uma caixa que destaca esse exemplo do resto do texto. Para ajudar a distinguir os exemplos são utilizadas as seguintes caixas:

BNF:

```
! Exemplo qualquer de código BNF ou EBNF
```

OCL:

```
-- Exemplo qualquer de uma expressão OCL
```

C#:

```
// Exemplo qualquer de código C#
```

2. A Linguagem OCL

2.1. Introdução

Na versão portuguesa da *Wikipedia* podemos encontrar a seguinte tradução para *Object Constraint Language*:

> "…*Object Constraint Language* (ou linguagem para especificação formal de restrições, em português)." [1]

Embora a tradução encontrada seja de algum modo feliz, por capturar algumas das características da linguagem, irei utilizar as inicias OCL para me referir á linguagem em questão. Relembro que o leitor que já está familiarizado com a linguagem OCL pode saltar este capítulo que pretende ser uma introdução à linguagem.

A linguagem OCL foi desenvolvida por Jos Warmer na IBM e foi inicialmente utilizada como linguagem de modelação de negócio. Em 1997 a IBM em parceria com ObjecTime apresentam uma proposta á OMG onde se inseria a linguagem OCL. Partes desta proposta seriam posteriormente incluídas no UML 1.1.

O OCL surge como resposta a um problema muito particular, o facto de a linguagem UML não ter os artefactos necessários para que seja possível modelar os aspectos semânticos de um problema.

O OCL ao combinar características das linguagens de programação (e.g. forma de navegação, acesso a propriedades) com lógica de primeira ordem e a orientação a objectos, fá-la uma linguagem textual precisa que não possui a complexidade de uma linguagem matemática formal nem a ambiguidade da linguagem natural, ideal para modeladores de sistemas e programadores.

> "OCL supplements UML by providing expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. ..." [2]

O OCL caracteriza-se por ser uma linguagem declarativa indicando o "quê" e não o "como", é fortemente tipificada, cada expressão tem um tipo associado que pode ser primitivo (Boolean, Integer, Real, ou String), colecções (Set, OrderedSet, Bag, ou Sequence) ou outro tipo definido no seu contexto, a comparação directa entre valores de diferentes tipos não é possível. Estas e outras características são abordadas com mais detalhe na secção seguinte.

Uma expressão OCL é tida como atómica, ou seja, no instante da avaliação a expressão não pode ser subdividida sendo esta avaliação instantânea.

Durante a avaliação de uma expressão OCL, o sistema modelado permanece inalterado. A avaliação de expressões OCL não tem qualquer efeito sobre o estado do sistema, simplesmente retorna um valor. Porem existe um tipo de expressões OCL que

permitem especificar uma mudança de estado do sistema, as pós-condições. Este e outros tipos de expressões são também abordados com mais detalhe na secção seguinte.

Contudo embora o OCL tenha sido desenhado para evitar a complexidade das linguagens formais e a ambiguidade das linguagens naturais, determinadas restrições podem se tornar de tal forma complexas que seria preferível descreve-las através de linguagem natural. O OCL não é perfeito, é importante que se tente simplificar ao máximo as restrições a modelar.

2.2. UML e OCL 2.0

No contexto do UML o OCL tem duas funções distintas, é utilizado para formalizar a semântica da própria linguagem UML e também permite expressar, de uma forma precisa, constrangimentos sobre a estrutura dos modelos definidos usando UML. A primeira aplicação está relacionada com a Metamodelação e é abordada na secção 2.3. Nesta secção são abordados os aspectos relacionados com a expressam de constrangimentos sobre modelos UML.

A modelação diagramática de problemas e soluções é um meio simples e rápido de comunicação entre participantes de um projecto de desenvolvimento de software. Contudo é frequente o recurso a outros elementos não diagramáticos, para completar ou desambiguar os diagramas. Não são raras as situações em que temos que colocar notas nos modelos para que não existam múltiplas interpretações, ou para simplesmente dizer que um determinado atributo não pode exceder um determinado valor.

A linguagem de modelação UML, no que diz respeito à componente gráfica, em determinadas situações não é suficientemente expressiva. Uma solução passa pela utilização de uma linguagem que permita definir restrições sobre objectos e suas propriedades, complementando assim a expressividade da linguagem UML.

O OCL faz parte da especificação do UML desde a versão 1.1. A inclusão do OCL na especificação do UML veio permitir a definição de restrições sobre elementos da linguagem e assim responder às limitações do UML. Entre outras coisas incrementou o UML com os tipos já existentes no OCL e uma serie de operações sobre esses tipos.

O OCL tem vindo a amadurecer à mediada que o UML também evolui. A versão da linguagem OCL utilizada nesta dissertação é a versão 2.0, sendo esta a versão que está de acordo com o UML 2.0 e com o MOF 2.0. A especificação seguida foi a “OCL 2.0 - OCL 2.0 Specification, Version 2 - formal.2006-05-01”.

As linhas seguintes pretendem ser um resumo dos aspectos mais relevantes e que importam reter na definição de restrições OCL sobre modelos UML. Toda informação aqui apresentada está mais detalhada no documento da especificação oficial. A informação e os exemplos utilizados foram retirados do Capitulo 7 do documento da

especificação do OCL [4] e dos sites da EmPowerTec AG [17], da Klasse Objecten [19] e do projecto Parlez|UML [14].

No que diz respeito ao UML a linguagem tem um vasto leque de aplicações, refira-se:

- Como linguagem para *queries*
- Especificar invariantes de classes e especificar tipos no modelo de classes
- Especificar tipos invariantes para *Stereotypes*
- Descrever pre e pos-condições em operações e métodos
- Descrever guardas
- Especificar mensagens e acções
- Especificar restrições em operação
- Especificar regras de derivação para atributos

O número de aplicações das expressões OCL pode ser aumentado.

Quais os três tipos de restrições existentes:

- Invariantes: restrições que se verificam sempre. “inv”
- Pre-condições: condições que se verificam a quando da execução de uma dada operação “pre”
- Pos-condições: condições que se verificam após a execução de uma dada operação “post”

È uma linguagem tipificada. Os quatro tipos primitivos são:

- Integer – Numero inteiro de qualquer tamanho
- Rear – Numero real de qualquer tamanho
- String – Conjunto de caracteres (‘UMa’)
- Boolean – *true* / *false*

A linguagem possui também vários conjuntos de operadores. Cada conjunto opera sobre um determinado tipo primitivo. Operadores sobre inteiros e reais (Tabela 1), operadores sobre Strings (Tabela 2) e operadores sobre booleanos (Tabela 3)

Tabela 1 - Operadores sobre Inteiros e Reais

Operador	Notação	Tipo do resultado
Equals	a = b	Boolean
Not equals	a <> b	Boolean
Less	a < b	Boolean
More	a > b	Boolean
Less or equal	a <= b	Boolean
More or equal	b >= b	Boolean
Plus	a + b	Intiger / Real
Minus	a – b	Intiger / Real
Multiply	a * b	Intiger / Real
Divide	a / b	Real
Modulus	a.mod(b)	Intiger
Intiger division	a.div(b)	Intiger
Absolute value	a.abs	Intiger / Real
Maximum	a.max(b)	Intiger / Real
Minimum	a.min(b)	Intiger / Real
Round	a.round	Intiger
Floor	a.floor	Intiger

Tabela 2 - Operações sobre Strings

Operador	Notação	Tipo do resultado
Concatenation	s.concat(String)	String
Size	s.size	Integer
To lower case	s.toLower	String
To upper case	s.toUpper	String
Substring	s.substring(int, int)	String
Equals	s1 = s2	Boolean
Not equals	s1 <> s2	Boolean

Tabela 3 - Operações sobre booleanos

Operador	Notação	Tipo do resultado
Or	a or b	Boolean
And	a and b	Boolean
Exclusive or	a xor b	Boolean
Negation	not a	Boolean
Equals	a = b	Boolean
Not Equals	a <> b	Boolean
Implication	a implies b	Boolean
If then else	if a then b1 else b2 endif	Type of b

Existem quatro tipos de colecções:

- Set – um elemento só pode ocorrer uma vez e sem nenhuma ordem especifica.
- OrderedSet – um elemento só pode ocorrer uma vez, possui a noção de ordem.
- Bag – um elemento pode ocorrer mais do que uma vez e sem ordem.
- Sequence – um elemento pode ocorrer mais do que uma vez, possui noção de ordem.

Tabela 4 - Descrição dos metodos existentes nas colecções

Operação	Descrição
size	Número de elementos na colecção
count(object)	Número de vezes que o objecto ocorre na colecção
includes(object)	Retorna True se o objecto for um elemento da colecção
includesAll(collection)	Retorna True se todos os elementos desta colecção pertencerem á colecção inicial
isEmpty	Retorna True se a colecção estiver vazia
notEmpty	Retorna True se a colecção contiver algum elemento
iterate(expression)	A expressão é avaliada para todos os elementos da colecção
sum(collection)	Somatório de todos os elementos da colecção
exists(expression)	Retorna True se a expressão for True para pelo menos um elemento
forAll(expression)	Retorna True se a expressão for valida para todos os elementos da colecção
select(expression)	Retorna o subconjunto de elementos que satisfazem a expressão
reject(expression)	Retorna o subconjunto de elementos que não satisfazem a expressão
collect(expression)	Forma uma nova colecção com os elementos retornados pela expressão
one(expression)	Retorna True se um e apenas um elemento da colecção satisfizer a expressão
sortedBy(expression)	Retorna uma sequencia com todos os elementos da colecção na ordem especificada (a expressão tem que possuir o operador <)

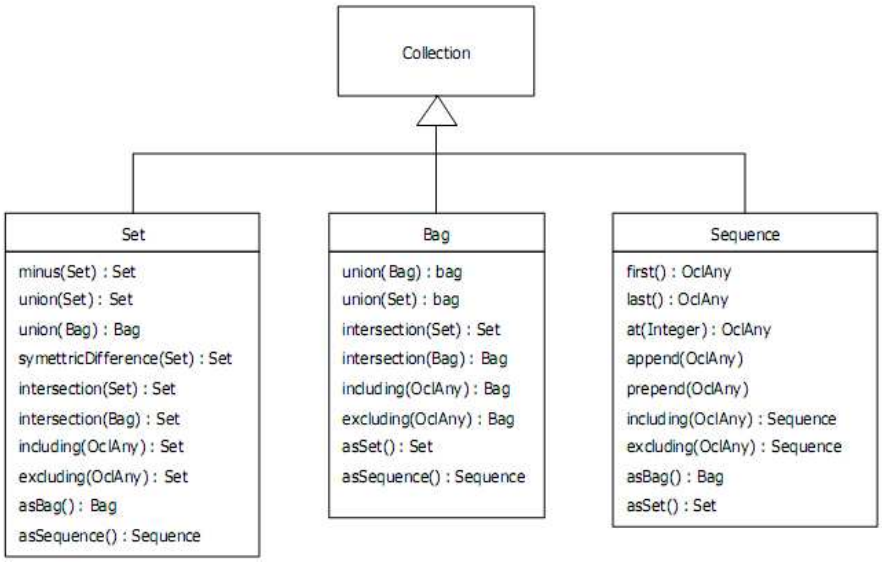


Ilustração 2 - Diagrama de classes com métodos específicos de Set, Bag, Sequence

2.2.1. Morfologia de uma expressão OCL

Existem diferentes modos de apresentação das expressões OCL podem ser colocadas directamente no diagrama na forma de nota, como mostra a Ilustração 3, ou podem ser apresentadas num documento separado onde deverá constar todas as expressões. No último caso a indicação do contexto ganha uma importância redobrada.

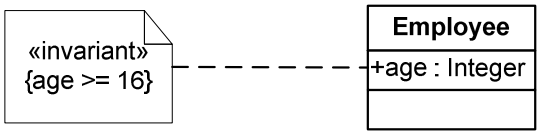


Ilustração 3 - Exemplo de uma expressão OCL

A nota que representa um invariante na Ilustração 4 é equivalente a ter:

```
context Employee
inv: self.age >= 16
```

ou

```
context Employee
inv: age >= 16
```

As expressões são compostas por um contexto, que neste caso é “Employee” o que indica que esta restrição se aplica à classe “Employee”. E um indicador que indica quando é que a restrição se verifica, que neste caso é “inv” o que significa que tem que se verificar sempre.

Por vezes é necessário indicar explicitamente qual a instancia da classe a que estamos a referir, para isso dispomos da palavra reservada “self”.

A palavra reservada “@pre” indica o valor do atributo ou associação antes da execução de uma operação. Devera estar depois do nome do artefacto a que nos referimos. No segundo exemplo apresentado na secção 2.2.3. é exemplificada a utilização do “@pre”

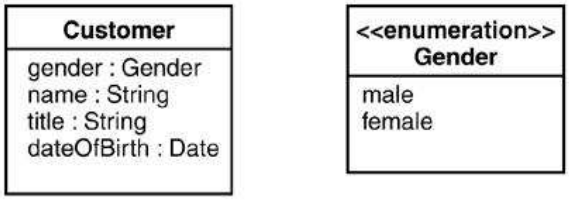


Ilustração 4 - Exemplo de uma classe "Customer"

```
context Customer
inv: gender = Gender::male implies title = 'Mr.'
```

Podemos utilizar expressões OCL para indicar o valor inicial de um atributo ou associação, para isso indicamos qual o atributo a que nos referimos no “context”, em seguida podemos ou não indicar qual o tipo desse atributo. O “init” é utilizado quando queremos dizer que o atributo ou associação em questão é um determinado valor que nós já sabemos.

```
context Typename::attributeName: Type
init: -- expressão que indique o valor inicial
```

Utilizamos o “derive” quando não sabemos de ante mão qual o valor deste atributo, mas sabemos que é calculado a partir de outros factores, por exemplo outros atributos.

```
context Typename::assocRoleName: Type
derive: -- expressão que indique o valor inicial derivado
```

2.2.2. Associações e Relações

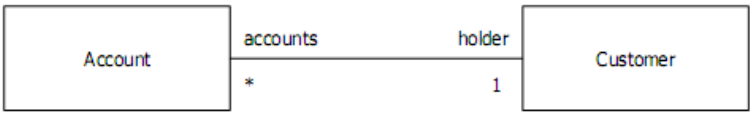


Ilustração 5 - Exemplo de uma relação entre as classes "Account" e "Customer"

Em UML as extremidades das associações tem um nome, são estes nomes que nos permitem aceder á classe do outro lado da associação. Ou seja, se estivermos na classe “Customer” e pretendemos aceder a classe “Account” utilizamos a extremidade “accounts” como mostra o seguinte exemplo:

```
...
Customer.accounts
...
```

No sentido contrário, se estivermos na classe “Account” e pretendemos aceder ao “Customer”, utilizamos a extremidade “holder”:

```
...
Account.holder
...
```

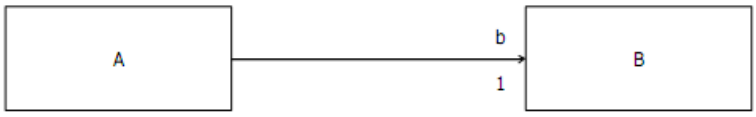


Ilustração 6 - Exemplo de uma associação navegável de "A" para "B"

Em UML as associações podem ter um sentido de navegabilidade como mostra a Ilustração 6, neste caso o acesso de “A” para “B” é permitido e representa-se da seguinte forma:

```
...
A.b
...
```

Contudo o acesso inverso não é possível:

```
...
B.a
...
```

Não é permitido

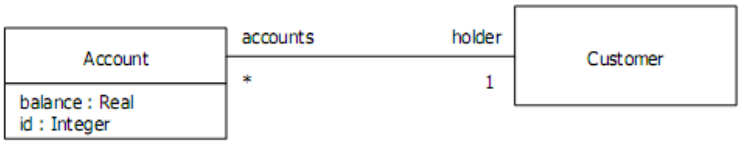


Ilustração 7 - Exemplo de como aceder a um atributo de uma classe

Neste caso a existe uma relação de um para muitos, cada “Customer” pode ter mais que uma “Account”. Esta relação é representada do lado do “Customer” com uma colecção chamada “accounts” quando pretendemos aceder ao “balance” de uma determinada “Account” podemos fazer o seguinte:

```
...
Customer.accounts->select(id = 2324).balance = 0
...
```

Quando estamos a trabalhar com colecção temos que utilizar sempre a seta “->”. Está errado utilizar o ponto “.” como no exemplo:

```
...
Customer.accounts.balance = 0
...
```

Não pode ser avaliado

2.2.3. Exemplos práticos

Nesta secção temos dois exemplos que elucidam a utilização de expressões OCL para especificar regras de negócio. No primeiro exemplo exemplifica uma situação onde temos que definir invariantes. No segundo exemplo são definidas pres e pos-condições.

Exemplo 1 – Empresa e empregados

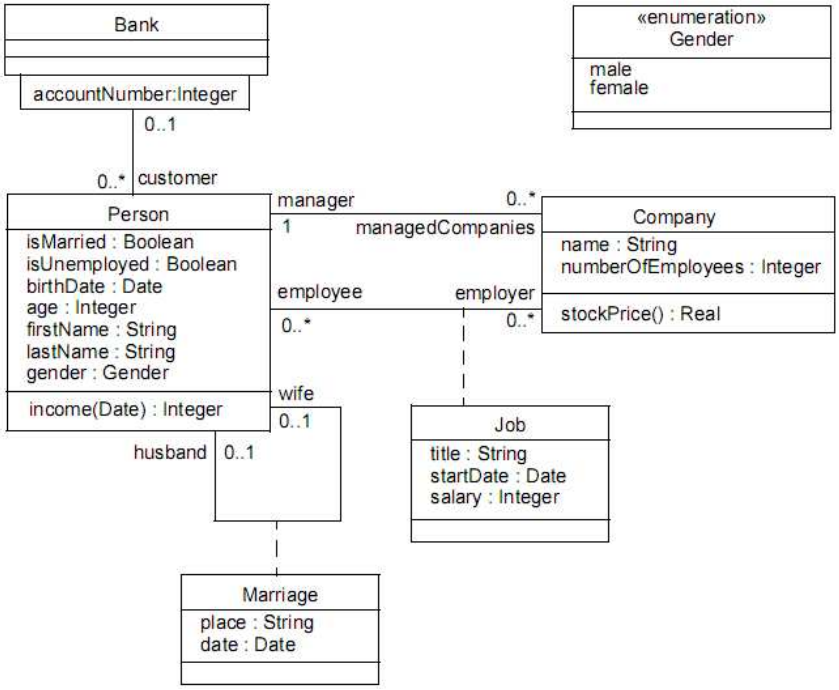


Ilustração 8 - Diagrama de classes, relação empresa e empregado

Neste exemplo iremos considerar um relacionamento entre uma empresa e seus empregados, este relacionamento é descrito no diagrama de classes UML da Ilustração 8. Contudo existem regras de negócio que não se consegue expressar através do diagrama de classes.

Consideremos as seguintes regras de negócio:

1. "If a person is male he can't have a husband and if he has a wife it must be female. If a person is female, she can't have a wife and if she has a husband it must be male."

2. "No employee may be older than 65 years."

3. "If a person is employed at a company, it is not unemployed."

Não existe nada no diagrama de classes que impeça que um homem seja casado com outro homem, o que viola claramente a regra de negócio 1. Também não existe nada que impossibilite um empregado ultrapassar os 65 anos de idade continuando a trabalhar para esta empresa, o que desrespeita a regra 2. Por fim também nada

impede de uma pessoa mesmo estando empregada nesta empresa possa declarar-se desempregada, o que também viola a regra 3.

Para que estas regras sejam implementadas e posteriormente respeitadas vamos definir as regras tirando partido da linguagem OCL:

Regra 1

```
context Person
inv: wife->notEmpty() implies
          (wife.gender = Gender::female and gender = Gender::male)
inv: husband->notEmpty() implies
          (husband.gender = Gender::male and gender = Gender::female)
```

Regra 2

```
context Company
inv: self.employee->forAll(age <= 65)
```

Regra 3

```
context Company
inv: employee->exists(isUnemployed) = false
```

Exemplo 2 – Stack (Pila)

No exemplo anterior vimos três invariantes, o próximo exemplo servirá para mostrar uma pos e uma pré-condição. Para o efeito utilizaremos uma pila que tem o comportamento normal de uma pila.

Comportamento da classe Stack:

1. "If an element is added to the stack, it is placed at its top."
2. "If the stack is empty, retrieving the top element is not possible."
3. "The operation pop() removes the element, that is in top of the stack and returns this element."

Vamos considerar a seguinte classe que podia ter sido definida em C#:

```
class Stack
{
          Stack();
          Object pop();
          void push(Object anObject);
          Object top();
          int size();
          boolean isEmpty();
}
```

Modelação do comportamento da classe “Stack”

```
context Stack::pop(): Object
         pre notEmpty: isEmpty() = false
         post topElementReturned: result = self@pre.top()
         post elementRemoved: size() = self@pre.size()-1

context Stack::top(): Object
         pre notEmpty: isEmpty() = false

context Stack::push(anObject: Object): void
         post pushedObjectIsOnTop: top() = anObject
```

2.3. OCL para Metamodelos

Como já vimos na secção 1.1 um Metamodelo é um modelo que descreve uma linguagem de modelação, como tal descreve os aspectos estruturais e semânticos dessa linguagem. Vimos também que os aspectos estruturais são definidos utilizando MOF e que os aspectos semânticos são descritos com OCL.

Ao ser utilizado para definir a semântica das linguagens de modelação, o OCL assume assim um papel preponderante na Metamodelação. Se excluirmos a linguagem OCL, as outras alternativas que nos permitem descrever a semântica de uma linguagem, são: a linguagem natural e as linguagens formais. As linguagens formais são praticamente ininteligíveis para pessoas sem formação matemática, e a linguagem natural é ambígua o que não permite a criação de um conjunto de automatismos que possibilitam a verificação e a validação das regras semânticas estabelecidas.

Portanto o OCL não só veio simplificar a descrição semântica, como também veio permitir a criação de verificadores e interpretadores que irão assistir na criação de Metamodelos, mais concretamente, irão verificar se as regras semânticas estão bem definidas e posteriormente irão verificar se estas são respeitadas pelos modelos.

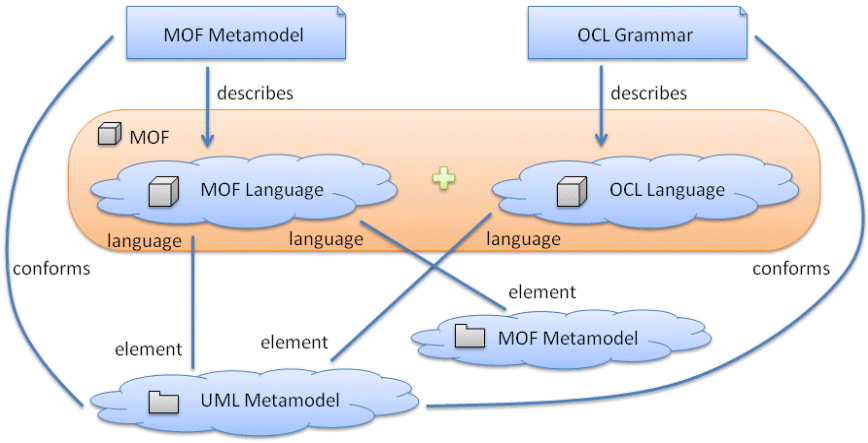


Ilustração 9 - O OCL na Metamodelação

O Metamodelo do MOF descreve a linguagem MOF que conjuntamente com a linguagem OCL é a base do Metamodelo do UML (ver Ilustração 9). Neste nível possuímos todos os elementos que tornam possível a criação de novas linguagens de modelação. Nesta perspectiva a linguagem UML é apenas um exemplo de uma linguagem de modelação descrita por um Metamodelo que é um elemento da linguagem MOF e da linguagem OCL, isto é, o Metamodelo do UML é um modelo que descreve a linguagem UML e que obedece à notação da linguagem MOF e à notação da linguagem OCL. Logo sempre que se fala em criar novas linguagens de modelação, estamos na realidade a falar em definir Metamodelos para essas novas linguagens, para isso recorremos à linguagem MOF para descrever a sintaxe e à linguagem OCL para descrever a semântica.

Para melhor perceber a importância da Metamodelação sobre a modelação e qual o papel do OCL em ambos os processos temos que ter uma visão mais "holística" (ver Ilustração 10). Atendendo a esta perspectiva é também interessante observar que o próprio Metamodelo do MOF é também definido utilizando a linguagem MOF.

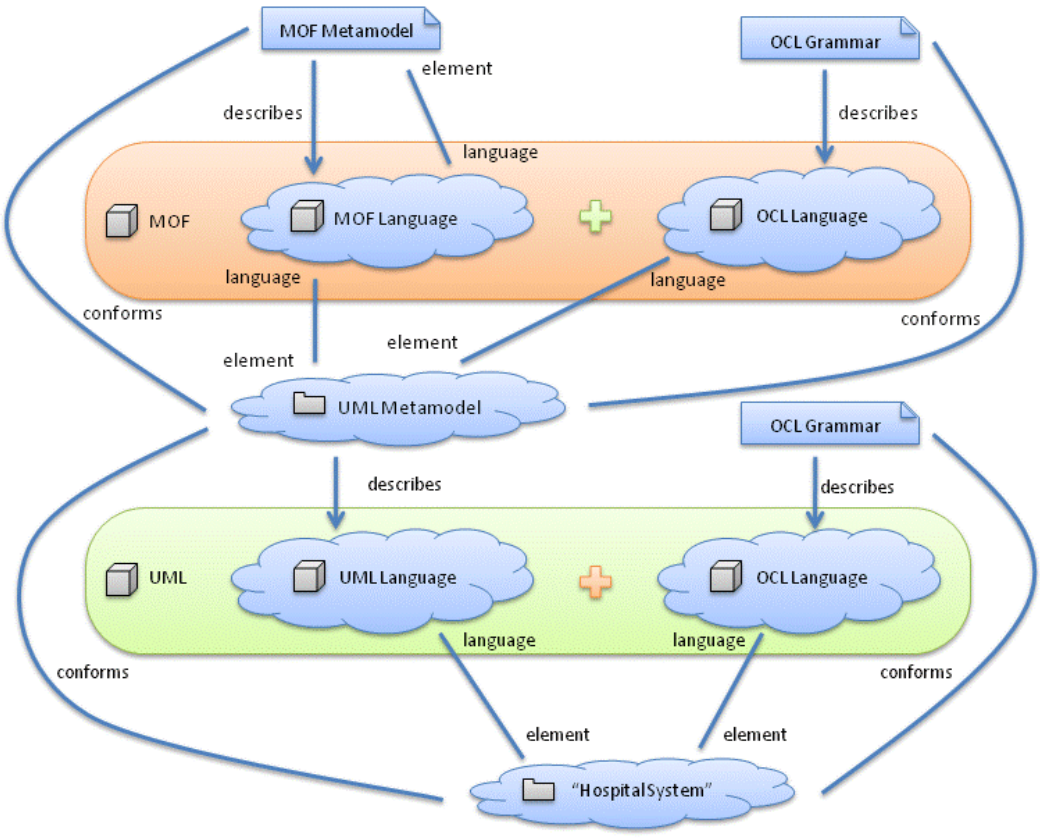


Ilustração 10 - O OCL na Metamodelação e na Modelação

No que diz respeito à definição de um Modelo, que por exemplo descreva um sistema para um Hospital (ver Ilustração 10Ilustração 10Ilustração 10 - O OCL), os aspectos envolvidos são os mesmos, continua a existir uma estrutura e uma semântica. Um

sistema hospitalar pode ser descrito utilizando a linguagem UML para descrever a estrutura e a linguagem OCL para descrever a semântica. O modelo terá que respeitar a notação do UML e do OCL. São evidentes as semelhanças com a Metamodelação.

É importante salientar também que o Metamodelo do UML está para a linguagem UML como a Gramática do OCL está para a linguagem OCL.

Quando é necessário verificar se um determinado modelo segue correctamente a sintaxe da linguagem modelação utilizada, é feita uma verificação de conformidade com o Metamodelo o que na Ilustração 10 é representado por “conforms”. Contudo esta verificação destina-se apenas aos aspectos sintácticos ou estruturais do modelo. A semântica do modelo por ser descrita noutra linguagem, o OCL, obriga a que a verificação da notação seja feita perante a Gramática do OCL.

Contudo a verificação do OCL levanta outros problemas. Sobre o OCL podem ser feitas verificações sintácticas e verificações semânticas, enquanto a primeira está relacionada com a Gramática do OCL, a segunda verificação não se adivinha tão trivial. A semântica das expressões OCL muitas vezes depende do seu contexto.

3. Ferramentas para o OCL

Actualmente já existem algumas ferramentas que de alguma forma suportam o OCL. Estas ferramentas podem ser ferramentas independentes, também conhecidas por *standalone tools*, ou as ferramentas que actuam como Plug-in's ou módulos para outras ferramentas, como o Eclipse, Netbeans, Magic Draw UML, ArgoUML, ou o RationalRose.

Esta área tem evoluído muito ao longo dos últimos tempos, muito recentemente, em 30 de Setembro de 2008, realizou-se em Toulouse um Workshop intitulado "8th OCL Workshop at the UML/MoDELS Conferences" onde foram demonstradas várias ferramentas que suportam o OCL, uma grande parte destas ferramentas são fruto de projectos recentes. Neste capítulo apenas irei falar das ferramentas que acompanhei e consegui reunir informação desde o inicio deste projecto.

Uma das referências na área é a Dresden OCL Toolkit [22], implementada em Java e desenhada para ser utilizada como um conjunto de bibliotecas que podem ser introduzidas ou reutilizadas em outras ferramentas, é actualmente mantido por estudantes e investigadores do Software Technology Group da Technische Universität Dresden. A versão mais recente desta Toolkit é a Dresden OCL2 for Eclipse, utiliza o *pivot model* desenvolvido por Matthias Bräuer como formato de troca de Modelos e Metamodelos. O objectivo desta arquitectura é estabelecer uma camada de abstracção que permita a interpretação de expressões OCL sobre instâncias de uma determinada linguagem como UML ou MOF. Além do repositório do NetBeans MDR, que já era utilizado na versão anterior da Toolkit, também é suportado o repositório EMF. Esta versão por ter sido lançada muito recentemente (9/12/2008) ainda não é utilizada em nenhum projecto. Contudo as anteriores versões são utilizadas em ferramentas como o ArgoUML.

O ArgoUML permite criar restrições sobre elementos de modelos UML. Através da utilização da Dresden OCL Toolkit é possível efectuar verificações ao nível da sintaxe e dos tipos. A funcionalidade dispõe ainda de um conjunto de *comboboxes* que permitem uma rápida utilização dos métodos existentes na Standard Library do OCL. Como é utilizada uma das primeiras versões do Toolkit, o Metamodelo do UML suportado é a versão 1.4.

O Oclarity é uma ferramenta desenvolvida pela EmPowerTec AG [17] e está actualmente disponível em duas versões, a Oclarity/XMI é uma aplicação *standalone* e a Oclarity for RationalRose é a versão que actua como Plug-in para o RationalRose 2000 (versão anterior à aquisição da Rational pela IBM). Ambas oferecem funcionalidades semelhantes, como a verificação sintáctica e semântica das expressões OCL, a capacidade de verificar todas as expressões no modelo de uma só vez e um editor sensível á sintaxe. O que destaca um do outro é o facto de a solução *standalone* suportar várias ferramentas de modelação que tenham a capacidade de exportar o modelo para um formado XMI compatível, actualmente os formatos suportados são do MagicDraw e do EnterPrise Architect.

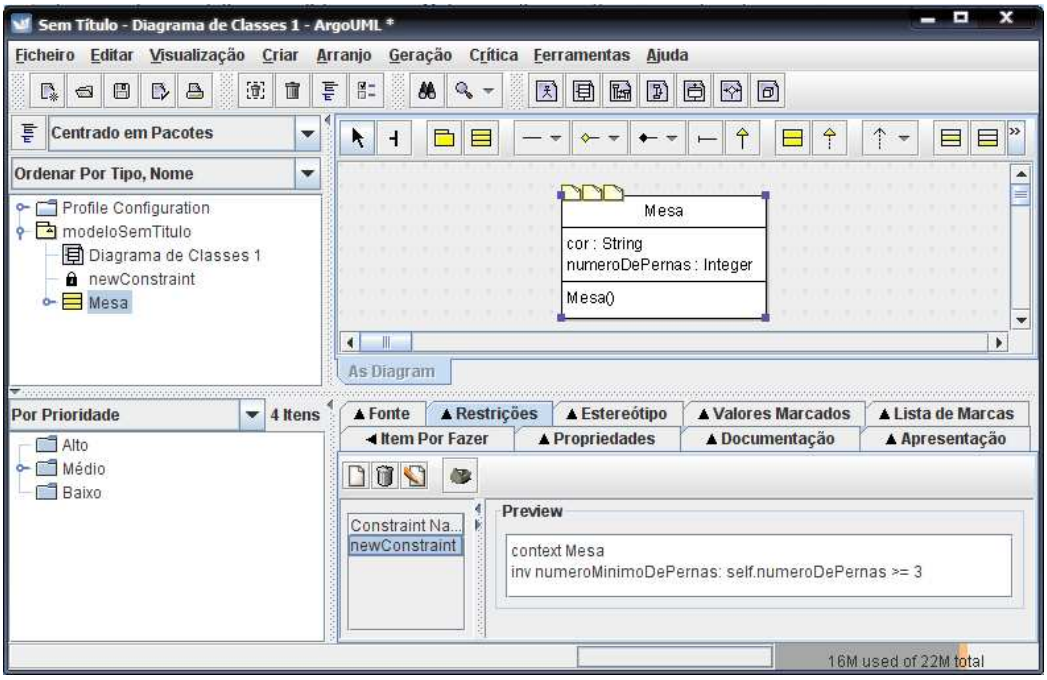


Ilustração 11 - ArgoUML, restrições sobre elementos do modelo

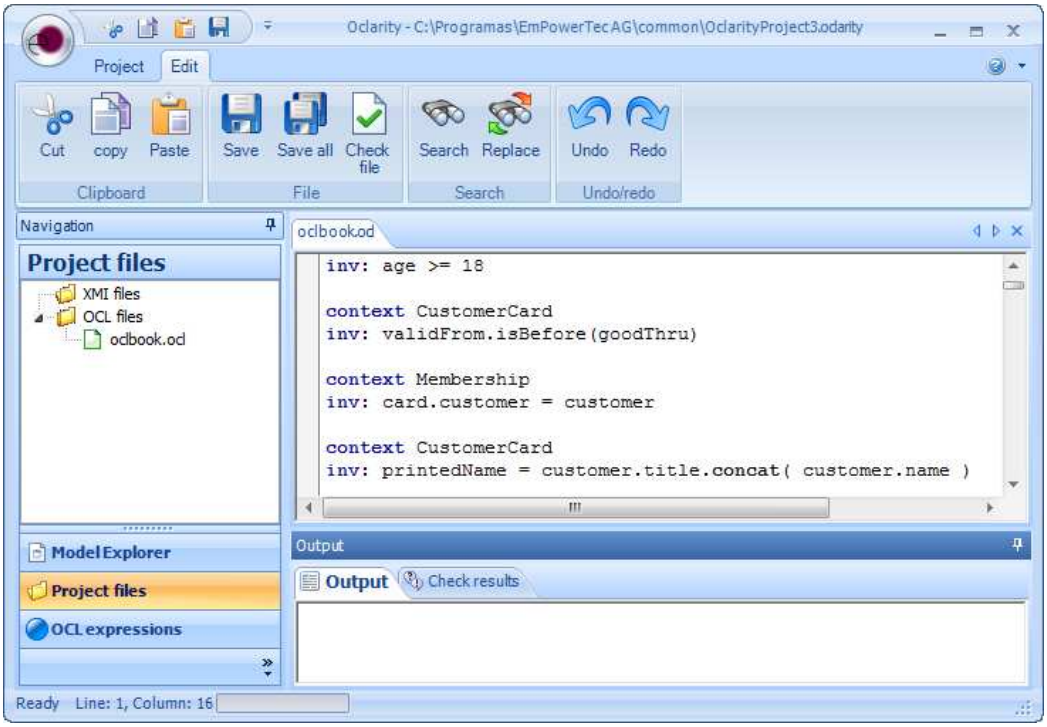


Ilustração 12 - Oclarity/XMI

A ferramenta Octopus foi desenvolvida por Jos Warmer e Anneke Kleppe [21] e a versão mais recente é a versão 2.2.0 que suporta o OCL 2.0. Esta ferramenta actua como Plug-in para o Eclipse e tem a capacidade de verificar a sintaxe e os tipos das expressões OCL, bem como a verificação da correcta utilização dos elementos do modelo, como associações, métodos e atributos. Possui também a capacidade de traduzir, em fase de protótipo, modelos UML, incluindo as expressões OCL, em código Java. Os modelos são importados para ferramenta utilizando o formato XMI, uma vez importados é possível definir restrições sobre o modelo.

A Tabela 5 pretende ser uma comparação das quatro ferramentas aqui abordadas. Uma vez que as ferramentas apresentadas possuem muitos elementos em comum como a verificação sintáctica, os aspectos comparados foram escolhidos para apontar as principais diferenças.

Tabela 5 - Comparação entre diferentes ferramentas que suportam OCL

Tipo de ferramenta:	Dresden OCL2 for Eclipse	ArgoUML[1]	Oclarity	Octopus
Standalone	Não[2]	Sim	Sim	Não
Plug-in / Modulo	Sim	Não	Sim	Sim
Permite restrições sobre instancias de:	Dresden OCL2 for Eclipse	ArgoUML	Oclarity	Octopus
UML	Sim (v2.0)	Sim (v1.4)	Sim (v2.0)	Sim (v2.0)
MOF	Sim[3]	Não	Não	Não
Funcionalidades suportadas:	Dresden OCL2 for Eclipse	ArgoUML	Oclarity	Octopus
Verificação da sintaxe	Sim	Sim	Sim	Sim
Verificação de tipos	Sim	Sim	Sim	Sim
Verificação cruzada com modelo	Sim	Sim	Sim	Sim
Verificação semântica	Sim	Não	Sim	Sim
Geração de código	Sim	Não	Não	Sim[4]

A grande maioria destas ferramentas, eventualmente poderá existir alguma que não tenha mencionado, mas na sua maioria, não exploram o facto de a infra-estrutura criada para a Metamodelação utilizar o OCL na definição de regras semânticas sobre os Metamodelos. Um melhor reconhecimento da potencialidade de uma ferramenta que possa verificar se essas regras são ou não compridas por modelos criados com uma linguagem metamodelada usando esta infra-estrutura, pode levar a que mais ferramentas de Modelação ou de Metamodelação possuam mecanismos de verificação sintáctica e semântica dos seus modelos, onde seria indicado quais os elementos que infringiam as regras da linguagem utilizada.

[1] Utiliza a Dresden OCL Toolkit, uma versão mais antiga que a Dresden OCL2 for Eclipse e que já caiu em desuso.

[2] Possui algumas ferramentas *standalone* que permitem testar a Toolkit, mas não foi desenhada como tal.

[3] Através do *pivot model*.

[4] Pode ser gerado código em fase de prototipo.

4. GOLD Parsing system

Esta secção pretende ser uma introdução ao sistema de *parsing* utilizado, são apontados também os factores que levaram à utilização deste sistema de *parsing*. A informação aqui presente foi sintetizada a partir do site oficial da ferramenta [16] e da Wikipedia [20].

O GOLD é um sistema de *parsing* que nos permite desenvolver linguagens de programação, linguagens de *scripting* e interpretadores.

Tal como noutros sistemas semelhantes, este sistema utiliza um algoritmo LALR para analisar a sintaxe e um DFA ou *Deterministic Finite Automaton* para identificar unidades lexicais. O que distingue o GOLD Parser dos restantes sistemas e o que o torna interessante para este projecto é a sua decomposição em três componentes lógicas: a primeira componente, *Builder*; a segunda, *Engine*; a terceira componente é a *Compiled Grammar Table* que é um ficheiro que actua como intermediário entre o *Builder* e o *Engine* (ver Ilustração 13).

Ilustração 13 - Componentes lógicas do GOLD parsing system

O *Builder* lê uma gramática definida com uma sintaxe proprietária, a *GOLD Meta-language*, e produz tabelas de *parsing* LALR e DFA que são passadas para um ficheiro *Compiled Grammar Table* com uma extensão cgt. O *Builder* é uma aplicação que corre em sistemas Windows 32-bit.

O ficheiro *Compiled Grammar Table* contém informação gerada pelo *Builder* que posteriormente é passado ao *Engine* (ver Ilustração 13).

É no *Engine* que se encontram as implementações dos algoritmos LALR e DFA e é feito o *parsing* propriamente dito, para o efeito, o *Engine* lê o ficheiro .cgt e um outro ficheiro que contem o texto a ser reconhecido, que no nosso caso é um ficheiro com expressões OCL (ver Ilustração 14). Como a gramática já sofreu um pré-tratamento, neste estágio o *parsing* é bastante mais simples.

No site oficial da ferramenta existem várias implementações do *Engine* em diferentes linguagens de programação, durante este projecto foi utilizada uma implementação

em C# mais concretamente a versão 2.1 do Morozov C# Engine desenvolvido por Vladimir Morozov.

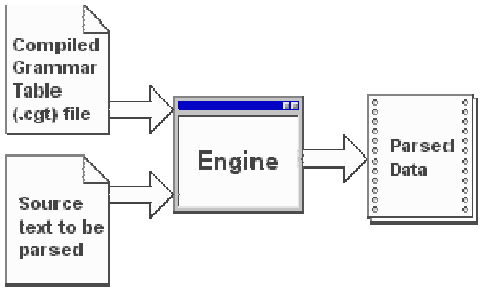


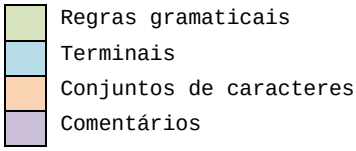
Ilustração 14 - Funcionamento do *Engine*

A GOLD Meta-language é a notação que é utilizada para definir as gramáticas aceites pelo sistema GOLD e combina três outras notações: para a definição das regras é utilizado a notação BNF (Backus-Naur Form); os terminais são representados com recurso a expressões regulares; e os conjuntos de caracteres são representados em notação de conjuntos.

Tabela 6 - Exemplo de uma gramática "Simple" definida para o GOLD parsing system

```
"Name"    = 'Simple'
"Author"  = 'Devin Cook'
"Version" = '2.1'
"About"   = 'This is a very simple grammar designed for use in examples'

"Case Sensitive" = False
"Start Symbol"   = <Statements>

{String Ch 1} = {Printable} - ['']
{String Ch 2} = {Printable} - ["]

Identifier    = {Letter}{AlphaNumeric}*

! String allows either single or double quotes

StringLiteral = ''  {String Ch 1}* ''
              | '"' {String Ch 2}* '"'

NumberLiteral = {Number}+('.'{Number}+)?

Comment Start = '/*'
Comment End   = '*/'
Comment Line  = '//'

<Statements>  ::= <Statements> <Statement>
                |  <Statement>

<Statement>   ::= display <Expression>
                |  display <Expression> read ID
                |  assign ID '=' <Expression>
                |  while <Expression> do <Statements> end
                |  if <Expression> then <Statements> end
                |  if <Expression> then <Statements> else <Statements> end

<Expression>  ::= <Expression> '>'  <Add Exp>
                |  <Expression> '<'  <Add Exp>
                |  <Expression> '<=' <Add Exp>
                |  <Expression> '>=' <Add Exp>
                |  <Expression> '==' <Add Exp>
                |  <Expression> '<>' <Add Exp>
                |  <Add Exp>

<Add Exp>     ::= <Add Exp> '+' <Mult Exp>
```

```
                |  <Add Exp> '-' <Mult Exp>
                |  <Add Exp> '&' <Mult Exp>
                |  <Mult Exp>

<Mult Exp>     ::= <Mult Exp> '*' <Negate Exp>
                |  <Mult Exp> '/' <Negate Exp>
                |  <Negate Exp>

<Negate Exp>   ::= '-' <Value>
                |  <Value>

<Value>        ::= Identifier
                |  StringLiteral
                |  NumberLiteral
                |  '(' <Expression> ')'
```



Este sistema, ao contrário de outros sistemas semelhantes como o YACC, não possui nenhum mecanismo que permita definir separadamente a precedência de operadores. Contudo essa precedência pode ser estabelecida na forma como as regras dos operadores são formadas, isto implica um maior número de regras. A gramática da Tabela 6 possui as regras `<Expression>`, `<Add Exp>` e `<Mult Exp>` que são exemplos da definição da precedência entre os operadores. Definir cada uma dessas regras adicionais acaba por ser um processo tedioso, prejudica o desempenho do parser e torna a gramática mais difícil de ler. Contudo nem tudo são aspectos negativos, o facto da precedência de operadores ser definida desta forma, permite que quem está a implementar o interpretador esteja mais consciente do processo de escolha de operadores durante a construção da árvore de sintaxe abstracta.

A Tabela 7 representa uma comparação entre o sistema de parsing GOLD e outros sistemas alternativos. Esta tabela foi retirada do site oficial da ferramenta e possui vários pontos de comparação, contudo, os pontos que interessam mais são o algoritmo de parsing e as linguagens suportadas. Procuramos um parser que implemente o algoritmo de parsing LALR e que possa ser utilizado com C#, o único que preenche estes dois requisitos é o sistema GOLD.

O facto de esta ferramenta também permitir testar a gramática foi muito útil durante o processo de elaboração da gramática do OCL. Utilizou-se também a funcionalidade que permitia a criação do esqueleto do parser e de acordo com o *Engine* utilizado. O *State Broser* auxiliou na identificação de problemas e erros da gramática.

Tabela 7 - Mapa de comparação entre o GOLD Parser system e outros sistemas idênticos

Comparison Chart					
Feature	GOLD	ANTLR	Grammatica	Spirit	YACC / Bison
License	Free	Free	Free	Free	Free
Parsing Algorithm	LALR	LL	LL(1)	LALR	LALR
Grammar Notation	BNF	EBNF	EBNF	EBNF	BNF
Grammar / Code	Independent	Mixed	Mixed	Mixed	Mixed
Parser Source Code	✓	✓	✓	✓	✓
Create Skeleton Programs	✓	✓	✓	✓	✓
Integrated Design Environment	GOLD	ANTLR(2)	Grammatica	Spirit	YACC / Bison
Integrated Testing	✓	✓			
State Browsing	✓	✓			
Generate Webpages	✓	?			
Export to XML	✓	?			
Export to Formatted Text	✓	?			
Supported Languages	GOLD	ANTLR	Grammatica	Spirit	YACC / Bison
ANSI C	✓				✓
Assembly - Intel x86	✓				
C++	✓	✓	✓	✓	✓
C#	✓	✓	✓		
Delphi 5 & 6	✓				
DigitalMars D	✓				
Java	✓	✓			
Pascal	✓				
Python	✓	✓			
Visual Basic 6	✓				
Visual Basic .NET	✓				
All Other .NET Languages (3)	✓				
All Other ActiveX Languages (4)	✓				

(1) Grammatica only supports LL for now. However, the website claims that they may support LR in the future.
(2) ANTLR has a IDE called ANTLRWorks. I haven't had a chance to use it, but it seems like a great product.
(3) Some Engines are compiled to a .NET module. This allows the componant to be used with any IDE that supports .NET.
(4) Some Engines are compiled to an ActiveX / COM plugin. This allows the componant to be used with any IDE that supports ActiveX / COM.

4.1. LALR Parsing e DFA

Como já vimos o LALR é o algoritmo de *parsing* utilizado e o DFA é utilizado para análise lexical. Ambos os algoritmos são máquinas de estado que utilizam tabelas para determinar acções. Vimos também que estas tabelas são produzidas pelo *Builder* e passadas ao *Engine*. É o facto de o sistema estar separado nestas unidades que permite que os *parsers* possam ser implementados em diferentes linguagens de programação e para diferentes plataformas.

No processo de LR parsing, ou *Left-to-right parsing* com *Right-derivation*, são utilizadas tabelas para determinar se uma regra está completa ou se é preciso ler mais *tokens*. Estas tabelas possuem todos os estados possíveis que o sistema pode encontrar durante o reconhecimento do texto. Cada estado representa um ponto no processo de reconhecimento onde um determinado número de *tokens* já foi efectivamente lido do texto original e as regras estão em diferentes estados de conclusão. Chamamos de configuração a cada produção num estado de conclusão e de conjunto de configurações a cada estado de conclusão. Em cada configuração existe um cursor que representa o ponto de conclusão da produção.

LALR Parsing, ou *Lookahead LR parsing*, é baseado no LR Parsing mas com a diferença que o LALR combina conjuntos de configurações relacionados, o que reduz o tamanho da tabela de *parsing* resultante. Contudo o algoritmo é menos poderoso que o algoritmo "pai" mas mais pratico.

Na prática o que acaba por acontecer é que as gramáticas reconhecidas por algoritmos LR podem não o ser pelo algoritmo LALR, embora isto seja raro.

DFA ou *Deterministic Finite Automaton*, como o próprio nome indica é um autómato onde finito significa que tem um número finito e conhecido de estados e transições entre esses estados, e determinístico significa que dado um estado qualquer e um determinado símbolo existe uma e apenas uma transição para o estado seguinte, ou seja não é ambíguo. O algoritmo é responsável por, dado um estado actual e um símbolo de entrada, determinar qual a transição a efectuar e consecutivamente qual o próximo estado.

A análise lexical é o processo de converter uma sequência de caracteres numa sequência de *tokens*. Este mecanismo de reconhecimento de *tokens*, é um dos mecanismos do sistema GOLD, e o seu objectivo é reconhecer diferentes *tokens* e passa-los ao algoritmo de *parsing*.

Expressões regulares podem ser utilizadas para definir linguagens regulares. As linguagens regulares, por sua vez, podem exibir padrões muito simples. Um DFA é um método de reconhecer estes padrões de forma algorítmica.

4.2. EBNF v.s. BNF

Como já vimos anteriormente o GOLD Parsing system utiliza a notação BNF para a definição das regras gramaticais, o que irá obrigar a uma adaptação da gramática do OCL avançada pela OMG por esta estar na notação EBNF. Mais à frente nesta secção vamos abordar em mais detalhe esta adaptação.

A notação BNF ou Backus-Naur Form, desenvolvida por John Backus e Peter Naur, é uma notação utilizada para descrever gramáticas independentes do contexto. As regras de derivação têm o seguinte aspecto:

- regras são representadas por um não terminal (nome da regra), normalmente delimitados por “<…>”,
- o nome da regra é seguido por “::=”,
- cada regra é definida por uma ou mais produções, e
- quando um terminal é definido por mais que uma produção e para melhorar a sua leitura, pode ser utilizado uma barra vertical “|” que tem o significado de “ou”.

Ilustração 15 - Estrutura de uma regra gramatical na notação BNF

No exemplo seguinte é definido uma regra `<Value>` que pode conter um terminal `Identifier` ou uma outra regra que corresponde ao não terminal `<Literal>`.

```
<Value>   ::= Identifier | <Literal>
<Literal> ::= Number | String
```

A regra `<Literal>` por sua vez pode conter um `Number` ou uma `String`. Consequentemente a regra `<Value>` pode conter um `Identifier`, um `Number`, ou uma `String`.

Também pode existir recursividade nas regras

```
<Identifiers> ::= Identifier <Identifiers>
                |  Identifier
```

O EBNF ou Extended BNF, é uma variante do BNF desenvolvida para simplificar a notação original, os elementos já existentes na notação BNF são herdados e são introduzidos novos elementos que permite definir listas e componentes opcionais. Existem alguma variedade de versões do EBNF mas quase todas utilizam a mesma notação:

- a utilização de parênteses rectos “[…]” ou de um ponto de interrogação “?” corresponde a uma componente opcional,
- elementos que surgem zero ou mais vezes ficam entre chavetas “{…}”,
- elementos que se repetem uma ou mais vezes são seguidos por “+”, e
- quando existe a necessidade de agrupar elementos é utilizado parênteses normais “(…)”.

Esta notação quando utilizada cria algumas regras implícitas, como no exemplo seguinte, onde a expressão *if-then-else* é definida de duas maneiras diferentes mas com o mesmo resultado pratico:

Na primeira é na notação EBNF e é utilizado parênteses rectos para indicar que o conjunto `ELSE <Stms>` é opcional.

```
<If Stm> ::= IF <Expression> THEN <Stms> [ ELSE <Stms> ]
```

Da regra anterior resultaria a seguinte regra na notação BNF:

```
<If Stm> ::= IF <Expression> THEN <Stms>
           |  IF <Expression> THEN <Stms> ELSE <Stms>
```

Como vimos no inicio desta secção é necessário remover todos os elementos da notação EBNF da gramática existente na especificação do OCL construindo regras equivalentes na notação BNF. Esta é umas das transformações que a gramática teve de sofrer para passar a ser aceite pelo sistema GOLD.

Quando a forma como as regras gramaticais são construídas é fundamental para o reconhecimento dessas mesmas regras, parece-me prejudicial a utilização de uma notação que produz regras implícitas. O EBNF acaba por não trazer vantagens para este caso específico.

5. MetaSketch OCL Interpreter

O interpretador é composto por um *Parser*; um construtor do Modelo de Sintaxe Abstracta; e uma unidade que gera o código C# a partir do Modelo de Sintaxe Abstracta (ver Ilustração 16).

O Parser recebe informação em forma de expressões OCL e com o auxílio de uma gramática irá reconhecer a sintaxe das expressões. Em seguida irá devolver uma árvore de sintaxe concreta ou CST. Nesta fase a semântica das expressões não é relevante, interessa apenas verificar se a sintaxe utilizada está em conformidade com a sintaxe descrita na gramática. Este tema é aprofundado na secção 5.1.2. O Parser.

O construtor do modelo de sintaxe abstracta recebe a árvore de sintaxe concreta produzida pelo Parser e constrói um modelo de sintaxe abstracta. Cada conceito deste modelo representa um nó da árvore recebida, eventualmente pode não conter todos os nós. A forma como o modelo é construído não reflecte a organização da sintaxe concreta, razão pela qual se chama "modelo de sintaxe abstracta" e muitas vezes é abreviado para ASM (Abstract Syntax Model). A sintaxe concreta é da exclusiva responsabilidade do *Parser*. O construtor do modelo de sintaxe abstracta é abordado em maior detalhe na secção 5.1.3. O Abstract Syntax Model Builder.

O gerador de código recebe o modelo de sintaxe abstracta e a partir da informação contida neste gera o código C# executável. Para isso pode aceder à OCL Standard Library e ao Modelo. O gerador de código é também aprofundado na secção 5.3. Interpretação/Execução das Expressões em OCL

Cada uma das restantes componentes que interagem com o MetaSketch OCL Interpreter, como a OCL Grammar e a OCL Standard Library são aprofundadas ao longo das secções seguintes.

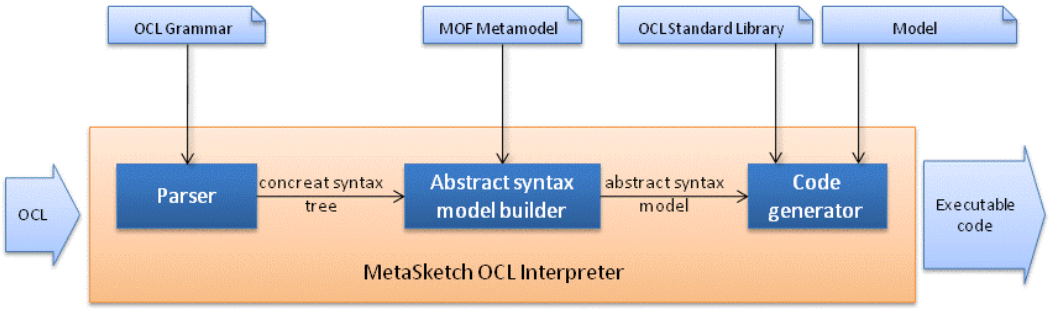


Ilustração 16 - Arquitectura do MetaSketch OCL Interpreter

Para a construção de um interpretador de OCL começamos por definir a gramática necessária para a implementação do *Parser*.

No documento da especificação do OCL 2.0 (OCL 2.0 - OCL 2.0 Specification, Version 2 - formal.2006-05-01) é apresentada a sintaxe abstracta em forma de diagrama de classes (Capitulo 8 da especificação); uma gramática EBNF que descreve a sintaxe

concreta da linguagem (Capitulo 9 da especificação); e o mapeamento entre a sintaxe concreta e a sintaxe abstracta que pode ser encontrado ao longo da gramática (Capitulo 9 da especificação).

A gramática proposta possui algumas propriedades que inviabilizam a sua utilização directa, somos alertados para essa possibilidade no próprio documento.

> "The grammar in this chapter might not prove to be the most efficient way to directly construct a tool. Of course, a tool-builder is free to use a different parsing mechanism. He can, for example, first parse an OCL expression using a special concrete syntax tree, and do the semantic validation against a UML model in a second pass. Also, error correction or syntax directed editing might need hand-optimized grammars. This document does not prescribe any specific parsing approach. The only restriction is that at the end of all processing a tool should be able to produce the same well-formed instance of the abstract syntax, as would be produced by this grammar." [4]

Alem das propriedades da gramática que inviabilizam a sua utilização directa, foram encontrados problemas de duas naturezas diferentes: regras gramaticais ambíguas e algumas inconsistências.

Foi necessário fazer uma extensa adaptação á gramática para que esta fosse aceite pelo sistema de *parsing* utilizado, o *GOLD Parsing system*. Neste capítulo irei apresentar alguns dos problemas identificados durante o processo de adaptação da gramática e possíveis soluções.

5.1. Tratamento da Gramática

Nesta secção vamos ver quais os aspectos mais relevantes na elaboração da gramática equivalente. Quais as alterações que foram feitas à sintaxe concreta que é apresentada pela OMG. Também iremos abordar a componente Parser e a componente Abstract Syntax Model Builder. A componente Code Generator é abodada e detalhada na secção 5.3. Interpretação/Execução das Expressões em OCL.

No Apêndice I dispõem da gramática LALR que foi definida para o sistema GOLD e que é equivalente à gramática específica pela OMG, ou seja, é uma gramática que representa a mesma linguagem. As próximas linhas pretendem explanar as alterações e assumpções mais pertinentes.

5.1.1. A sintaxe concreta

A sintaxe concreta do OCL é descrita no capítulo 9 da especificação através de uma gramática EBNF. Contudo e a sintaxe concreta apresentada no capitulo 9 não está completa, não descreve a sintaxe desde o *packageDeclaration* até chegar ao *OclExpression*, esta informação esta disponível mais à frente no documento da OMG na secção 12.12.

Depois de todos os elementos da sintaxe concreta reunidos foram feitas várias iterações no processo de construção da gramática equivalente. Em cada iteração, a gramática era introduzida no sistema GOLD Parser e compilada, em seguida o relatório da compilação era estudado e possíveis erros ou conflitos eram corrigidos. Quando não existam conflitos graves, a gramática era sujeita a alguns testes a fim de apurar se seria equivalente. O processo foi repetido até obtermos uma gramática equivalente. Note-se que uma gramática sem erros não é sinónimo de gramática equivalente.

Os testes foram realizados com recurso a expressões OCL retiradas do próprio documento da especificação do OCL, e dos documentos XMI da especificação do MOF 2.0 e UML 2.0. Escolheu-se estas expressões por se querer ser de fonte credível, o que conferia algum grau de confiança. Contudo alguma dessa confiança rapidamente se esbateu. Existem alguns problemas quer com as expressões existentes na especificação do OCL, quer nos XMIs do UML e do MOF. Expressões incompletas e alguns erros sintácticos evidenciam a necessidade para a existência de uma ferramenta que auxilie na definição de expressões OCL.

Durante a definição da gramática, alguns dos conflitos que surgiram eram *Reduce-Reduce*. Utilizaram-se várias técnicas para eliminar este tipo de conflitos, uma dela foi a remoção de produções singulares.

Como já foi dito no capítulo anterior, uma das alterações que a gramática sofreu foi a passagem de EBNF para BNF. Mas esta alteração não é suficiente para que a gramática seja aceite pelo sistema GOLD Parser e reproduza a linguagem tal como é pretendido.

Um dos problemas existentes na gramática sugerida está relacionado com o que se assumiu ser uma pequena gafe. A caixa seguinte contém um excerto da gramática descrita na especificação.

```
...
1  CollectionLiteralPartsCS ::= CollectionLiteralPartCS
2                               ( ',' CollectionLiteralPartsCS[2] )?
3
4  CollectionLiteralPartCS ::= CollectionRangeCS
5  CollectionLiteralPartCS ::= OclExpressionCS
6
7  CollectionRangeCS ::= OclExpressionCS ',' OclExpressionCS
...
```

Um *CollectionLiteralParts* pode ser um *CollcetionLiteralPart*, ou uma sequência de CollectionLiteralParts separados por vírgulas, linhas 1 e 2. Um *CollectionLiteralPart* pode ser um *CollcectionRange* ou um *OclExpression*, linhas 4 e 5. Por outro lado, um *CollectionRange* é constituído por dois *OclExpressions* separados por vírgulas.

O problema torna-se evidente quando fazemos:

```
Bag{4,7}
```

É uma colecção do tipo Bag<Integer> que contém os elementos 4 e 7, ou os elementos de 4 a 7, isto é, 4, 5, 6, 7? Não se consegue saber. Não faz qualquer sentido que para o CollectionRange seja utilizada uma vírgula, pois é o mesmo símbolo que é utilizado para separar elementos da Collection, como é mostrado na regra das linhas 1 e 2.

Na página 62 da especificação [4], na introdução do capítulo 9, quando é explicada a estrutura da sintaxe concreta, é utilizado o seguinte excerto da gramática:

```
CollectionRangeCS ::= OclExpressionCS[1] '..' OclExpressionCS[2]
```

Se consultarmos a versão anterior da especificação do OCL [13] a gramática possui esta mesma regra. O que nos levou a pensar que seria um lapso, corrigiu-se a regra e o resultado final foi:

```
...
<CollectionLiteralParts> ::= <CollectionLiteralPart>
                           |  <CollectionLiteralParts> ',' <CollectionLiteralPart>

<CollectionLiteralPart> ::= <CollectionRange>
                          |  <OclExpression>

<CollectionRange> ::= <OclExpression> '..' <OclExpression>
...
```

Outro problema identificado é uma ambiguidade. A caixa que está logo a seguir possui um excerto da sintaxe concreta descrita na especificação.

```
...
1  NavigationCallExpCS ::= AssociationEndCallExpCS
2  NavigationCallExpCS ::= AssociationClassCallExpCS
3
4  AssociationEndCallExpCS ::= OclExpressionCS '.' simpleNameCS
5                              ( '[' argumentsCS ']' )? isMarkedPreCS?
6
7  AssociationClassCallExpCS ::= OclExpressionCS '.' simpleNameCS
8                                ( '[' argumentsCS ']' )? isMarkedPreCS?
...
```

A regra *NavigationCallExp* pode ser ou uma *AssociationEndCallExp* ou uma *AssociationClassCallExp*, linhas 1 e 2. Contudo se repararmos nas produções *AssociationEndCallExp* (linhas 4 e 5) e na *AssociationClassCallExp* (linhas 7 e 8) podemos verificar que são exactamente iguais o que introduz ambiguidade. A sintaxe concreta sugerida pela OMG possui regras de desambiguação. Contudo estas regras não se adequam à realidade da nossa estratégia, estas regras de desambiguação possuem referências a variáveis de ambiente e a artefactos que só se encontram disponíveis no Modelo. Logo, isto obriga a que toda esta informação esteja disponível a quando do reconhecimento sintáctico do texto, o que de acordo com a nossa arquitectura, é impossível (ver Ilustração 16). Delegou-se esta responsabilidade para o *Abstract Sintax Model Builder*.

Para resolver este problema e problemas idênticos optamos por eliminar as produções que estavam duplicadas. A regra equivalente a que se chegou foi:

```
...
1  <PropertyCallExp> ::= <PreAndPath> <PropertyCallExpAux>
2
3  <PropertyCallExpAux> ::= '.' <SimpleName> <PropertyCallExpAux>
4                        |  '.' <SimpleName> <isMarkedPre> <PropertyCallExpAux>
5                        |  '.' <SimpleName> '[' <Arguments> ']' <PropertyCallExpAux>
6                        |  '.' <SimpleName> '[' <Arguments> ']' <isMarkedPre> <PropertyCallExpAux>
7                        |
8
9  <PreAndPath> ::= <SimpleName> <isMarkedPre>
10               |  <PathName>
11
12 <PathName> ::= <SimpleName>
13            |  <PathName> '::' <SimpleName>
...
```

As regras *<PropertyCallExpAux>* e *<PreAndPath>* não estão completas, versão completa destas regras esta no Apêndice I. Este excerto representa apenas a parte da regra que esta relacionada com a *NavigationCallExpCS*.

Como já vimos na secção anterior o sistema GOLD Parser não tem nenhum mecanismo para definir a precedência de operadores. A precedência é definida pela ordem em que as regras são definidas. Contudo a sintaxe concreta definida pela OMG assenta na utilização de um mecanismo separado da sintaxe concreta para definir a precedência de operadores. A regra que define a sintaxe das operações binárias é a que está na seguinte caixa:

```
...
1  OperationCallExpCS ::= OclExpressionCS simpleNameCS OclExpressionCS
...
```

O simpleName representa o operador binário. A precedência está definida na especificação através da seguinte sequência de operadores, sendo o "@pre" o mais prioritário e o "implies" o menos prioritário:

- @pre
- dot and arrow operations: '.' and '->'
- unary 'not' and unary minus '-'
- '*' and '/'
- '+' and binary '-'
- 'if-then-else-endif'
- '<', '>', '<=', '>='
- '=', '<>'
- 'and', 'or', and 'xor'
- 'implies'

A precedência dos operadores não binários, com a excepção do *if-then-else-endif*, não levantou problemas, a sua prioridade já tinha sido implementada em regras dedicadas a cada um desses operadores. O problema estava na regra genérica que tinha sido definida para os operadores binários e no facto de o *if-then-else-endif* se encontrar "no meio da precedência" dos operadores binários, ou seja, o *if-then-else-endif* não tem

prioridade sobre o “+”, “-” ou “*” e “/”, mas tem mais prioridade sobre o “<”, “<=”, “<>” ou “and” e “implies”.

O que se optou por fazer foi criar um conjunto de regras que descreveriam individualmente a sintaxe de cada um destes operadores. O facto de estas regras serem definidas separadamente permite que, à semelhança do que acontece na Tabela 6, seja definida a precedência dos diferentes operadores através da ordem em que eles ocorrem como mostra a caixa seguinte:

```
...
1  <OclExpressionAux> ::= <AndOrXorExp>
2                       |  <AndOrXorExp> 'implies' <OclExpressionAux>
3
4  <AndOrXorExp>   ::= <EqualOrNotExp>
5                   |  <EqualOrNotExp> 'xor' <AndOrXorExp>
6                   |  <EqualOrNotExp> 'or'  <AndOrXorExp>
7                   |  <EqualOrNotExp> 'and' <AndOrXorExp>
8
9  <EqualOrNotExp> ::= <LessOrMoreExp>
10                  |  <LessOrMoreExp> '='  <EqualOrNotExp>
11                  |  <LessOrMoreExp> '<>' <EqualOrNotExp>
12
13 <LessOrMoreExp> ::= <IfExp>
14                  |  <IfExp> '>'  <LessOrMoreExp>
15                  |  <IfExp> '<'  <LessOrMoreExp>
16                  |  <IfExp> '<=' <LessOrMoreExp>
17                  |  <IfExp> '>=' <LessOrMoreExp>
18
19 <IfExp> ::= <AddExp>
20          |  'if' <OclExpression> 'then' <OclExpression> 'else' <OclExpression>
21             'endif'
22
23 <AddExp>     ::= <MultExp>
24               |  <MultExp> '+' <AddExp>
25               |  <MultExp> '-' <AddExp>
26
27 <MultExp>    ::= <NegateAndNotExp>
28               |  <NegateAndNotExp> '*' <MultExp>
29               |  <NegateAndNotExp> '/' <MultExp>
...
```

Porque o algoritmo de *parsing* implementado pelo sistema GOLD Parser é LALR as regras que surgem em primeiro lugar na gramática são as que têm menos prioridade e as que surgem no fim têm mais prioridade. Cada regra tem uma produção singular para que seja possível passar para regra seguinte. A recursão é feita à direita o que tem implicações na associatividade dos operadores, o facto de a recursão ser à direita faz com que a que as operações reconhecidas sejam associativas à direita.

5.1.2. O Parser

O Parser é um dos três módulos que constituem o MetaSketch OCL Interpreter, este módulo é responsável por reconhecer o texto que contem as expressões OCL, e construir uma árvore de sintaxe concreta que posteriormente é passada ao Abstract Sintax Model Builder. Este módulo utiliza um dos Engines que existem disponíveis no site oficial do sistema GOLD Parser. Como este projecto é todo implementado em C# foi escolhido um Engine também implementado em C#. Optamos por utilizar o Morozov C# Engine que foi desenvolvido por Vladimir Morozov. No Apêndice J é possível consultar a documentação do Engine utilizado.

Como já foi anteriormente afirmado (na secção 4.) o papel do Engine é receber o texto a ser reconhecido, e com o auxílio da gramática pré-compilada, produzir uma sequência de tokens. É com esta sequência de tokens que o Parser constrói a árvore de sintaxe concreta. Cada token reconhecido corresponde a um nó da árvore. Cada um destes nós possuem, alem da informação sobre token, informação de qual foi a regra utilizada para o reconhecido desse token.

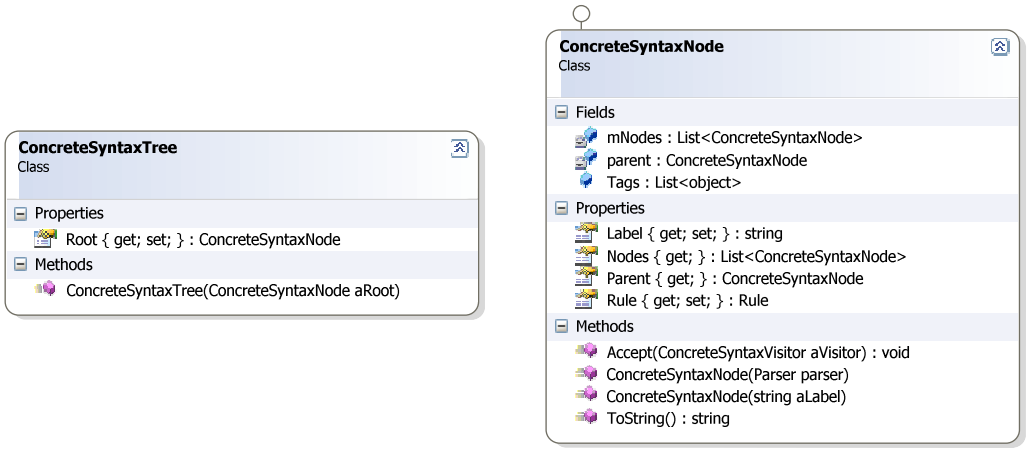


Ilustração 17 - Diagrama de classe, estrutura da árvore de sintaxe concreta

O diagrama anterior representa a estrutura utilizada para construir a árvore de sintaxe concreta. O *ConcreateSyntaxTree* é a classe que contém a raiz da árvore. O *ConcreateSyntaxNode* é a classe que representa os nós da árvore, contem um *Lable* que é o token reconhecido, contem uma *Rule* que é a regra utilizada no reconhecimento do token e contem também um *Parent* e um *Nodes* que representam, respectivamente, uma referência ao nó pai e uma referência aos nós filhos. Esta classe pode ser instanciada passando o *Parser* ou uma *Lable* ao seu construtor. O Parser possui toda a sequência de tokens reconhecidos e pode ser percorrido.

É utilizado o padrão de desenho *Visitor* nesta árvore, razão pela qual o *ConcreateSyntaxNode* possui um método *Accept* que recebe como argumento o "visitante". Caso exista a necessidade de registar alguma informação relativa à visita, o "visitante" pode utilizar o atributo *Tags*. Este padrão de desenho foi implementado para permitir que o Abstract Syntax Model Builder possa percorrer mais facilmente a árvore de sintaxe concreta, isso é conseguido ao "estender" o *ConcreateSyntaxNode* com todos os métodos necessários para o correcto reconhecimento da árvore. No Apêndice C possui mais informação sobre o padrão de desenho *Visitor*.

5.1.3. O Abstract Syntax Model Builder

O Abstract Syntax Model Builder é a segunda componente do MetaSketch OCL Interpreter, esta componente é o "visitante" da árvore de sintaxe concreta, isto é, implementa o *Visitor* da árvore de sintaxe concreta. Este é responsável por percorrer toda a árvore de sintaxe concreta, quando "visita" um nó, o ASM Builder verifica qual foi a regra utilizada no reconhecimento do token do nó visitado e instancia o elemento

do modelo de sintaxe abstracta que se adequa a essa regra. Nem todos os nós são traduzidos em elementos do modelo, nós que não tenham significado como os que correspondem a produções singulares não têm qualquer correspondência no ASM.

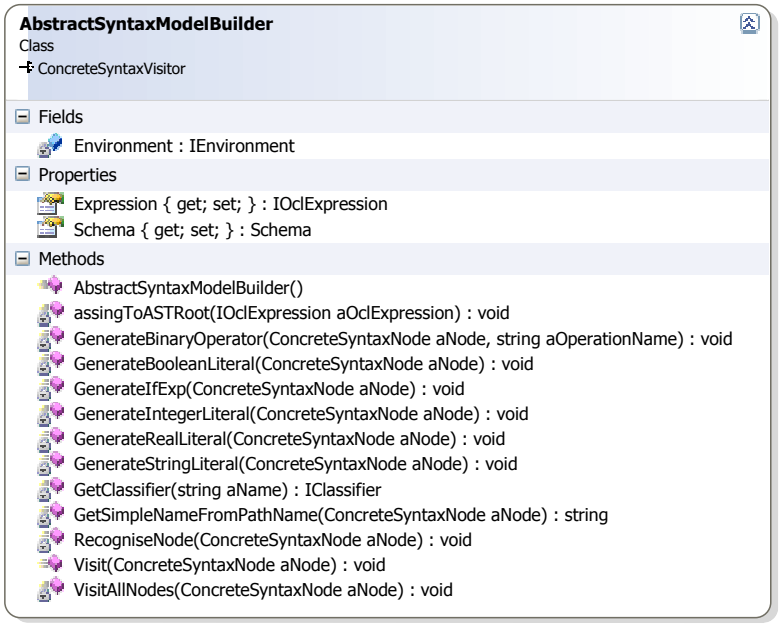


Ilustração 18 - Diagrama de classe, Abstract Syntax Model Builder

A implementação do ASM Builder possui duas propriedades, o *Expression* que é onde fica guardado o modelo da expressão OCL e o *Schema* que contem a informação do contexto sobre o qual a expressão OCL irá ser aplicada. O *Environment* é uma representação do ambiente e pretende conter toda a informação sobre o ambiente de interpretação da expressão reconhecida. Ou seja, o *Environment* guarda as variáveis que eventualmente são declaradas ou utilizadas num *iterator*, para que posteriormente possam ser consultadas ou modificadas sempre que for solicitado durante a interpretação da expressão.

Sendo o ASM Builder a implementação do *Visitor* da CST possui um método *Visit* que recebe como argumento o *ConcreteSyntaxNode* que é visitado. Este método usufrui de vários outros auxiliares que ajudam a manter o código mais fácil de ler e compreender.

É no método *Visit* (e seus auxiliares) que toda a lógica necessária para identificar qual o elemento do modelo que é necessário instanciar é implementada. Essa lógica inclui as verificações necessárias para identificar qual a regra gramatical utilizada no reconhecimento do token. Após a regra ter sido identificada, o *Visit* terá que instanciar o elemento da ASM que melhor se adequa passando a informação que já sabe estar disponível nos nós filhos ou eventualmente no nó pai.

Esta estratégia foi inspirada numa estratégia apresentada por Birgit Demuth, Heinrich Hussmann e Ansgar Konermann em “Generation of an OCL 2.0 Parser” [18].

5.2. Implementação da OCL Standard Library em C#

A OCL Standard Library é uma biblioteca que contem todos os tipos predefinidos utilizados no OCL e todas as operações possíveis em cada um desses tipos. Sendo o OCL uma linguagem tipificada, está biblioteca assume assim uma grande importância.

No capítulo 11 da especificação do OCL podemos encontrar toda a informação relacionada com o conteúdo da biblioteca. Como é óbvio não irei transcrever a especificação para este documento, esta secção deve ser acompanhada de uma leitura do capítulo 11 da especificação.

A implementação da OCL Standard Library aqui apresentada assenta nos tipos primitivos do C# e na .NET Framework 3.5. Para os tipos relacionados com colecções foi utilizada a interface extensível e implementável IEnumerable<T>, deste modo as colecções herdam propriedades desta interface. Também foi utilizado a List<T> para aproveitar ao máximo as capacidades existentes na .NET Framework no que diz respeito a colecções. Quando não foi possível recorrer a métodos já existentes, ou porque não se comportavam como pretendido ou por estes não existirem, foram definidos novos métodos. Neste processo foi dada especial atenção à comparação de colecções, pois o método utilizado para a comparação é depois utilizado em vários outros métodos.

A OMG decompõem a Standard Library em quatro subconjuntos de tipos, os tipos primitivos, as colecções, os tipos especiais e os tipos restantes que não tem nenhuma classificação específica.

Tabela 8 - Mapeamento entre os tipos OCL e os tipos C#

<table>
<tr><th></th><th>Tipo OCL</th><th colspan="2">Tipo C# (.NET Framework type/library)</th></tr>
<tr><td rowspan="4">Tipos primitivos</td><td>Integer</td><td colspan="2">int (System.Int32)</td></tr>
<tr><td>Real</td><td colspan="2">double (System.Double)</td></tr>
<tr><td>String</td><td colspan="2">string (System.String)</td></tr>
<tr><td>Boolean</td><td colspan="2">bool (System.Boolean)</td></tr>
<tr><td rowspan="5">Colecções</td><td>Collection</td><td rowspan="5">Implementa a interface System.Collections. Generic.IEnumerable<T></td><td>OclCollection<T></td></tr>
<tr><td>Bag</td><td>OclBag<T></td></tr>
<tr><td>Sequence</td><td>OclSequence<T></td></tr>
<tr><td>Set</td><td>OclSet<T></td></tr>
<tr><td>OrderedSet</td><td>OclOrderedSet<T></td></tr>
<tr><td rowspan="2">Tipos especiais</td><td>OclElement</td><td colspan="2"></td></tr>
<tr><td>OclType</td><td colspan="2">Class (System.Class)</td></tr>
<tr><td rowspan="4"></td><td>OclAny</td><td colspan="2">Object (System.Object)</td></tr>
<tr><td>OclVoid</td><td colspan="2"></td></tr>
<tr><td>OclInvalid</td><td colspan="2">Null</td></tr>
<tr><td>OclMessage</td><td colspan="2"></td></tr>
</table>

5.2.1. Tipos Primitivos

Como não é possível a extensão de tipos primitivos em C#, foi criado um conjunto de classes estáticas que contêm as operações permitidas sobre cada um dos tipos. Note-se que estas classes não implementam nenhum dos tipos primitivos, contêm apenas os métodos que são possíveis utilizar em qualquer um dos tipos primitivos do OCL (ver Ilustração 19). Tal como as classes todos os métodos são estáticos. Os métodos não se encontram na notação infixa como os métodos originais, foram implementados na notação prefixa, o que implica que algo que em OCL seria 3+5 seja traduzido para OclInteger.plus(3,5).

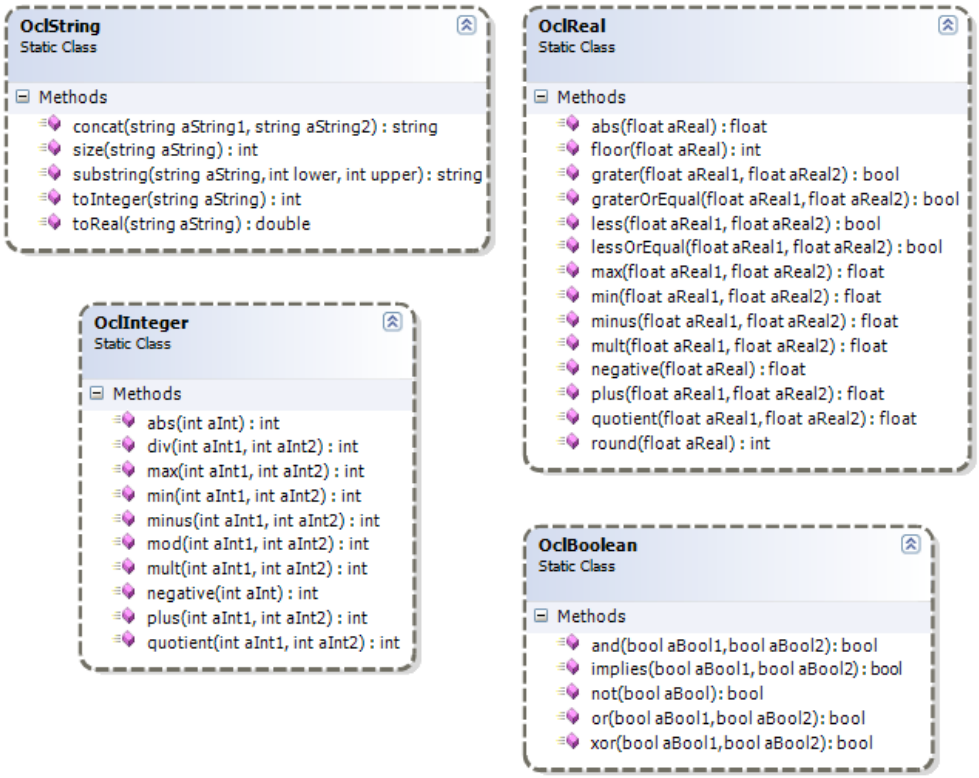


Ilustração 19 - Classes que contêm as implementações dos métodos permitidos sobre os tipos primitivos do OCL

Algumas das operações permitidas sobre os tipos primitivos são representadas na especificação através de símbolos como por exemplo o *, ou +. Foi feita uma tradução para outras representações. Também é dito na especificação que o tipo Integer é subclasse de Real com o objectivo de indicar que o tipo Integer é compatível com o tipo Real, o que na prática permite efectuar operações como 5.3*7; sendo 5.3 do tipo Real e 7 do tipo Integer. Contudo os tipos int e double existentes no C# já prevêem este comportamento. Tendo em mente que as classes OclReal, OclInteger, OclString e OclBoolean são classes estáticas que contêm apenas os métodos permitidos em cada um dos tipos correspondentes (ver Ilustração 19), resta apenas o motivo da herança de funções para que a classe estática OclInteger seja subclasse de OclReal. Dito isto, os únicos métodos que fariam sentido herdar de Real (OclReal) seriam: “<”, “>”, “<=” e “>=”. Contudo estes métodos teriam que ser redefinidos, uma vez que os argumentos

aceites têm que ser do tipo int e não do tipo double. Optamos por não implementar esta extensão, por acharmos que não traz nenhuma vantagem, ficamos então com as duas classes separadas.

Tabela 9 - Mapeamento de operações sobre tipos primitivos

Tipo	Representação do operador na especificação		Operador implementado
Real	+ (r : Real) : Real		plus(double aReal1, double aReal2) : double
	- (r : Real) : Real		minus(double aReal1, double aReal2) : double
	* (r : Real) : Real		mult(double aReal1, double aReal2) : double
	- : Real The negative value of self.		negative(double aReal) : double
	/ (r : Real) : Real		quotient(double aReal1, double aReal2) : double
	abs() : Real		abs(double aReal) : double
	floor() : Integer		floor(double aReal) : int
	round() : Integer		round(double aReal) : int
	max(r : Real) : Real		max(double aReal1, double aReal2) : double
	min(r : Real) : Real		min(double aReal1, double aReal2) : double
	< (r : Real) : Boolean		less(double aReal1, double aReal2) : bool
	> (r : Real) : Boolean		grater(double aReal1, double aReal2) : bool
	<= (r : Real) : Boolean		lessOrEqual(double aReal1, double aReal2) : bool
	>= (r : Real) : Boolean		graterOrEqual(double aReal1, double aReal2) : bool
Integer	- : Integer		negative(int aInt) : int
	+ (i : Integer) : Integer		plus(int aInt1, int aInt2) : int
	- (i : Integer) : Integer		minus(int aInt1, int aInt2) : int
	* (i : Integer) : Integer		mult(int aInt1, int aInt2) : int
	/ (i : Integer) : Real		quotient(int aInt1, int aInt2) : int
	abs() : Integer		abs(int aInt) : int
	div(i : Integer) : Integer		div(int aInt1, int aInt2) : int
	mod(i : Integer) : Integer		mod(int aInt1, int aInt2) : int
	max(i : Integer) : Integer		max(int aInt1, int aInt2) : int
	min(i : Integer) : Integer		min(int aInt1, int aInt2) : int
	Operadores herdados do tipo Real	< (r : Real) : Boolean	less(int aInt1, int aInt2) : bool
		> (r : Real) : Boolean	grater(int aInt1, int aInt2) : bool
		<= (r : Real) : Boolean	lessOrEqual(int aInt1, int aInt2) : bool
		>= (r : Real) : Boolean	graterOrEqual(int aInt1, int aInt2) : bool
String	size() : Integer		size(string aString) : int
	concat(s : String) : String		concat(string aString1, string aString2) : string
	substring(lower : Integer, upper : Integer) : String		substring(string aString, int lower, int upper) : string
	toInteger() : Integer		toInteger(string aString) : int
	toReal() : Real		toReal(string aString) : double
Boolean	or (b : Boolean) : Boolean		or(bool aBool1, bool aBool2) : bool
	xor (b : Boolean) : Boolean		xor(bool aBool1, bool aBool2) : bool
	and (b : Boolean) : Boolean		and(bool aBool1, bool aBool2) : bool
	not : Boolean		not(bool aBool) : bool
	implies (b : Boolean) : Boolean		implies(bool aBool1, bool aBool2) : bool

5.2.2. Tipos de Colecções

Na especificação da OCL Standard Library é dito que Collection é o supertipo abstracto de todos os tipos. Isto é, os tipos de colecções Bag, Sequence e Set são subclasses de Collection e embora não seja explicitamente dito o OrderedSet é subclasse de Set (ver Ilustração 20). Na implementação adoptada as classes OclBag<T>, OclSequence<T>, OclSet<T> e OclOrderedSet<T> implementam efectivamente os tipos de colecções

correspondentes, o contrário do que acontece no caso dos tipos primitivos onde são utilizados classes abstractas. A implementação de cada um dos tipos de colecção possui dois tipos de operações: iteradores e.g. select, exist; e os não iteradores e.g. size, union. É possível saber quais são os iteradores existentes consultando a Tabela 10, todos os métodos das colecções que não constem desta tabela não são iteradores.

Os tipos de colecções são definidos com recurso a um *template type*. O *template type* é suportado pelo C# da mesma forma como é utilizado na especificação desta biblioteca. Onde um tipo concreto de uma colecção é criado substituindo o parâmetro T do template pelo tipo pretendido, ou seja, Set(Integer) e Bag(Person) são tipos de Collection, nesta implementação da OCL Standard Library, os tipos equivalentes seriam OclSet<Integer> e OclBag<Person>.

Cada tipo de colecção tem características diferentes, alguns tipos podem admitir a repetição de elementos, alguns tipos podem ter a noção de ordem. Um tipo de colecção que permita a existência de elementos repetidos permite {3,5,1,9,1}. Num tipo de colecção que seja ordenado {2,6,4,1} é diferente {6,1,2,4}.

Características dos diferentes tipos de colecção:

- **Set** é o conjunto matemático, não admite a ocorrência de elementos repetidos.
- **OrderedSet** é um conjunto ordenado, e tal como o seu supertipo não admite elementos repetidos.
- **Bag** admite a existência de elementos duplicados ou repetidos, isto quer dizer que um objecto pode ser várias vezes elemento de Bag, não possui qualquer noção de ordem.
- **Sequence** possui a noção de ordem, e admite a existência de elementos repetidos.

Ilustração 20 - Diagrama de classes da hierarquia dos diferentes tipos de colecções

Todas as operações sobre colecções, iteradores e não iteradores, não têm qualquer efeito sobre as colecções onde são aplicadas. Isto é, se uma colecção “Bag{2,4,7}” com o nome “saco” onde seja aplicada a operação “excludes”, por exemplo: “saco->excludes(4)”, o resultado seria “Bag{2,7}”, contudo o “saco” continua igual a “Bag{2,4,7}”. Lembrem-se que a avaliação de expressões OCL não tem qualquer efeito sobre o estado do sistema modelado.

Se consultarmos a especificação podemos constatar que não existe nenhum método flatten no tipo Collection, porem, devido à decisão de utilizarmos o flatten na

implementação do método collect, tal como também é sugerido na restrição do método collect descrito na citação seguinte, somos obrigados a incluir o flatten no Collection.

> "collect
>
> The Collection of elements that results from applying body to every member of the source set. The result is flattened. Notice that this is based on collectNested, which can be of different type depending on the type of source. collectNested is defined individually for each subclass of CollectionType.
>
> `source->collect(iterators | body) = source->collectNested (iterators | body)->flatten()`" [4]

De entre os diferentes métodos existentes nas colecções os que se revelaram ser um desafio na sua implementação foram: product, isUnique, equal, including, excluding, flatten, select, reject, collectNested, sortedBy, sum, append, prepend, insertAt, complement e symmetricDifference. Os restantes métodos são mais triviais e em alguns casos apenas foi feito um mapeamento para um método já existente em List<T>.

Tabela 10 - Métodos iteradores dos tipos de colecções em OCL

Tipos de colecções	Métodos iteradores
OclCollection<T>	exists(Predicate<T> aBody) : bool
	forAll(Predicate<T> aBody) : bool
	isUnique(Predicate<T> aBody) : bool
	any(Predicate<T> aBody) : T
	one(Predicate<T> aBody) : bool
	collect(Predicate<T> aBody) : OclCollection<object>
OclSet<T>	select(Predicate<T> aBody) : OclSet<T>
	reject(Predicate<T> aBody) : OclSet<T>
	collectNested(Body<T> aBody) : OclCollection<object>
	sortedBy(Body<T> aBody) : OclSet<T>
OclBag<T>	select(Predicate<T> aBody) : OclBag<T>
	reject(Predicate<T> aBody) : OclBag<T>
	collectNested(Body<T> aBody) : OclCollection<object>
	sortedBy(Body<T> aBody) : OclBag<T>
OclSequence<T>	select(Predicate<T> aBody) : OclSequence <T>
	reject(Predicate<T> aBody) : OclSequence <T>
	collectNested(Body<T> aBody) : OclCollection<object>
	sortedBy(Body<T> aBody) : OclSequence<T>

Alguns iteradores recebem como argumento um Predicate<T>, este tipo de argumento é na verdade um método, um *delegate*, que define um determinado critério e verifica se um objecto satisfaz esse critério ou não. Este *delegate* está disponível no namespace System da .NET Framework. Um possível exemplo da sua utilização seria:

O segmento da expressão OCL tem como objectivo verificar se existe no Bag “a” algum elemento do tipo string que possua apenas dois caracteres, se existir deverá devolver true.

```
...
a = Bag{‘Universidade’, ‘da’, ‘Madeira’}

a.exists( s : String | s.size() = 2 )
...
```

Codiogo C# equivalente:

```
...
String [] text = {“Universidade”, “da”, “Madeira”}
Bag<String> a = Factory.Bag( text );

bool functionS(string s) = { OclInteger.equal(OclString.size(s), 2); };

a.exists(functionS);
...
```

O Predicate<T> permite que seja passado como argumento para a função exists uma função (functionS), que recebe como argumento algo do mesmo tipo que os elementos da colecção (string) e que implementa a verificação pretendida (s.size == 2). A função exists irá aplicar a função que recebeu (functionS) a cada um dos seus elementos. Se em alguma aplicação obtiver true, então o resultado de exists irá ser true. O resultado da avaliação de um Predicate<T> é sempre um booleano.

O Body<T> é semelhante ao Predicate<T>, é também um *delegate*, contudo o resultado da avaliação de um Body<T> pode ser de qualquer tipo desde que comparável, e.g. int, real. Para melhor percebermos a utilização do Body<T> possuímos o seguinte exemplo:

O objectivo do excerto seguinte é obter uma sequência com os elementos do Bag “a” ordenada por ordem crescente de número de caracteres. O resultado esperado é “Sequence(‘da’, ‘Madeira’, ‘Universidade’)”.

```
...
a : Bag(String) = Bag{‘Universidade’, ‘da’, ‘Madeira’}

a.sortedBy( s : String | s.size() )
...
```

Codigo C# eqivalente:

```
...
String [] text = {“Universidade”, “da”, “Madeira”}
Bag<String> a = Factory.Bag( text );

int functionT(string s) = { OclString.size(s); }
a.sortedBy(functionT);
...
```

Por se tratar do sortedBy, o retorno da functionT não pode ser de qualquer tipo, tem que ser obrigatoriamente de um tipo para o qual a operação “<” esteja definida.

```
public delegate IComparable Body<T>(T aObject);
```

O *delegate* acima definido estabelece a estrutura do Body, diz que um Body recebe como argumento um objecto do mesmo tipo que os elementos da colecção onde irá ser utilizado. O IComparable é uma interface que é utilizada por tipos cujos valores podem ser ordenados, como por exemplo os tipos numéricos. É esta interface que irá garantir que os valores retornados por Body podem ser comparados para ordenação.

No site de apoio à programação em ambientes .NET, "msdn", é possível encontrar mais informação sobre este delegate e o Predicate<T> que complementa aquilo que já se falou anteriormente, no apêndice A e D disponibilizo essa informação.

Vamos ver agora em mais detalhe os métodos das colecções que exigem uma implementação mais elaborada, iremos ver individualmente quais as restrições existentes sobre cada método e qual a estratégia adoptada na implementação para responder a essas restrições. Em alguns casos existem varias implementações diferentes do mesmo método, uma para cada tipo de colecção.

Para aproveitarmos ao máximo o que a .NET Framework disponibiliza, em alguns casos foi feito um mero mapeamento de métodos das colecções para métodos já existentes no tipo List<T>. Nos casos mais elaborados onde esse mapeamento não é possível, houve um esforço no sentido de utilizar ao máximo os métodos de List<T> sempre que se achou pertinente e adequado. Contudo não iremos entrar em detalhe nas situações onde o mapeamento é evidente e de trivial implementação.

O **product** é definido na classe OclCollection e o resultado da sua aplicação é o produto cartesiano de duas colecções.

```
post: result = self->iterate (e1; acc: Set(Tuple(first: T, second: T2)) = Set{} |
                c2->iterate (e2; acc2: Set(Tuple(first: T, second: T2)) = acc |
                        acc2->including (Tuple{first = e1, second = e2}) ) )
```

Para ajudar a perceber a restrição dispomos da definição de produto cartesiano existente na Wikipedia.

> "Na Matemática, dados dois conjuntos X e Y, o produto cartesiano (ou produto direto) dos dois conjuntos (escrito como X × Y) é o conjunto de todos os pares ordenados cujo primeiro elemento pertence a X e o segundo, a Y.
>
> $$X \times Y = \{(x, y) \mid x \in X \ \wedge \ y \in Y\}.$$"...
>
> ..."Por exemplo, se o conjunto X é o dos treze elementos do baralho inglês X = {A,K,Q,J,10,9,8,7,6,5,4,3,2}
>
> e o Y é o dos quatro naipes: Y = {♠, ♥, ♦, ♣}
>
> então o produto cartesiano desses dois conjuntos será o conjunto com as 52 cartas do baralho:
>
> X × Y = {(A, ♠), (K, ♠), ..., (2, ♠), (A, ♥), ..., (3, ♣), (2, ♣)}." [5]

Neste caso se seguirmos a estratégia apresentada na restrição e com a definição de produto cartesiano em mente, a implementação é relativamente trivial. Não obstante, a utilização de Tuple não é uma alternativa, por não existir na Standard Library nenhuma definição explícita para este tipo e por não existir na .NET Framework nenhum tipo equivalente. O tipo que mais se aproxima do Tuple é o Dictionary, chegou-se a utilizar o List<T> por ser uma estrutura um pouco mais simples mas verificou-se não ser a melhor solução visto que obriga a que os elementos da lista sejam todos do mesmo tipo. Portanto adoptamos por utilizar o Dictionary por não ter esta restrição. O tipo de retorno do método fica então na seguinte forma: OclCollection<IDictionary<T,U>> onde o T representa o tipo dos elementos da colecção fonte e U representa o tipo dos elementos da colecção passada com argumento do método, como por exemplo: Bag(T).product(Set(U)).

O **equal** é um operador definido individualmente para o Set, Bag e Sequence por possuir uma implementação diferente para cada um destes tipos. Na especificação da Standard Library apresentada pela OMG a função é representada pelo sinal de igual '=', contudo optamos por não utilizar esta notação porque este operador tem uma outra utilização na linguagem C#. O operador '=' é utilizado em C# para fazer a atribuição de valores ou instancias a variáveis.

A definição para a colecção Set deve retornar true se self e s contiverem os mesmos elementos.

```
post: result = (self->forAll(elem | s->includes(elem)) and
                                    s->forAll(elem | self->includes(elem)) )
```

No caso do Bag deve devolver true se self e bag contiverem os mesmos elementos o mesmo numero de vezes.

```
post: result = (self->forAll(elem | self->count(elem) = bag->count(elem)) and
                bag->forAll(elem | bag->count(elem) = self->count(elem)) )
```

O equal no OclSequence retorna true se self contiver os mesmos elementos de s pela mesma ordem.

```
post: result = Sequence{1..self->size()}->forAll(index : Integer |
                                    self->at(index) = s->at(index))
                                    and
                                    self->size() = s->size()
```

O que estas restrições não esclarecem é se é legitimo comparar colecções de diferentes tipos. Uma das características do OCL é não permitir a comparação directa de tipos diferentes não existindo nenhuma razão para que neste caso façamos de modo diferente. Consequentemente as diferentes implementações só admitem como argumento colecções do mesmo tipo, ou seja, não são admitidas situações como: Bag{3,8,1}.equal(Set{3,8,1}).

Achou-se útil utilizar este método para ajudar a redefinir a função Equals presente em todas as classes C# que é herdada de System.Object. Esta redefinição permite que este

método possa ser utilizado na implementação de outros métodos, ou que os métodos que internamente utilizem o Equals possam fazer esta comparação e obter os resultados pretendidos.

Outro aspecto que nos chamou à atenção foi o facto de não existir uma especificação do método equal para o tipo OrderedSet, a implementação do equal herdada do seu supertipo Set é manifestamente desajustada, este ultimo não possuir noção de ordem ao contrário do seu subtipo, que possui a noção de ordem e tal como o Sequence precisa de uma implementação dedicada. Logo achou-se por bem adoptar uma implementação análoga à utilizada no Sequnce, mas em vez de implementarmos um método equal redefinimos o Equals.

O **including**, possui também uma implementação diferente para cada um dos tipos de colecções, é definido no OclSet, OclBag e no OclSequence. Estas diferentes implementações devem-se ao facto de envolverem verificações diferentes para cada um dos tipos de classes e também porque cada uma devolve um resultado de tipo diferente.

No OclSet retorna um Set com todos os elementos de self mais o object.

```
post: result->forAll(elem | self->includes(elem) or (elem = object))
post: self->forAll(elem | result->includes(elem))
post: result->includes(object)
```

No OclBag devolve um Bag que inclui todos os elementos de self mais o object.

```
post: result->forAll(elem |
        if elem = object then
           result->count(elem) = self->count(elem) + 1
        else
           result->count(elem) = self->count(elem)
        endif)
post: self->forAll(elem |
        if elem = object then
           result->count(elem) = self->count(elem) + 1
        else
           result->count(elem) = self->count(elem)
        endif)
```

No OclSequence retorna uma Sequence com todos os elementos de self mais o object.

```
post: result = self.append(object)
```

Porque Bag e Sequence admitem repetições a adição de um elemento à colecção é feito de uma forma mais despreocupada, no entanto, em relação ao Set já não podemos adicionar elementos sem que saibamos se esse elemento já existe na colecção inicial. Caso exista o resultado ira ser a própria colecção inicial tal como é descrito nas restrições referentes ao Set.

As pos-condições para o caso do Bag, pretendem garantir que o número de ocorrências do elemento introduzido seja devidamente calculado após a sua introdução.

O método add é utilizado em cada um dos casos, este método é disponibilizado pela classe List<T>, a única diferença é que para o Set é verificado se o elemento já faz parte da colecção.

O **excluding** possui uma implementação diferente para cada um dos tipos Set, Bag e Sequence e pode ser justificado da mesma forma que justificamos as diferentes implementações do including.

Quando aplicado a um Set deverá devolver um Set com todos os elementos de self excluindo o object.

```
post: result->forAll(elem | self->includes(elem) and (elem <> object))
post: self-  >forAll(elem | result->includes(elem) = (object <> elem))
post: result->excludes(object)
```

Para o Bag deve retornar um Bag com todos os elementos de self excluindo todas a ocorrências de object.

```
post: result->forAll(elem |
        if elem = object then
           result->count(elem) = 0
        else
           result->count(elem) = self->count(elem)
        endif)
post: self->forAll(elem |
        if elem = object then
           result->count(elem) = 0
        else
      result->count(elem) = self->count(elem)
   endif)
```

No Sequence o método excluding retorna uma Sequence que contém todos os elementos de self com a excepção de todas as ocorrências de object. A ordem dos restantes elementos não é alterada.

```
post:result->includes(object) = false
post: result->size() = self->size() - self->count(object)
post: result = self->iterate(elem; acc : Sequence(T)
     = Sequence{}|
         if elem = object then acc else acc->append(elem) endif )
```

As diferentes implementações deste método são todas definidas com recurso ao método excludes que é um método definido no OclCollection<T>. Como o Set não admite repetição é garantido que um elemento só ocorre uma vez, por esta razão é utilizado o Remove, um método disponível em List<T>, que irá remover a única ocorrência do elemento pretendido. Como o Bag e Sequence admitem repetição foi adoptada uma estratégia diferente. Foi utilizado o RemoveAll que também é um método de List<T> que remove todos os elementos que verifiquem uma determinada condição, como neste caso a nossa intenção é remover todas ocorrências de um determinado objecto é passado como argumento do RemoveAll a condição que verifica se um elemento é igual ao elemento pretendido. No exemplo seguinte é mostrado o código C# utilizado na implementação do excluding do OclSequence.

```
public OclSequence<T> excluding( T aObject )
{
    if ( this.excludes( aObject ) ) return this;

    List<T> elements = new List<T>( this );
    elements.RemoveAll( delegate( T item ) { return aObject.Equals( item ); } );

    return new OclSequenceImpl<T>( elements );
}
```

O **flatten** é implementado no Set, Bag e no Sequence devido aos diferentes tipos de retorno. Quando aplicado a um Set retorna um Set, isto é análogo para os restantes tipos. No OclCollection este método é abstracto.

No Set se os elementos não forem de algum tipo de colecção, então o resultado irá ser o mesmo Set. Se os elementos forem de algum tipo de colecção, então o resultado é um Set com todos os elementos dos elementos. Ou seja se self for do tipo Set(Bag(Integer)) o resultado irá ser do tipo Set(Integer).

```
post: result = if self.type.elementType.oclIsKindOf(CollectionType) then
                   self->iterate(c; acc : Set() = Set{} |
                        acc->union(c->asSet() ) )
               else
                   self
               endif
```

Esta pos-condição é utilizada para descrever o flatten em todos os tipos de colecção.

A implementação deste método segue a mesma estratégia para todos os tipos de colecções. Contudo no caso do Set é necessário verificar se o elemento que estamos a adicionar ao Set final já ocorre ou não. Esta verificação tem que ser feita porque o Set não admite repetições e o método union utilizado na pos-condição não é possui qualquer verificação deste género. Se consultarmos o union temos a seguinte descrição.

```
post: result->forAll(elem | result->count(elem) = self->count(elem) + bag-
>count(elem))
post: self->forAll(elem | result->includes(elem))
post: bag ->forAll(elem | result->includes(elem))
```

O que nos parece ser sinceramente desajustada às características do Set. A solução passou por implementar o flatten da seguinte maneira:

```
public override OclCollection<T> flatten()
{
    if (this is OclSet<OclCollection<T>>)
    {
        List<T> result = new List<T>();
        foreach ( IEnumerable<T> collItem in this )
        {
            foreach ( T itemOfcollItem in collItem )
            {
                if ( result.Contains( itemOfcollItem ) ) continue;

                result.Add( itemOfcollItem );
            }
        }
        return new OclSetImpl<T>(result);
    }
    return this;
}
```

Desta forma é utilizado o método existente em List<T> para verificar se o elemento já existe ou não, e só é adicionado caso não exista.

O **select** é um iterador que é definido no Set, Bag e Sequence por ter um tipo de retorno diferente para cada um dos tipos de classes. Quando aplicado a um Set este irá devolver um Set, para o Bag e o Sequence o raciocínio é análogo.

Quando aplicado a um Set devolve um Set que representa um subconjunto de elementos de Set para os quais o body devolve true. Podemos facilmente generalizar para os outros tipos de colecções.

```
source->select(iterator | body) =
         source->iterate(iterator; result : Set(T) = Set{} |
                         if body then result->including(iterator)
                                 else result
                          endif)
```

Para implementar o select foi utilizado o Predicate<T> para o body, o Predicate como já foi descrito anteriormente retorna um booleano. O elemento será incluído na colecção resultante se a aplicação do Predicate der true, a solução encontrada foi a seguinte:

```
public OclSequence<T> select(Predicate<T> aBody)
{
    List<T> result = new List<T>();
    foreach (T item in this)
    {
        if (aBody.Invoke(item)) result.Add(item);
    }
    return new OclSequenceImpl<T>(result);
}
```

Tal como noutros métodos iteradores o iterate presente na expressão OCL é traduzido para o foreach em C#. O Invoke é forma como podemos aplicar o Predicate ao elemento, o resultado irá ser sempre do tipo boolean. A implementação é análoga para os restantes tipos.

O **reject** é um iterador que é implementado no Set, Bag e Sequence por ter um tipo de retorno diferente para cada um dos tipos de colecções. Quando aplicado a um Set este irá devolver um Set. Para o Bag e o Sequence o raciocínio é análogo.

```
source->reject(iterator | body) =
        source->select(iterator | not body)
```

A implementação do reject é semelhante à do select, alias como é sugerido pela expressão OCL descrita em cima. Assim sendo a implementação é:

```
public OclSequence<T> reject(Predicate<T> aBody)
{
    List<T> result = new List<T>();
    foreach (T item in this)
    {
        if (!aBody.Invoke(item)) result.Add(item);
    }
    return new OclSequenceImpl<T>(result);
}
```

A diferença do reject para o select está precisamente na negação do aBody, tal como na expressão OCL.

O **collectNested** é um iterador que é implementado no Set, Bag e Sequence por retornar tipos diferentes. Quando aplicado a um Set este irá devolver um Set, para o Bag e o Sequence o raciocínio é análogo.

Quando aplicado ao tipo Bag devolve um Bag com os resultados da aplicação de body a cada um dos elementos do Bag inicial. Quando aplicado a um Set este irá devolver um Set, para o Bag e o Sequence o raciocínio é análogo.

```
source->collect(iterators | body) =
        source->iterate(iterators; result : Bag(body.type) = Bag{} |
                        result->including(body ) )
```

Tal como noutros iteradores já abordados, no collectNested é utilizado o foreach para representar o iterate. O body que representa uma expressão OCL qualquer, é representado por Body que já abordamos anteriormente quando falamos dos diferentes tipos de argumentos.

```
public override OclCollection<Object> collectNested(Body<T> aBody)
{
    List<Object> result = new List<Object>();
    foreach (T item in this)
    {
        result.Add(aBody(item));
    }
    return new OclBagImpl<Object>(result);
}
```

O **sortedBy** é um iterador que é implementado em Set, Bag e Sequence porque neste caso a aplicação de sortedBy a um Set retorna um OrderedSet e quando aplicado a um Bag ou Sequence é retornado um Sequence. Isto é assim porque, como vimos no inicio, OrderedSet possui a noção de ordem e não admite elementos repetidos. No caso do Bag e do Sequence, ambos admitem elementos repetidos mas o Sequence é o único que possui a noção de ordem.

No Set o sortedBy devolve um OrderedSet com todos os elementos da colecção inicial. O elemento para o qual o body obteve o resultado mais baixo vem primeiro, os elementos seguintes são ordenados da mesma forma.

```
source->sortedBy(iterator | body) =
    iterate( iterator ; result : OrderedSet(T) : OrderedSet {} |
         if result->isEmpty() then
           result.append(iterator)
         else
           let position : Integer = result->indexOf (
                        result->select (item | body (item) > body (iterator)) -
>first() )
             in
               result.insertAt(position, iterator)
         endif
```

As restrições para os outros casos são análogas a esta, com as devidas adaptações dos tipos.

O **sum** é implementado no Collection por ter uma implementação igual para todos os tipos de colecção.

O resultado é a adição de todos os elementos da colecção. O tipo dos elementos da colecção tem que suportar a operação de adição matemática, os tipos Integer e Real respeitam esta condição.

```
post: result = self->iterate( elem; acc : T = 0 | acc + elem )
```

No C# não é possível fazer operações matemáticas sobre um tipo Template, portanto o seguinte código não é permitido. Quando compilado da erro.

```
public T Sum()
{
    T sum = 0;
    foreach(T elem in this)
                sum += elem;
    return sum;
}
```

A alternativa que apresentamos para ultrapassar esta limitação. Traduz-se na criação de uma classe estática adicional representada no seguinte código:

```
public static class Collectable
    {
        public static int Sum(this IEnumerable<int> elements)
        {
            int sum = 0;

            foreach (int elem in elements)
                sum += elem;

            return sum;
        }

        public static double Sum(this IEnumerable<double> elements)
        {
            double sum = 0.0;

            foreach (double elem in elements)
                sum += elem;

            return sum;
        }

        public static T Sum<T>(this IEnumerable<T> elements)
        {
            throw new NotSupportedException("Unexpected type");
        }
    }
```

Esta classe estende de uma forma especial a interface IEnumerable adicionando uma definição especial para cada um dos casos, para o caso do int, do double e para os restantes casos. Esta extensão é conseguida pela colocação da palavra reservada this como argumento. Só resta na definição de sum chamar um dos métodos auxiliares criados, como mostra o código seguinte.

```
public T sum()
{
    return this.Sum<T>();
}
```

O **append** é implementado apenas no OrderedSet e no Sequence por serem os únicos tipos que possuem a noção de ordem.

O resultado de append irá ser um OrderedSet ou um Sequence, consoante o caso, que contem todos os elementos do OrederedSet ou do Sequence mais o object no final da colecção. O object tem que ser do mesmo tipo que os elementos da colecção.

```
post: result->size() = self->size() + 1
post: result->at(result->size() ) = object
post:   Sequence{1..self->size() }->forAll(index : Integer |
         result->at(index) = self ->at(index))
```

As restrições são exactamente iguais para ambos os casos.

O **prepend** é implementado no OrderedSet e no Sequence exactamente pela mesma razão que o append anteriormente descrito.

O resultado é um OrderedSet ou um Sequence, consoante o caso em que se esteja, que contem no inicio o object seguido de todos os restantes elementos da colecção. O object tem que ser do mesmo tipo que os elementos da colecção.

```
post: result->size = self->size() + 1
post: result->at(1) = object
post:   Sequence{1..self->size()}->forAll(index : Integer |
        self->at(index) = result->at(index + 1))
```

As restrições são exactamente iguais para ambos os tipos.

O **insertAt** é implementado no OrderedSet e no Sequence exactamente pela mesma razão que o prepend e append são implementados nestes dois tipos de colecções.

O resultado é uma OrderedSet ou um Sequence com os elementos da colecção original mas com o elemento object na posição índex. O object tem que ser do mesmo tipo que os elementos da colecção.

```
post: result->size = self->size() + 1
post: result->at(index) = object
post: Sequence{1..(index - 1)}->forAll(i : Integer |
        self->at(i) = result->at(i))
post: Sequence{(index + 1)..self->size()}->forAll(i : Integer |
        self->at(i) = result->at(i + 1))
```

As restrições são exactamente iguais para ambos os tipos.

O **complement**, ou "ñ", como é utilizado na especificação da Standard Library, é implementado apenas no Set.

ñ (s : Set(T)) : Set(T)

O resultado é um Set que contém todos os elementos de self menos os elementos que ocorrem em s.

```
post: result->forAll(elem | self->includes(elem) and s->excludes(elem))
post: self  ->forAll(elem | result->includes(elem) = s->excludes(elem))
```

O **symmetricDifference** é implementado apenas no Set.

symmetricDifference(s : Set(T)) : Set(T)

O resultado é o Set com todos os elementos contidos em self ou em s, mas nunca elementos que ocorram em ambos.

```
post: result->forAll(elem | self->includes(elem) xor s->includes(elem))
post: self->forAll(elem | result->includes(elem) = s   ->excludes(elem))
post: s   ->forAll(elem | result->includes(elem) = self->excludes(elem))
```

Para verificar se os diferentes métodos das colecções estavam bem implementados, foram realizados testes unitários sobre os mesmos. Nestes testes foram utilizadas verificações que traduziam as pos-condições expressas em OCL e definidas na especificação da OCL Standard Library. Assim é garantido que o método tem o comportamento desejado e especificado.

5.3. Interpretação/Execução das Expressões em OCL

Neste capítulo é abordada a estratégia adoptada para a tradução de expressões OCL em código C# executável.

Para a tradução das expressões OCL iremos precisar de um Modelo de Sintaxe Abstracta, também referido neste documento como ASM, que representa a estrutura da expressão OCL que iremos traduzir para C#. Esta estrutura contem informação necessária para podermos construir o código C# equivalente. Porem esta informação por si só não é suficiente, é necessário o acesso à OCL Standard Library e ao modelo sobre o qual se aplica as expressões, para que seja possível instanciar os elementos representados no ASM. A instanciação é conseguida através dos tipos e métodos existentes na biblioteca e através das classes, operações e propriedades existentes no modelo.

A componente responsável por esta tradução é o *Code Generator* que como é mostrado na Ilustração 16 é a ultima componente do módulo MetaSketch OCL Interpreter.

Como também podemos observar na Ilustração 16 o Modelo de Sintaxe Abstracta é gerado no *Abstract Syntax Model Builder*. Esta componente quando gerou a ASM gerou-a já com a informação necessária para a criação de código C#. Cada elemento que constitui a ASM contém em si a informação relevante para a criação do código correspondente a esse elemento. Por exemplo no caso de um *OperationCallExp*, este elemento da ASM possui informação sobre a operação a que se refere (referredOperation), sobre que expressão foi aplicada (source) e qual os seus argumentos (argument). O *OperationCallExp* por ser um *TypedElement* possui o seu próprio tipo que neste caso é obtido a partir da operação a que se refere. É este o género de informação que podemos encontrar num elemento de ASM e que irá ajudar a criar o código C#.

Na implementação do MetaSketch OCL Interpreter houve o cuidado em aplicar o princípio “Aberto/Fechado” que diz que as implementações devem estar abertas a extensões mas fechadas a modificações. Com isto em mente tentou-se ver qual seria a melhor maneira de implementar o gerador de código. Como não fazia sentido

adicionar métodos auxiliares aos conceitos que constituem a ASM, acabamos por adaptar a estratégia já anteriormente utilizada no *Abstract Syntax Model Builder*, que como já foi referido foi desenhada por Birgit Demuth, Heinrich Hussmann e Ansgar Konermann [18]. A estratégia assenta na utilização do padrão Visitor, que é utilizado para percorrer a árvore de sintaxe concreta. Para obter mais informação sobre este padrão de desenho dispõem no Apêndice C a definição que pode ser encontrada na *Wikipedia*.

O *Visitor* permite que se adicione a cada um dos conceitos da ASM as funções necessárias para gerar código. A estas funções que mapeiam expressões OCL para funções C# equivalentes vamos chamar de funções geradoras para evitar confusões com as funções que são geradas. Estas funções geradoras têm que gerar código em tempo de execução, o que obriga a que se utilize um mecanismo que permita a geração das funções equivalentes às expressões OCL em tempo de execução. Esse mecanismo é implementado com recurso ao *delegate* disponível na .NET Framework. O *delegate* permite basicamente a criação de funções anónimas em tempo de execução. No Apêndice D dispõem de mais informação sobre o este artefacto.

Com a implementação do Visitor o *Code Generator* irá percorrer a ASM "visitando" cada um dos conceitos que o constitui, quando o conceito aceita a visita permite que a função geradora, implementada especificamente para esse conceito, aceda à informação contida nesse conceito para instanciar o código equivalente. Como também vimos inicialmente nesta secção, esta informação por si só não é suficiente, a função geradora tem que poder instanciar objectos descritos no Modelo sempre que for necessário e tem que poder instanciar qualquer tipo disponibilizado na OCL Standard Library.

O exemplo que se segue é de uma possível função geradora utilizada para gerar uma função que faz uma multiplicação.

```
public delegate object Functional(IMemento parameters);
...
public static Functional Mult( Functional aFactor, Functional aTerm )
{
    return
        delegate( IMemento aState )
        {
            int x = (int)aFactor(aState);
            int y = (int)aTerm(aState);

            return (x * y);
        };
}
```

A declaração do *delegate* indica a estrutura que o Funcional deverá ter, que tipo de argumentos e que tipo de retorno. A função *Mult* é a função geradora que recebe dois argumentos do tipo *Functional* e retorna um *Functional*. O *Functional* devolvido por Mult só é executado quando evocado, mas como se trata de um método anónimo é fácil "perder-lhe o rasto" por isso os resultados das funções geradoras são associados a variáveis.

Na Ilustração 21 possuem a interface gráfica utilizada em testes de apoio ao desenvolvimento do MetaSketch OCL Interpreter, o que podem ver é uma árvore gráfica que representa um Modelo de Sintaxe Abstracta de uma simples expressão OCL.

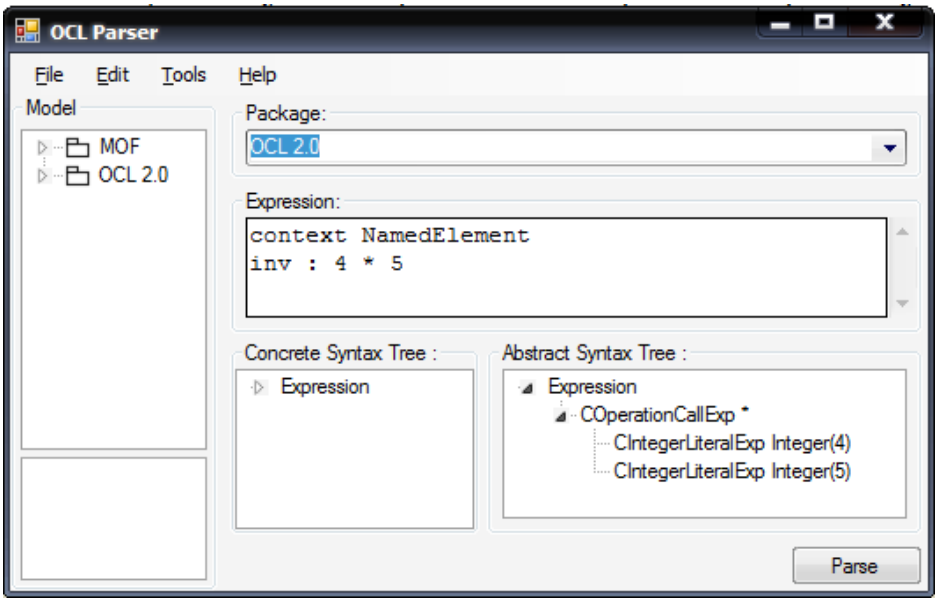


Ilustração 21 - Interface para testes de desenvolvimento

A Ilustração 22 representa a "visita" do *Code Generator* ao ASM resultante do tratamento da expressão apresentada na Ilustração 21.

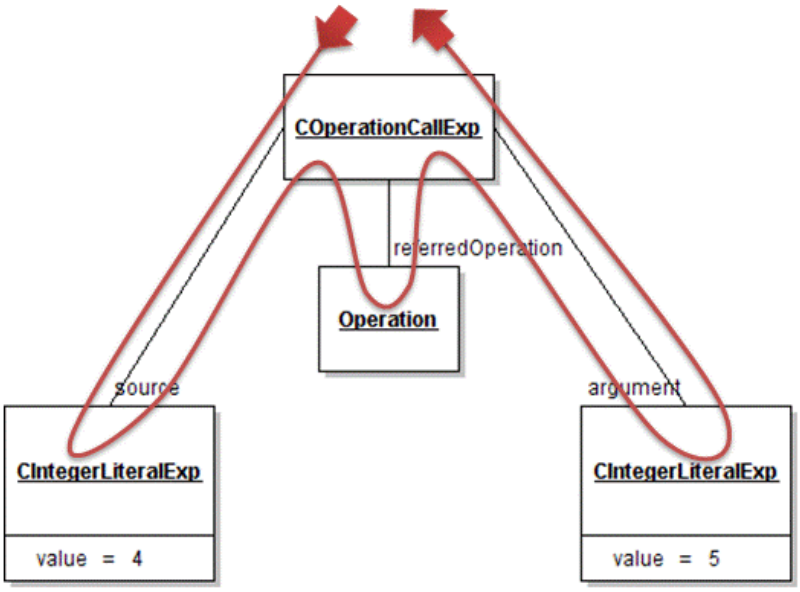


Ilustração 22 - Visita do *Code Generator*

O Code generator ao sair da ASM já gerou todo o código que traduz a expressão original em OCL.

5.4. Integração no MetaSketch Editor

Se é permitido, iremos utilizar uma metáfora para melhor elucidar a importância da integração. Do nosso lado da vedação, o lado do MetaSketch OCL Interpreter, temos as componentes que implementam todos os mecanismos necessários para transformar expressões OCL em código C# passível de ser executado. Do outro lado da vedação, lado do MetaSketch Editor, possuímos as expressões OCL, o MOF Metamodel e o Modelo alvo. O papel da integração pretende ser o de estabelecer uma comunicação entre os dois lados da vedação sem que se destrua a vedação.

A integração com o MetaSketch Editor foi feita em três níveis distintos, ao nível da entrada do OCL no *parser*, ao nível da construção do modelo de sintaxe abstracta e ao nível do mecanismo de validação dos modelos.

No primeiro ponto de integração, ou seja a componente da integração que permite alimentar o *parser* com OCL, é feita uma simples passagem de uma *string* que contem a restrição OCL a ser verificada. A *string* sofre um tratamento para ser introduzida no *parser*. Este pré-tratamento da *string* envolve a retirada de elementos inúteis ao *parser* ou que levantam algum tipo de problema ao *parser*. Em alguns casos pode ser necessário introduzir elementos fundamentais que estejam em falta, tal como o contexto ou a palavra reservada self. Em alguns casos o *self* poderá estar implícito, não aparecendo na expressão OCL, para simplificar o *parser* foi decidido que seria introduzido no pré-tratamento da expressão OCL. Em alguns testes e experiencias com a infra-estrutura do MOF foram detectadas restrições que estavam incompletas ou que eram simplesmente *true*. Este pré-tratamento também pode ser utilizado para corrigir as restrições incompletas, em relação às restrições que são apenas *true* não têm solução a este nível[5].

O segundo ponto de integração é a componente que permite o acesso ao Metamodelo da linguagem de modelação. Isto permite criar o *Environment* que contem as variáveis de ambiente durante a passagem de CST para ASM, permite criar também o *Schema* que contem toda a informação sobre os tipos e suas operações. Portanto este módulo tem que garantir o acesso a todos os tipos existentes no MOF bem como os tipos existentes no OCL.

O terceiro e último ponto de integração está relacionado com a verificação do Modelo alvo. O Modelo alvo é o modelo que o utilizador do MetaSketch Editor está a especificar. Contudo no nosso contexto um modelo é na realidade um Metamodelo de uma linguagem de modelação, como de resto também é explicado na introdução deste documento. Com isto em mente podemos então afirmar que o modelo alvo é um Metamodelo que o utilizador desta ferramenta está a definir.

[5] Ao contrário do que se possa esperar a especificação disponibilizada pela OMG do UML ainda não está completa e em algumas situações não está correcta.

A verificação que irá ser feita pretende dizer se este Modelo, que no nosso contexto corresponde ao Metamodelo que o utilizador está a definir, compre com as regras de modelação estabelecidas pelo Metamodelo da linguagem de modelação utilizada, ou seja, verificar a relação que na Ilustração 10 é representada por *conforms* entre o UML Metamodel e o MOF Metamodel. Note-se que aqui o UML Metamodel é o nosso Modelo alvo.

Esta terceira componente de integração permitirá ao Code Generator aceder ao Modelo alvo e obter os objectos que precisar deste Modelo, porque como já vimos na secção anterior, onde abordamos o tema da geração de código, o Code Generator tem que ter acesso ao Modelo para que possua toda a informação necessária e assim possa traduzir o ASM para codigo C#.

Com estas três componentes de integração implementadas é possível estabelecer uma comunicação entre os dois lados da vedação sem que se destrua a vedação, como se dizia na metáfora. Ou seja, a separação do Modulo MetaSketch OCL Interpreter da ferramenta MetaSketch Editor é evidente, contudo a integração permite uma comunicação transparente entre os dois.

6. Conclusões

Embora a linguagem OCL partilhe alguns aspectos e conceitos das linguagens de programação, não foi desenhada como tal. A sintaxe distinta da linguagem requer alguma adaptação e habituação principalmente para quem está muito familiarizado com linguagens de programação como Java e C#.

A especificação do OCL disponibilizada pela OMG é muito detalhada e extensa, o que torna a tarefa de dominar a linguagem OCL algo demorada e trabalhosa. A existência de alguns documentos e sites, que pretendem ser de alguma forma manuais de introdução ao OCL, atenua a dificuldade que possa existir na compreensão da linguagem do ponto de vista do utilizador. Contudo para a elaboração do MetaSketch OCL Interpreter a visão do utilizador é manifestamente insuficiente.

A gramática da linguagem OCL apresentada na especificação da OMG não é LALR, o que obriga a uma adaptação da gramática se quisermos utilizar um sistema de *parsing* que implemente este algoritmo. A gramática também apresenta alguns erros e inconsistências, e alguns destes problemas são bem identificados por outros autores. Contudo a implementação de uma solução para esses problemas depende de sistema para sistema.

A implementação da OCL Standard Library revelou ser um desafio interessante, exigindo a utilização de novas técnicas de extensão de classes em C#.

Pude constatar que as especificações, quer do UML quer do MOF, avançadas pela OMG e que assentam na utilização do OCL para representar as suas semânticas, possuem alguns problemas. Existem regras que estão incompletas e outras que são apenas “true”, o que denota a falta de um mecanismo de verificação e de definição destas especificações. O MetaSketch Editor pretende ser tal mecanismo, munido do MetaSketch OCL Interpreter para lhe conferir o suporte ao OCL, pode contribuir activamente para uma melhor especificação quer do UML como do MOF. Em última analise, uma melhor especificação do UML levará a uma melhor plataforma para a MDA uma vez que a ultima assenta na utilização dos modelos como principal artefacto no desenvolvimento de software.

Durante a integração do módulo MetaSketch OCL Interpreter tive o privilégio de contactar directamente com o trabalho realizado pelo Professor Doutor Leonel Nóbrega no âmbito da implementação da ferramenta MetaSketch Editor. Este contacto e o trabalho de equipa que envolveu foi muito enriquecedor, a integração permitiu o contacto com estratégias e abordagens só ao alcance de um programador maduro e clarividente.

É um facto que a utilidade deste projecto pode ser incrementada com a sua generalização ao UML. A actual implementação visa suportar o MOF para auxiliar à Metamodelação, a implementação de uma versão do OCL Interpreter cujo objectivo visasse o auxílio à Modelação teria que suportar o UML. A estratégia desenvolvida por

Matthias Bräuer denominada Pivot [22] parece-me uma possível estratégia a adoptar com a vantagem de se poder generalizar o interpretador para qualquer outra linguagem de Modelação ou Metamodelação.

Com execução deste projecto tive a oportunidade de aprender a trabalhar com OCL. Contudo e como já afirmei anteriormente, a perspectiva do utilizador não é suficiente para implementar um interpretador de OCL. Para atingir os objectivos deste projecto tive de estudar toda a especificação do OCL e estuar alguns conceitos do MOF e do UML que estavam relacionados com a Metamodelação e que são fundamentais à compreensão do problema e à tomada de decisões durante a implementação.

Para terminar em jeito de introspecção e olhando para trás, posso afirmar que tenho agora uma melhor e mais alargada visão do problema. Não considerando a área da Metamodelação trivial e de simples compreensão, actualmente tenho uma percepção muito mais completa dos conceitos envolvidos: MOF, UML, OCL e outros. Contudo, sendo esta área tão vasta e em constante evolução, estou consciente que sou um novato que se debate para reter o máximo de informação possível e acompanhar a evolução.

Referencias Bibliográficas

[1] Wikipedia, “Breve definição de OCL em português”, from http://pt.wikipedia.org/wiki/OCL

[2] Wikipedia, “Breve definição de OCL em inglês”, form http://en.wikipedia.org/wiki/Object_Constraint_Language#OCL_and_UML

[5] Wikipedia, “Definição de produto cartesiano”, from http://pt.wikipedia.org/wiki/Produto_cartesiano

[3] Fadi Chabarek (2004). “Development of an OCL-Parser for UML-Extensions” Technische Universit□at Berlin

[4] OMG (2006). “UML 2.0 OCL Specification” from www.omg.org/docs/ptc/06-05-01.pdf

[6] Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise (2004). “MDA Distilled: Principles of Model-Driven Architecture” Addison Wesley

[7] Douglas Baldwin, Greg W. Scragg (2004). “The Object Primer, Agile Model–Driven Development with UML 2.0” Cambridge University Press

[8] David S. Frankel (2003). “Model Driven Architecture, Applying MDA to Enterprise Computing” Wiley Publishing, Inc.

[9] Anneke Kleppe, Jos Warmer, Wim Bast (2003). “MDA Explained: The Practice and Promise Of The Model Driven Architecture” Addison Wesley

[10] Sami Beydeda, Matthias Book, Volker Gruhn (1998). “Model-Driven Software Development” Springer

[11] Leonel D. T. Nóbrega (2007). “Tese de Doutoramento, O Editor MetaSketch” Universidade da Madeira

[12] OMG (2003). “MDA Guide Version 1.0.1” from www.omg.org/docs/omg/03-06-01.pdf

[13] OMG (2003). “UML 2.0 OCL Specification” from www.omg.org/docs/ptc/03-10-14.pdf

[14] Parlez|UML (2005). “UML for Java Developers, Model Constraints & The Object Constraint Language” from http://www.parlezuml.com/tutorials/umlforjava/java_ocl.pdf

[15] MSDN, “.NET Framework Class Library”, form http://msdn.microsoft.com/pt-pt/library/ms229335(en-us).aspx

[16] GOLD Parsing System, from http://www.devincook.com/goldparser/

[17] EmPowerTec AG, “Overview of OCL”, from http://www.empowertec.de/ocl/what-is-ocl.htm

[18] Birgit Demuth, Heinrich Hussmann, Ansgar Konermann (2005). Presentation on “Generation of an OCL 2.0 Parser”, from http://lgl.epfl.ch/members/baar/oclwsAtModels05/slides/hussmann.pdf

[19] Klasse Objecten (2005). Jos Warmer, Anneke Kleppe. from http://www.klasse.nl/

[20] Wikipedia, Breve descrição do sistema GOLD Parser em inglês, from http://en.wikipedia.org/wiki/GOLD_(parser)

[21] Jos Warmer, Anneke Kleppe (2006), “The professional site of Jos Warmer and Anneke Kleppe”, from http://www.klasse.nl/index.html

[22] “The Project pages of the Dresden OCL Toolkit”, from http://dresden-ocl.sourceforge.net/

[23] Christian Hein (2006), “A Presentation of OCL2”, from http://www.modelware-ist.org/index.php?option=com_wrapper&Itemid=133