Universidade do Minho
Escola de Engenharia

João António Cardoso da Mata Oliveira da Paz

Verificação de consultas .QL usando Alloy

Outubro, 2010

Universidade do Minho
Escola de Engenharia

João António Cardoso da Mata Oliveira da Paz

Verificação de consultas .QL usando Alloy

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Professor Doutor Manuel Alcino Cunha

Outubro, 2010

Resumo

As ferramentas de consulta do código-fonte permitem aos programadores de software entender e validar o seu código facilmente. O *SemmleCode* é um exemplo de tais ferramentas e está implementado como um *plugin* para o ambiente de desenvolvimento *Eclipse*. Possibilita o cálculo de métricas, a verificação estática de alguns *bugs* e a consulta de outras informações acerca do código-fonte.

Nesta tese, foi utilizada a linguagem de especificação *Alloy* e respectiva ferramenta de verificação *Alloy Analyser*, para validar consultas escritas em *.QL*, a linguagem de programação do *SemmleCode*. Para tal, foi necessário especificar formalmente o modelo de dados internos do *SemmleCode* e desenvolver um tradutor de *.QL* para *Alloy*. A ferramenta resultante permite ao programador verificar a consistência e equivalência de consultas *.QL*, ajudando-o a detectar consultas irrelevantes ou ambíguas.

Abstract

Source code querying tools allow software developers to easily understand and validate their code. *SemmleCode* is a powerful example of such tools: it is deployed as an *Eclipse* plug-in for querying *Java* source code, retrieving metrics, likely bugs and other data.

In this thesis we will use *Alloy*, and its verification tool *Alloy Analyzer*, to validate source code queries written in *.QL*, the query language of *SemmleCode*. For this, we need to dissect *SemmleCode* and convert its internal data model into a formal *Alloy* model. The resulting tool allows the programmer to check consistency and equivalence of queries written in *.QL*, thus helping the programmer to detect irrelevant and ambiguous queries.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Uma consulta ao código-fonte permite identificar pontos críticos fundamentais para a compreensão e validação do mesmo. O comando *grep*, do sistema operativo *UNIX*, é o exemplo mais comum de definição de consultas para se extrair informação sobre código.

O *SemmleCode* [21] é uma ferramenta única e poderosa que permite definir consultas sobre o código *Java*. As consultas são expressas na linguagem *.QL* [7] e têm como principais objectivos:

- Detectar erros originários de programação ambígua ou práticas inadequadas de programação;
- Codificar regras de programação para garantir o cumprimento de bons estilos de codificação;
- Obter vários tipos de métricas.

No entanto, e aqui incide o nosso trabalho, as consultas estão sujeitas a erros ou ambiguidades, levando a resultados incorrectos ou a um sentimento de falsa confiança no código. Tomemos, como exemplo, a seguinte consulta expressa em *.QL*, que pesquisa todas as classes que não são subclasses, directa e indirectamente da classe `Object`:

```
from Class c, TypeObject o
where not c.getASupertype*() = o
select c
```

Geralmente, uma consulta *.QL* é composta por três secções:

1. Na cláusula `from` são declaradas as variáveis de consulta e seus tipos. Neste caso, vamos procurar os elementos do tipo `Class` e `TypeObject`. Observe-se que `TypeObject` representa a classe *Java* `java.lang.Object` e, portanto, só existe um elemento deste tipo.

2. A cláusula `where` impõe uma restrição sobre o resultado. Neste, incidiremos a pesquisa sobre as classes que não têm `java.lang.Object` como super classe. O `*` no método `getASupertype()` denota o fecho transitivo.

3. Na cláusula `select` são filtrados os resultados da consulta.

O *SemmleCode* executa consultas *.QL* num meta-modelo *Java* instanciado a partir de uma base de dados *Datalog*. Esta base de dados é preenchida com informações extraídas da análise dos ficheiros de código-fonte *Java*.

Se uma consulta não retorna resultados pode ser por esta ser inconsistente ou incorrecta. Ao utilizar o termo *inconsistente* estamos a referir consultas que se contradizem logicamente e nunca serão satisfeitas; por sua vez, o termo *incorrecto* é usado para se identificar as consultas que podem dar resultados, mas que não são o esperado. A consulta dada como exemplo é inconsistente porque nunca retorna resultados, visto não ser possível existir uma classe que não descenda da classe `Object` directa ou indirectamente, de acordo com a afirmação:

> *"A classe (Object) é super classe de todas as outras classes."*

presente na *Java Specification Language* [13], garantida estaticamente pelos compiladores *Java*.

Como investigação preliminar no âmbito deste trabalho, procurámos abordagens para a verificação de consultas. O tema "verificação de consultas" é amplamente estudado e documentado pelo que foi necessário focar os objectivos desta investigação. Sendo assim, decidimos investigar o trabalho existente sobre verificação de consultas *SQL* e *XPath*, duas das mais populares linguagens de consulta, para três problemas de decisão: satisfiabilidade, inclusão e equivalência. Satisfiabilidade é a verificação da existência de alguma estrutura de dados que satisfaz uma dada consulta. Inclusão é-nos dada na teoria de conjuntos pelo operador $\subseteq$; $B \subseteq A$, significando que A contém B, ou seja, todos os resultados da consulta B são também os resultados da consulta de A. Note-se que, vendo as consultas como predicados lógicos, o problema da inclusão corresponde à implicação, ou seja, $A \Rightarrow B$.

A equivalência é representada pelo operador $\equiv$; e pode ser representada como a conjunção de duas inclusões, ou seja, $A \equiv B$ é igual a $A \subseteq B \wedge A \supseteq B$.

Várias abordagens foram estudadas, tanto para *SQL* [5, 28] como para *XPath* [11, 3, 22, 12, 24, 19], o que permitiu uma melhor compreensão do problema que é a verificação de consultas. É usual traduzir as consultas para um formalismo lógico, bem estudado, permitindo assim reutilizar todas as ferramentas e algoritmos de verificação previamente desenvolvidas para esse formalismo. Devido à expressividade destas linguagens, estes problemas de decisão são indecidíveis: a abordagem usual para contornar este problema passa pela definição de um fragmento da linguagem de consulta para a qual estes problemas sejam decidíveis.

Nesta tese, vamos seguir uma abordagem semelhante: a consulta *.QL* será traduzida para a linguagem de especificação formal *Alloy*, para posterior verificação. O *Alloy* foi desenvolvido pelo *MIT*: a sua lógica de especificação é muito simples, baseado no lema "tudo é uma relação", que permite ao utilizador criar modelos de sistemas com relativa facilidade através da abstracção de detalhes desnecessários.

A lógica do *Alloy* é também indecidível: este problema é contornado na ferramenta *Alloy Analyser* usando verificação baseada em técnicas de *model checking*, ou seja, dentro de um *scope* limitado.

Podemos enumerar os passos gerais da estratégia a adoptar:

- Traduzir a linguagem de consulta *.QL* para *Alloy*;
- Expressar os problemas de decisão propostos na mesma notação. Neste caso, apenas a satisfiabilidade de consultas;
- Verificar os problemas de decisão elaborados na notação comum, através da ferramenta de verificação *Alloy Analyser*.

Na Figura 1.1 podemos identificar as vários componentes do projecto e os capítulos onde serão abordados. No capítulo 2, será descrita a ferramenta *SemmleCode* e respectiva linguagem de consulta *.QL*. No capítulo 3, a linguagem de modelação *Alloy* e respectiva ferramenta de verificação *Alloy Analyser* serão apresentadas. A especificação do modelo de dados do *SemmleCode* será descrita no capítulo 4 e as regras de tradução de *.QL* para *Alloy* no capítulo 5. Por fim as conclusões serão apresentadas no capítulo 6.

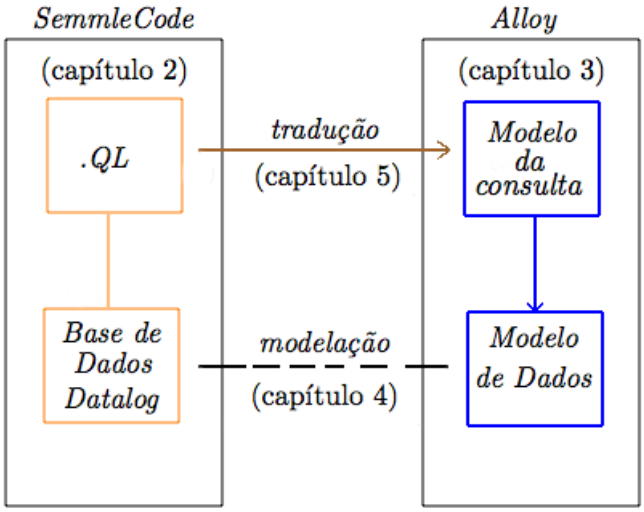

Figura 1.1: Visão global do projecto

Capítulo 2

SemmleCode e *.QL*

Neste capítulo, vamos apresentar a ferramenta de análise de código *SemmleCode* [21] e respectiva linguagem de consulta *.QL* [7]. Esta ferramenta está disponível como um *plugin* para o ambiente de desenvolvimento *Eclipse*, e permite realizar consultas a código *Java*, obter métricas e visualizar os resultados de diferentes maneiras.

A linguagem *.QL* [7] resulta da combinação de várias ideias diferentes [8]:

SQL *.QL* é bastante semelhante ao *SQL*, ao nível da sintaxe. Isto foi uma preocupação dos criadores, para que o *.QL* seja assimilado pelos programadores mais facilmente;

Datalog *.QL* é uma extensão de *Datalog*. *Datalog* é uma linguagem de consulta lógica, semelhante ao *Prolog*. É através de predicados *Datalog* que se acede a informações (nomes, dependências) relativas ao código;

Notação de quantificadores Eindhoven Esta notação proposta por Edsger W. Dijkstra da escola de Eindhoven [9], permite expressar quantificadores que se revelam bastante úteis para a computação de métricas;

Orientação a objectos *.QL* é uma linguagem orientada a objectos. Esta característica é essencial para a reutilização e modularidade de código. A nível semântico a linguagem adopta o princípio de que cada classe é um predicado e herança é uma implicação.

Vamos começar por apresentar com algum detalhe a linguagem de consulta *.QL*. Serão abordadas as consultas, os predicados, as funções agregadoras e as classes.

2.1 A linguagem *.QL*

Comecemos por apresentar a sintaxe das consultas recorrendo a exemplos, como o seguinte:

```
from Class c
where c.declaresMethod("equals")
select c, c.getPackage()
```

Antes de esmiuçar a semântica desta consulta, note-se a semelhança da sintaxe com *SQL*. Em *SQL* as três principais cláusulas das consultas são o **SELECT**, **FROM**, **WHERE**, tal como na sintaxe do *.QL*. A diferença incide na sua ordem.

Sendo assim, uma consulta *.QL* é composta por três secções/cláusulas:

1. Na cláusula `from` são declaradas as variáveis de consulta;
2. A cláusula `where` impõe uma restrição sobre o resultado, através de um conjunto de condições;
3. Na cláusula `select` são filtrados os resultados da consulta, através de um conjunto de expressões;

Destas três cláusulas, apenas a cláusula *.QL* `where` é obrigatória, como se pode consultar no Anexo A, onde é apresentada a gramática completa da linguagem *.QL* na notação Extended Backus-Naur Form (EBNF).

Em termos de semântica, a consulta procura na representação abstracta do código por objectos do tipo `Class` (cláusula `from`) que declarem o método `equals` (cláusula `where`). A consulta devolve uma tabela com duas colunas, uma por cada expressão da cláusula `select`. Com as duas expressões `c` e `c.getPackage()` é seleccionado como resultado a classe e respectiva *package*.

A ferramenta *SemmleCode* vem com uma biblioteca pré-definida que contém uma variedade de consultas que analisam desde propriedades da arquitectura e métricas até verificações de possíveis *bugs* e violações de boas práticas. Das várias consultas de verificação de violações de boas práticas podemos dar como exemplo a seguinte:

```
import default

from RefType sub, RefType sup
where sub.getASupertype() = sup and
```

```
      sub.getName() = sup.getName() and
      sub.fromSource()
select sub, "Class has the same name as its superclass"
```

Esta consulta procura classes (`sub`) que tenham o mesmo nome que a sua super-classe (`sup`): isto revela-se confuso aquando da declaração de variáveis, visto não se saber se o tipo declarado é o da classe ou da super-classe.

Existem também algumas consultas que analisam métricas. A seguinte consulta retorna as classes (`t`) ordenadas pelo índice de especialização (`f`). Este índice de especialização mede a redefinição de métodos herdados numa classe. Por princípio, uma subclasse deve adicionar novos métodos, ou seja, adicionar comportamento e não alterar apenas o comportamento herdado.

```
import default

from RefType t, float f
where t.fromSource() and
      f = t.getMetrics().getSpecialisationIndex() and
      f > 5
select t as Type,
       f as SpecialisationIndex
order by SpecialisationIndex desc
```

Também podemos obter métricas através de outro tipo de consultas como veremos mais adiante.

A utilização de análises pré-definidas e o desenvolvimento de novas consultas faz da ferramenta *SemmleCode* uma peça bastante útil no processo de *reverse engineering*. Este processo consiste na análise estática de um sistema para criar representações abstractas do mesmo sistema.

Medir um conjunto de resultados torna-se uma tarefa fundamental para a análise de código, nomeadamente para obter métricas. Inspirados pela notação dos quantificadores de Eindhoven [9], o *.QL* inclui funções agregadoras para realizar vários tipos de operações aritméticas. Nesta notação, conjuntos podem ser definidos por compreensão usando triplos que obedecem à seguinte forma:

$$\{x : P(x) : f(x)\} \tag{2.1}$$

Este conjunto contém todos os *f(x)* que obedecem à condição *P(x)*.

Como exemplo de utilização desta notação em funções agregadoras, veja-se a consulta seguinte que devolve como resultado o número de linhas de cada *package*. Neste caso, a função *sum* soma todas as linhas de todos as unidades de compilação (ficheiros `.class` ou `.java`) contidas numa *package*.

```
from Package p
select p, sum( CompilationUnit cu  |
               cu.getPackage() = p |
               cu.getNumberOfLines() )
```

Existem outras funções agregadoras para além da operação de somar (*sum*): também se consegue contar (*count*), calcular a média (*avg*), calcular o máximo (*max*) e calcular o mínimo (*min*). Todas elas seguem a mesma gramática, presente no Anexo A, no símbolo não terminal `aggregate`.

As classes possibilitam a reutilização e modularidade de código. O código a seguir exposto contém um excerto da declaração da classe `CompilationUnit`, presente na biblioteca pré-definida, que é subclasse das classes `Element`, `File` e `@cu` que representa um ficheiro `.java` ou `.class` (mais tarde será apresentada a hierarquia de classes da biblioteca pré-definida).

```
/*A compilation unit for a type declaration, i.e., a .java or .class file*/
class CompilationUnit extends Element, File, @cu {

  /** a printable representation of this compilation unit */
  string toString() {
   result = Element.super.toString()
  }
  (...)
}
```

No código é possível identificar um método. De reparar que em *.QL* não existe a instrução de `return`, mas sim, a atribuição do resultado à variável `result`. Esta característica faz com que os métodos sejam não determinísticos, ou seja, os métodos podem devolver vários resultados diferentes. Qualquer valor pode ser atribuído à variável `result`, desde que torne a expressão correspondente verdadeira.

Uma característica fundamental da orientação a objectos é a herança. Nesta linguagem em particular, e ao contrário de outras linguagens orientadas a objectos, uma classe pode

herdar o comportamento de várias classes. Uma das classes presentes na biblioteca pré-definida que tira partido desta característica é precisamente a classe `CompilationUnit`.

Como se verá na Secção 2.2.2, esta biblioteca serve para estabelecer a relação entre a estrutura de classes abstracta e a representação concreta do código presente na base de dados. O símbolo @ junto a um nome identifica uma relação primitiva [7]. Esta relação primitiva é o tipo de uma coluna presente numa tabela da base de dados da ferramenta. Os outros nomes identificam classes da biblioteca pré-definida, que serão descritas na Secção 2.2.2.

2.2 Arquitectura

Uma abstracção da arquitectura da ferramenta SemmleCode encontra-se na Figura 2.1. A ferramenta é constituída por dois componentes principais:

1. Uma base de dados relacional para onde são extraídas todas as informações referentes ao código *Java*. A base de dados e respectivas tabelas constituem a representação concreta do código *Java*;

2. A linguagem de consulta *.QL* que, por sua vez, utiliza a linguagem de consulta lógica (*Datalog*) para aceder aos dados armazenados na base de dados. Esta ponte entre a linguagem *Datalog* e a base de dados fornece a semântica à linguagem *.QL*, permitindo visualizar a informação concreta relativa ao código sob análise (contida na base de dados) numa estrutura de classes *.QL* muito mais conveniente para efectuar consultas complexas.

Em termos de entrada e saída, a ferramenta recebe uma consulta *.QL* e devolve o resultado dessa consulta realizada a um projecto *Java*. Uma característica da ferramenta *SemmleCode* é a possibilidade de visualizar os resultados de diversas maneiras. Os resultados podem ser retornados como tabelas, árvores, gráficos ou *graphs*.

2.2.1 Datalog

A ligação entre a representação abstracta do código em classes *.QL* e a informação armazenada na base de dados relacional é feita através de predicados *Datalog*.

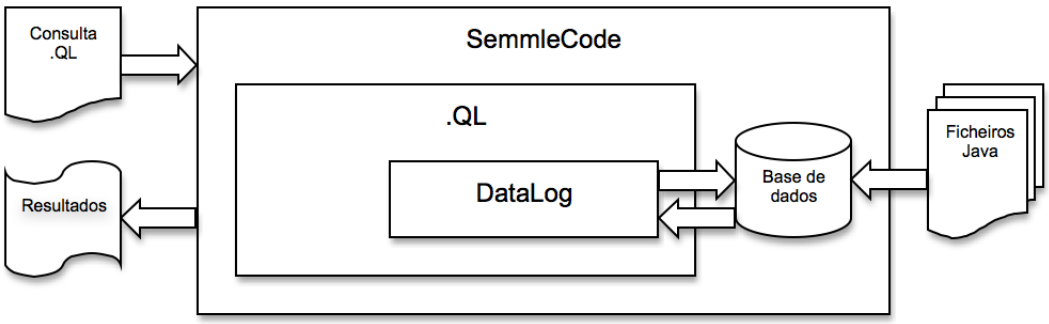

Figura 2.1: Arquitectura da ferramenta SemmleCode

Podemos encontrar esta ligação no predicado `hasName` apresentado de seguida. Este predicado verifica a existência de uma classe *Java* com o nome dado como parâmetro.

```
predicate hasName(@class X, string Name) {
            classes(X,Name,_,_,_)
}
```

O predicado *Datalog* `classes` procura na tabela respectiva, por registos ou tuplos que contenham a classe e o nome dados como parâmetro. A descrição dos campos desta tabela está presente na seguinte declaração:

```
classes    (unique int id: @class,
            varchar(900) nodeName: string ref,
            int parentid: @package ref,
            int cuid: @cu ref,
            int location: @location ref);
```

Convém destacar neste código duas características:

unique Esta palavra identifica uma chave primária que, tal como o nome indica, tem de ser única, para além de não possuir valores nulos;

@ (arroba) Este operador identifica tipos. Os tipos combinados com `ref` identificam uma chave estrangeira como acontece nos campos `parentid`, `cuid` e `location`. Assim, o valor presente nestes campos tem de existir previamente na tabela onde é declarado. A declaração de tipos pode ser realizada de duas maneiras:

Simples Declaração numa tabela. Como o tipo `class` da tabela `classes`;

Composta Declaração fora das tabelas, à custa de uniões de tipos. Vejamos a declaração do tipo `@reftype`:

```
@reftype = @interface | @class | @array | @typevariable;
```

Se algum campo for do tipo `@reftype` está a referenciar qualquer um dos quatro tipos no lado direito da definição.

Mais à frente voltaremos a abordar estas características.

2.2.2 Biblioteca

Um conjunto de classes constitui a biblioteca pré-definida da ferramenta *SemmleCode*. Esta biblioteca possui classes que permitem representar a estrutura da linguagem *Java*. As 184 classes e 289 predicados permitem ao comum programador de *Java*:

- Utilizar as análises pré-definidas, concebida pelos criadores da ferramenta;
- Conceber novas consultas usando as classes e predicados existentes.

Parte da hierarquia da biblioteca de classes está presente na Figura 2.2, onde cada rectângulo representa uma classe e cada seta uma relação de herança.

Algumas das classes presentes na biblioteca serão agora descritas:

Element Classe, que representa um elemento. O elemento é um nodo da árvore de sintaxe que tem nome;

Exception Classe, que representa uma excepção que é lançada por um `Callable`;

CompilationUnit Classe, que representa um ficheiro *.java* ou *.class*;

Type Classe, que representa a abstracção de todos os tipos;

RefType Classe, que representa uma classe, interface, variável de tipo ou array;

Class Classe, que representa um classe *Java*;

Interface Classe, que representa uma interface *Java*;

Callable Classe, que representa um método ou construtor *Java*;

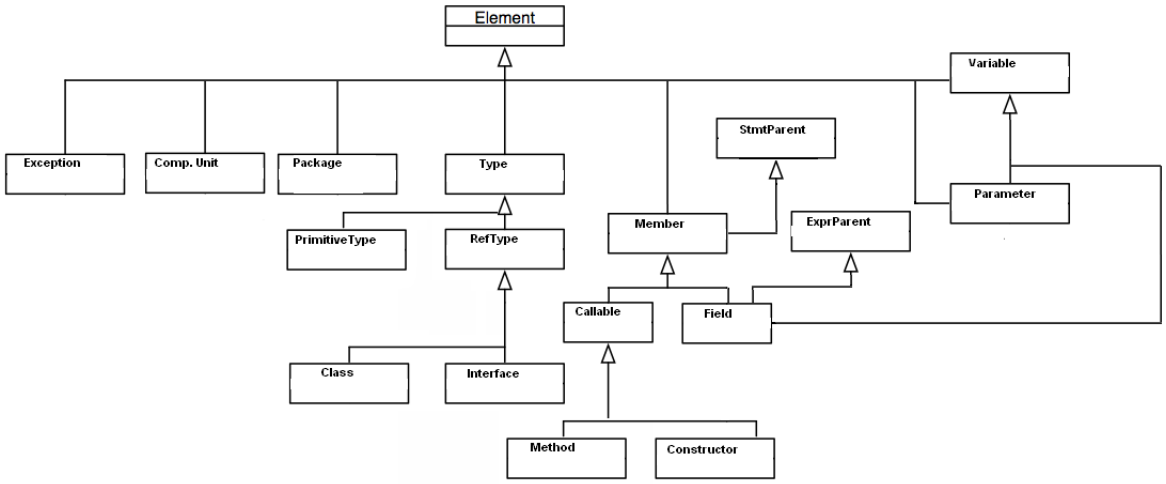

Figura 2.2: Extracto da hierarquia das classes presentes na biblioteca

Method Classe, que representa um método *Java*;

Constructor Classe, que representa um construtor *Java*;

StmtParent Classe, que representa qualquer elemento que possa ter uma instrução;

Parameter Classe, que representa um parâmetro *Java*;

Member Classe, que representa uma abstracção de todos os membros dos tipos, ou seja, métodos, construtores e variáveis de instância;

ExprParent Classe, que representa uma expressão que pode ter outras expressões contidas;

Variable Classe, que representa uma variável;

2.3 Ferramenta

Escalabilidade, eficiência e as características da orientação a objectos [7] fazem parte dos pontos fortes da ferramenta *SemmleCode* e correspondente linguagem de consulta *.QL*.

Escalabilidade A escalabilidade da ferramenta é obtida graças à presença de uma base de dados que armazena dados relativos ao código *Java*. As bases de dados têm a

característica de serem rápidas (eficientes) a executar consultas, tendo em conta que podem conter informação de programas *Java* com milhões de linhas;

Eficiência A eficiência é um tema bastante estudado pelos criadores desta ferramenta. Podemos destacar o trabalho [26] que estuda a eficiência através da optimização do compilador da linguagem de consulta lógica *Datalog*. A optimização da linguagem é obtida através de várias transformações com objectivo de eliminar redundância. A eficiência desta ferramenta está também relacionada com a recursividade da linguagem *.QL*. A recursividade é implementada de modo a que seja eficiente [7];

Orientação a objectos Esta característica presente na linguagem *.QL* permite a reutilização e modularidade do código. Através da biblioteca pré-definida podemos criar consultas/classes personalizadas e posteriormente agrupá-las em novas bibliotecas.

A ferramenta *SemmleCode* está implementada como um *plugin* para o ambiente de desenvolvimento *Eclipse*. Nas Figuras 2.3 e 2.4 é possível visualizar essa mesma integração. Na Figura 2.4 são apresentados os resultados na forma de tabela da consulta da Figura 2.3.

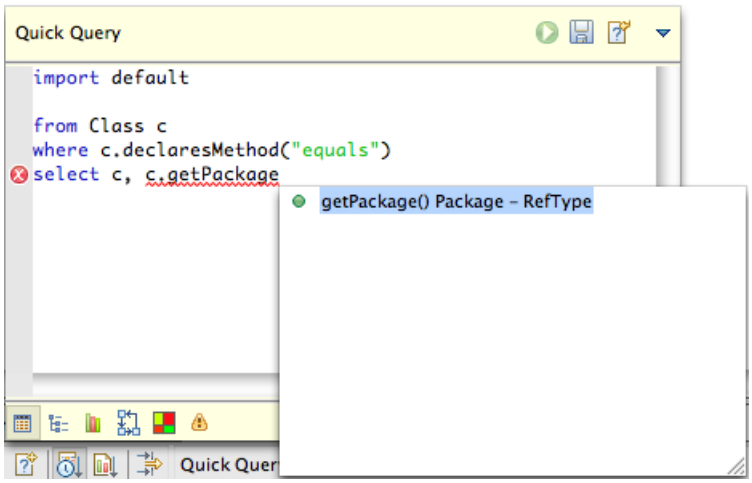

Figura 2.3: Elaboração de uma consulta

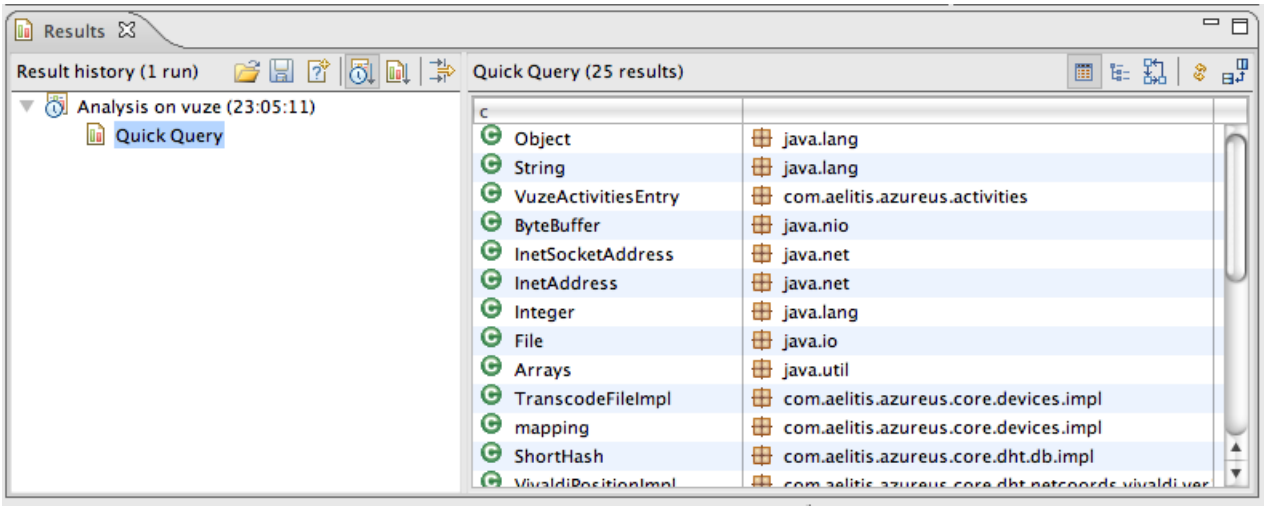

Figura 2.4: Apresentação dos resultados da consulta

Capítulo 3

Alloy

Neste capítulo, será descrita a linguagem de modelação *Alloy* [1] e a respectiva ferramenta de verificação *Alloy Analyser*, desenvolvidos no *MIT* pelo *Software Design Group* sobre a coordenação de Daniel Jackson [15].

O processo de modelação consiste na concepção de modelos que expliquem as características ou o comportamento de um sistema. O processo de verificação de modelos (*model checking*) consiste no teste automático de um modelo sobre uma dada especificação.

A combinação dos processos de modelação e verificação de modelos permite ao programador investigar um sistema, o que se revela fundamental para a compreensão do mesmo (características e comportamento) seja antes ou depois da sua construção.

A linguagem de modelação *Alloy* está associada ao termo "leve" porque tira partido dos benefícios do baixo custo dos métodos de verificação automática [18].

Esta linguagem é semelhante a outras linguagens e técnicas de modelação existentes, como *Z* [27], *VDM* [17] e *B* [20], mas com algumas diferenças, podendo-se destacar os seguintes pontos fortes:

Sintaxe e Semântica Utiliza uma sintaxe e respectiva semântica simples centrada no conceito de relação;

Declarativa Esta linguagem é declarativa, ou seja, responde à pergunta: "como sei que X aconteceu?", em oposição à modelação imperativa/operacional: "como posso atingir X?". O modelador declarativo preocupa-se com o descrever das regras para algo acontecer, enquanto o modelador imperativo com as movimentações para atingir algo;

Análise automática Ao contrário de linguagens como *Z* e *OCL* [29] (a linguagem de restrição de objectos da linguagem *UML* [4]), *Alloy* pode ser analisado automaticamente e gerar instâncias do modelo e contra-exemplos de afirmações realizadas acerca do mesmo.

Neste capítulo, será descrita a linguagem de modelação *Alloy* juntamente com o *Alloy Analyser*. Este processo será acompanhado da construção de um modelo, percorrendo assim as funcionalidades da linguagem.

3.1 A linguagem de modelação *Alloy*

A construção de modelos possibilita a descoberta de falhas. Estas podem resultar do excesso/carência de restrições ou à não obediência por parte do modelo de certas propriedades.

Imaginemos um sistema de ficheiros estático, que contém ficheiros e directorias. Em *Alloy*, teríamos de representar estes componentes através de assinaturas e relações.

```
abstract sig Object {
   name: Name,
   parent: lone Dir
}
sig File extends Object {}
sig Dir extends Object {
   contents: set Object
}
sig Name {}
```

As instâncias de assinaturas são átomos que, ao serem associados entre si, formam relações. De seguida, vamos descrever com mais pormenor cada componente:

`Object` Assinatura que representa um objecto de um sistema de ficheiros. Define as características de outras duas assinaturas (`Dir` e `File`) através da herança. Esta relação entre super e sub assinatura é realizada através da instrução `extends`. A assinatura é precedida pelo qualificador `abstract` que, tal como no paradigma da orientação a objectos, define uma assinatura abstracta e consequentemente sem instâncias para além das instâncias das suas sub-assinaturas;

`File` e `Dir` Assinaturas que representam um ficheiro e directoria, respectivamente;

`Name` Assinatura que representa o nome de um objecto. Devido ao nível de abstracção, o *Alloy* não possui nenhum tipo primitivo que represente uma sequência de caracteres como *string*;

`name` Relação entre `Object` e `Name`, que relaciona cada objecto a um nome. Cada instância de `Object` tem associado só uma instância de `Name`. Por omissão, a declaração da relação `name:Name` é equivalente a `name:one Name`. O qualificador `one` define o tipo de relacionamento de muitos para um (um objecto só pode ter um nome, mas esse nome pode ser usado por muitos objectos);

`parent` Relação entre `Object` e `Dir`, que relaciona o objecto à sua directoria pai. A diferença com a relação `name` é a inclusão do qualificador `lone` que restringe a relação. Cada instância de `Object` tem então associada uma ou nenhuma instância de `Dir`, ou seja, relacionamento de muitos para um ou zero. Esta restrição cobre o caso da directoria raiz que não tem pai;

`contents` A assinatura `Dir` possui, para além das relações herdadas, a relação `contents`, que associa uma directoria a vários objectos (ficheiros ou directorias). O relacionamento de muitos para muitos é obtido através do qualificador `set`.

O comportamento do sistema de ficheiros é definido através de factos. Através da instrução `fact` é possível definir restrições ao modelo de modo a que este seja fiel ao sistema que se pretende modelar. Como já foi referido anteriormente, o excesso/carência de restrições ao modelo pode levar a falhas. Uma restrição que nos parece evidente é a de que toda directoria é pai (`parent`) do seu conteúdo (`contents`).

```
fact {
   all d: Dir, o: d.contents | o.parent = d
}
```

O facto utiliza o quantificador `all` para definir: para todas as directorias `d` e objectos `o` pertencentes ao conteúdo de `d`, `o` tem como pai a directoria `d`.

Existem outros quantificadores que permitem elaborar restrições. Para além do universal (para todos: `all`) e do existencial (existe: `some`), temos o para nenhum (`no`), para um (`one`) e para um ou nenhum (`lone`). Todos seguem a mesma gramática.

Até agora criámos um modelo estático com objectos e relações, mas estas relações não podem ser alteradas. Para isso, teremos de incluir aspectos dinâmicos, por exemplo, as

operações de mover e eliminar um objecto do sistema de ficheiros. Estas operações são normalmente especificadas através de predicados (`pred`) ou funções (`fun`). A diferença entre os predicados e funções é a possibilidade de nas funções se definir um tipo de retorno, enquanto os predicados retornam verdadeiro ou falso.

Antes de criarmos estas operações através de predicados, temos de incorporar num novo modelo a noção de estado.Vejamos o código seguinte, referente a um modelo de um sistema de ficheiros e de seguida a respectiva descrição:

```
abstract sig Object { }
sig File, Dir extends Object { }
sig FileSystem {
  live: set Object,
  root: Dir & live,
  contents: Dir lone-> Object,
  parent: Object ->lone Dir
}

fact {
  root not in parent.Object
  no d : Dir, fs : FileSystem | d -> d in fs.parent
  all o : Object, fs : FileSystem | o -> (fs.contents).o in fs.parent
}
```

Deve-se reparar que, neste novo modelo, as relações passaram das assinaturas `Object`, `File` e `Dir` para a assinatura `FileSystem` que podemos considerar agora como sendo um estado do sistema de ficheiros. A relação entre objecto e nome também foi excluída e apareceram as relações `root` e `live`.

`Object`, `File` e `Dir` As assinaturas representam respectivamente um objecto de um sistema de ficheiros, um ficheiro e uma directoria;

`FileSystem` Esta assinatura representa um sistema de ficheiros e suas características;

`live` Esta relação associa um sistema de ficheiros a todos os objectos do mesmo;

`root` Esta relação identifica a directoria raiz do sistema de ficheiros, que tem a particularidade de não poder ser eliminada (tem de estar na relação `live`). O operador `&` determina a intercepção dos conjuntos de todas as directorias (`Dir`) e de todos os objectos do sistema de ficheiros (`live`);

`contents` Esta relação do sistema de ficheiros, associa uma ou zero directoria a um objecto;

`parent` Esta relação do sistema de ficheiros associa um objecto a uma ou zero directoria. Representa o mapeamento de todas as directorias pai e respectivos objectos de um sistema de ficheiros;

O facto contém três expressões que indicam que a directoria `root` não tem pai, que não existem directorias pai de si mesmo e que os conteúdos de uma directoria têm como pai a mesma directoria.

Vejamos agora a operação de eliminar (`delete`) e mover (`move`). O predicado `move` recebe como argumento dois sistemas de ficheiros que representam o estado inicial (`fs`) e o final (`fs'`), o objecto `x` (ficheiro ou directoria) a mover e a directoria destino `d`.

```
pred move [fs, fs': FileSystem, x: Object, d: Dir] {
  (x + d) in fs.live
  fs'.parent = fs.parent - (x -> x.(fs.parent)) + (x -> d)
  fs'.live = fs.live
  fs'.root = fs.root
  fs'.contents = fs.contents
}
```

O predicado `delete` recebe como argumento dois sistemas de ficheiros que representam o estado inicial (`fs`) e o final (`fs'`), e o objecto `x` (ficheiro ou directoria) a remover/apagar.

```
pred delete [fs, fs': FileSystem, x: Object] {
  x in (fs.live - fs.root)
  fs'.root = fs.root
  fs'.parent = fs.parent - (x -> x.(fs.parent))
  fs'.live = fs.live
  fs'.contents = fs.contents
}
```

Na especificações das operações podemos encontrar as pré e pós-condições: estas condições provêm da necessidade de que o modelo formal tenha o seu domínio bem definido.

O predicado `move` tem uma pré-condição e quatro pós-condição. Na pré-condição, está especificado que o objecto a mover `x` e a directoria destino `d` têm de estar no sistema de ficheiros, ou seja, não podem estar apagados. As quatro pós-condições têm que relacionar o estado inicial (`fs`) e o final (`fs'`) do sistema de ficheiros. Aqui, ao mapeamento das directorias pai do estado inicial é subtraída a relação da directoria pai do objecto `x` e

adicionada a nova relação em que o objecto `x` fica associado com a directoria destino `d`. O resto das relações (`live`, `root` e `contents`) mantém-se do estado inicial para o final.

O predicado `delete` tem uma pré-condição e quatro pós-condições. Na pré-condição, está especificado que o objecto a mover `x` tem de estar no sistema de ficheiros, não pode estar apagado e não pode ser a raiz (raiz não pode ser eliminada). A primeira pós-condição especifica que a raiz do sistema de ficheiros se mantém. A segunda subtrai ao mapemento das directorias pai do estado inicial a relação da directoria pai do objecto `x`. As restantes duas pós-condições mantêm as relações (`live` e `contents`) do estado inicial para o final.

A gramática completa da linguagem *Alloy* está presente no Anexo B na notação EBNF.

3.2 Visualização do meta-modelo e de instâncias

Pegando no modelo da secção anterior, vamos utilizar o *Alloy Analyser* para visualizar o meta-modelo, instâncias do modelo e obter contra-exemplos. A partir das instâncias e dos contra-exemplos vamos refinar o modelo.

Na Figura 3.1 podemos ver o meta-modelo gerado pelo *Alloy Analyser*, onde cada rectângulo amarelo representa uma assinatura e cada seta uma relação, que tem associado o seu nome. O nome das relações pode ter anexado um tipo entre '[' ']' que representam as relações ternárias onde o tipo é o intermédio.

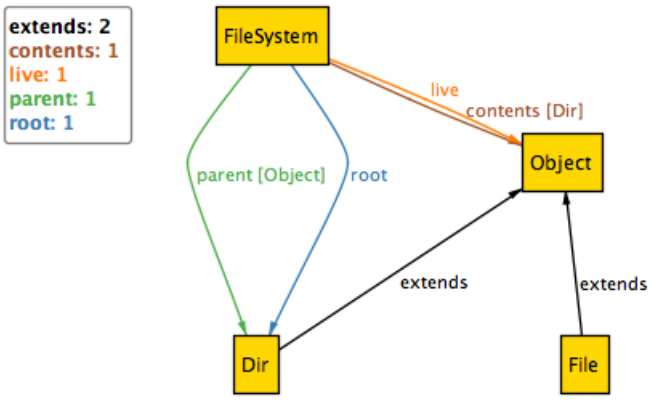

Figura 3.1: Meta-modelo gerado pelo *Alloy Analyser*

Além do meta-modelo, podemos visualizar instâncias do modelo. Estas instâncias são importantes na medida em que ajudam o programador a visualizar exemplos concretos do sistema modelado, que pode não corresponder ao pretendido.

Para visualizar instâncias, podemos utilizar o comando `run`:

```
run {} for 4 but exactly 1 FileSystem,exactly 4 Dir
```

Este comando permite que se gerem instâncias e se defina o tamanho (*scope*) dos mesmos, bem como o número máximo de sistemas de ficheiros, directorias e ficheiros que podem ser incluídos nas instâncias.

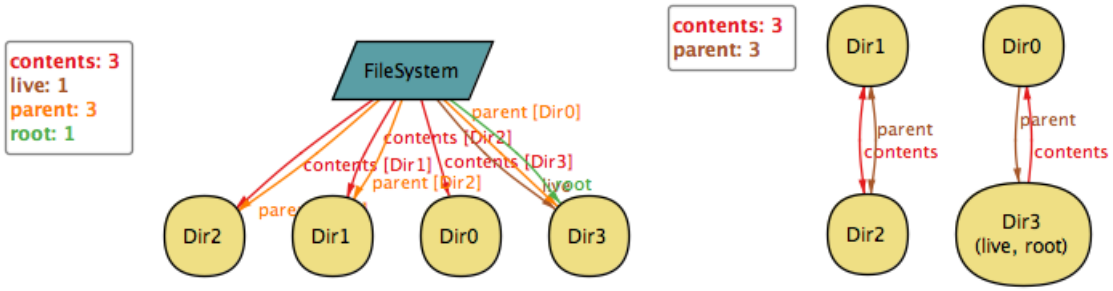

Figura 3.2: Instância gerada pelo comando `run`

A Figura 3.2 é uma instância do modelo. As duas imagens representam a mesma instância, mas a da direita tem projectado a assinatura `FileSystem` o que diminui a complexidade das relações. Nesta Figura podemos identificar duas incorrecções:

1. Todos objectos (neste caso directorias) deveriam ser descendentes da directoria root (`Dir3`);

2. A directoria `Dir1` é pai de `Dir2` e vice-versa.

Como já referido anteriormente, estas instâncias ajudam no refinamento do modelo, pois permitem detectar incorrecções. Para retirar estas incorrecções temos de adicionar factos ao nosso modelo. Neste caso, temos que adicionar factos para garantir que todos os objectos do sistema de ficheiros têm de descender da directoria `root` e a não existência de circularidade entre directoria.

```
fact RootDescendent {
     all f : FileSystem, o : Object | o in (f.root).^(f.contents)
}
fact NoCycles {
      all f : FileSystem, o : Object | o not in o.^(f.parent)
}
```

Estes dois factos explicitam as regras que corrigem as incorrecções detectadas nas instâncias da Figura 3.2. No primeiro facto está escrito que, todos os objectos `o`, têm de estar no fecho transitivo (`^`) dos conteúdos da directoria raiz. No segundo facto, está escrito que para todos os objectos `o`, esse mesmo objecto não pode estar no resultado do fecho transitivo da relação `parent`, isto garante que não há ciclos na relação `parent`, ou seja, não podem existir objectos que sejam antecessores de si próprios directa ou indirectamente.

3.3 Verificação de propriedades

A análise do modelo é feita num determinado tamanho, o que implica que a análise é *incompleta* por não conseguir verificar tudo. Este tamanho limita, tal como o nome indica, o número de instâncias do domínio do problema a verificar. A análise também é *correcta*, ou seja, se encontrar um contra-exemplo para uma asserção, ela é necessariamente falsa. Esta verificação é realizada através de um algoritmo baseado em *SAT* (*satisfiability problem*). Este processo é realizado através da ferramenta *Alloy Analyser* onde, depois da compilação, o problema é traduzido da linguagem de modelação para uma fórmula *booleana* e entregue ao *SAT solver*. As soluções são depois interpretadas e visualizadas pela ferramenta.

No processo de refinação do modelo podemos, a qualquer momento, verificar se o mesmo obedece a certas propriedades. Caso não obedeça, temos de introduzir novas restrições. Para verificar, é necessário expressar a propriedade através da instrução `assert`.

A ferramenta *Alloy Analyser* irá tentar retornar contra-exemplos que invalidem a propriedade. A visualização destes contra-exemplos permite localizar graficamente as falhas do modelo. A propriedade a verificar terá de ser expressa pela pela negativa, ou seja, se não forem retornados contra exemplos significa que a propriedade é correcta no modelo. Por exemplo, a seguinte asserção procura directorias que sejam pais de si mesmas.

```
assert NoSelfParent {
     no f : FileSystem, d : Dir | d.(f.parent) = d
}
```

Para verificar a propriedade é usado o comando `check`.

```
check Propriety1 for 6
```

Como seria de esperar, não é devolvido nenhum contra-exemplo (Figura 3.3), ou seja, a propriedade verificada é verdadeira para o tamanho definido que, neste caso, é seis. Para

ter mais confiança na veracidade da propriedade, podemos aumentar o tamanho e voltar a verificar. Como já foi referido anteriormente, o não retorno de contra-exemplos não assegura a veracidade, visto a ferramenta *Alloy Analyser* ser *incompleta*.

Outras propriedades podem então ser verificadas neste processo incremental de verificação/modelação.

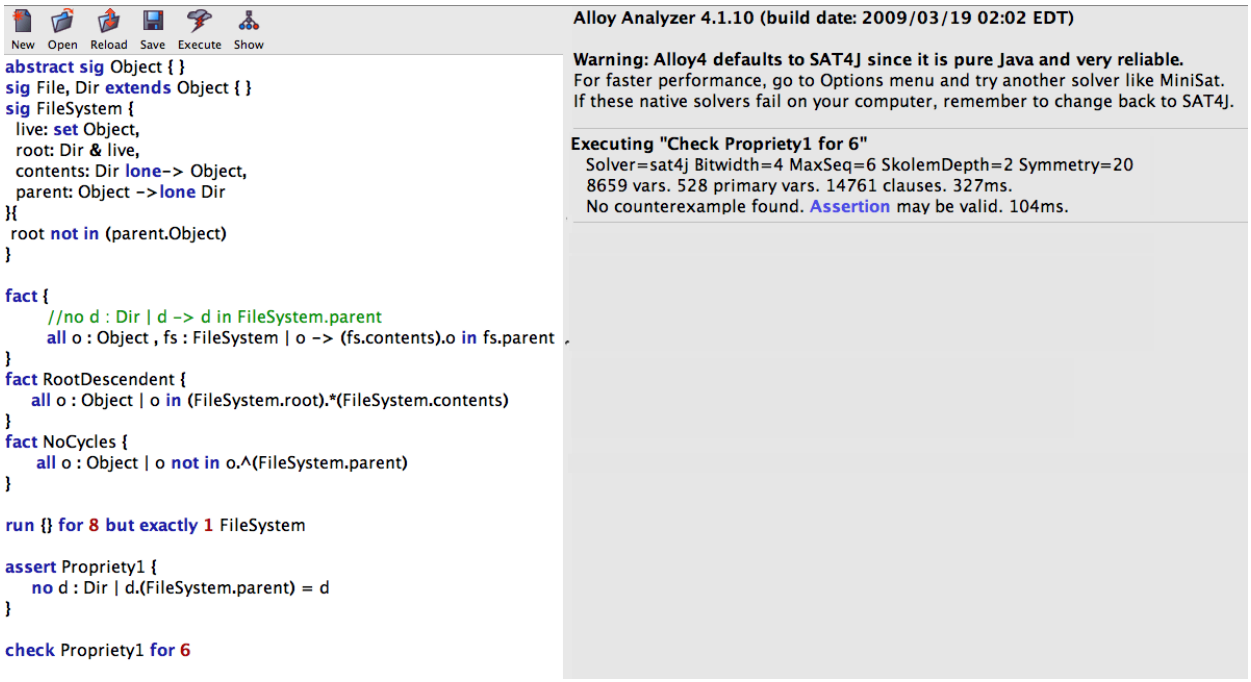

Figura 3.3: Interface do *Alloy Analyser*

Capítulo 4

Especificação e visualização do modelo de dados

A verificação de consultas *.QL* terá por base a linguagem *Alloy* e respectiva ferramenta de verificação *Alloy Analyser*. Como foi apresentado no capítulo 2, a ferramenta *SemmleCode* é composta por dois componentes:

- A linguagem de consulta *.QL*, que segue o paradigma da orientação a objectos;
- Uma base de dados que armazena informação do código *Java*, que é acedida através de predicados *Datalog*, e que corresponde ao modelo de dados da ferramenta.

Neste capítulo será descrita a especificação em *Alloy* do modelo de dados, assim como as técnicas implementadas para permitir a fácil visualização das respectivas instâncias. A tradução das consultas *.QL* para *Alloy* será descrita no próximo capítulo.

4.1 Especificação das tabelas e predicados Datalog

As tabelas da base de dados que armazenam toda a informação do código *Java* são acedidas na linguagem *.QL* através de predicados *Datalog*. Como pretendemos traduzir a linguagem *.QL* para *Alloy* é necessário especificar primeiro as tabelas e predicados *Datalog* nesta linguagem. Mais concretamente, a especificação em *Alloy* do modelo de dados será dividida em três partes:

1. A especificação dos diferentes tipos (referências *Datalog*);

2. A especificação das tabelas;

3. A especificação dos predicados *Datalog* que acedem às tabelas.

Relembremos a definição da tabela `classes` (secção 2.2.1):

```
classes (unique int id: @class,
         varchar(900) nodeName: string ref,
         int parentid: @package ref,
         int cuid: @cu ref,
         int location: @location ref);
```

Cada campo é identificado por dois tipos [7]:

representação Tipo para uso da base de dados adjacente, neste caso: `int` e `varchar`;

coluna Tipo para uso da linguagem *.QL*. Estes são precedidos de uma arroba (`@`), à excepção dos primitivos (como `string` e `integer`).

É nos tipos de coluna que nos vamos focar para especificar em *Alloy* as tabelas. Na tabela `classes`, o atributo `id` define o tipo de coluna que é declarado nesta tabela, neste caso designado por `@class`. Os outros campos são referências a outros tipos de coluna declarados noutras tabelas. Por exemplo, o campo `parentid` faz referência ao tipo `@package`, que é declarado na tabela `packages`:

```
packages (unique int id: @package,
          varchar(900) nodeName: string ref);
```

É necessário criar várias assinaturas para os diferentes tipos de coluna das tabelas a especificar. Neste caso, as seguintes assinaturas foram criadas:

```
sig string {}
sig package {}
sig compilationunit {}
sig location {}
sig class {}
```

Porém, nem todos os tipos de coluna são declarados em tabelas. Existem tipos que são uniões de outros tipos de coluna, por exemplo o tipo `@reftype`:

```
@reftype = @interface | @class | @array | @typevariable;
```

Consequentemente, é necessário declarar os tipos no lado direito desta definição como sub-tipos de `reftype`. Teríamos algo como:

```
sig abstract reftype {}
sig interface extends reftype {}
sig class extends reftype {} // substituindo a anterior assinatura
sig array extends reftype {}
sig typevariable extends reftype {}
```

No entanto, esta abordagem nem sempre é possível. Se um tipo de coluna descender de dois tipos, é necessário optar por uma estratégia diferente, visto que em *Alloy* não existe herança múltipla. Vejamos o tipo de coluna `@typeorpackage`:

```
 @typeorpackage = @type | @package;
```

Como `type` e `package` estendem também outras assinaturas, a alternativa será a seguinte:

```
sig typeorpackage in univ {}{
  typeorpackage = package + type
}
```

A assinatura `typeorpackage` é declarada como subconjunto do universo de todas as assinaturas (`univ`) e o facto adjacente restringe essa assinatura à união de `package` e `type`. Esta abordagem poderá ser usada para todos os tipos, mas torna a verificação menos eficiente, pelo que apenas é usada para os tipos que aparecem em mais do que uma união.

Em relação às tabelas, optámos pela seguinte estratégia de especificação: para cada tabela é criada uma assinatura e para cada campo da tabela é criada uma relação dentro da assinatura. Por exemplo, o resultado da especificação da tabela `classes` em *Alloy* é apresentado de seguida.

```
sig tab_classes      {
      id : class,
      nodeName : string ,
      parentid : package ,
      cuid : compilationunit ,
      location : location ,
}
```

Por omissão as relações são de muitos para um: isto faz com que cada instância de `tab_classes` represente uma linha contida numa tabela de uma base de dados. Sobre

esta representação, torna-se premente especificar as noções de chave primária e chave estrangeira. Uma chave primária pode ser um ou mais campos e não pode conter nem valores nulos nem valores repetidos; uma chave estrangeira é uma referência a uma chave primária. Para a relação `id` da assinatura `tab_classes` se comportar como uma chave primária optamos por introduzir o seguinte facto:

```
id  in  tab_classes one -> class
```

Neste facto, a relação binária `id` só pode associar uma instância de `tab_classes` a uma instância `class`. O comportamento das chaves estrangeiras é obtido a partir do facto anterior, que obriga a que os identificadores referenciados existam nas respectivas *tabelas*, ou seja, se alguma tabela possuir um campo do tipo `class`, ele terá que estar referenciado numa das instâncias da assinatura `tab_classes`.

Dada esta representação para as tabelas, podemos agora especificar os predicados *Datalog* usando predicados *Alloy*. Estes predicados têm de verificar se existe alguma instância da assinatura respectiva que possa ter as diferentes relações iguais aos argumentos. Na sequência do último exemplo, obtemos:

```
pred classes [ a : class, b : string, c : package,
               d : compilationUnit, e : location ] {
   some x : tab_classes | x.id = a  and  x.nodeName = b  and
                          x.parentid = c  and  x.cuid = d  and
                          x.location = e
}
```

O predicado `classes` é verdadeiro se existe alguma instância da assinatura `tab_classes` que tem os seus atributos (`id`, `nodeName`, `parentid`, `cuid` e `location`) iguais aos recebidos como parâmetros (`a`, `b`, `c`, `d` e `e`).

4.2 Semântica estática do *Java*

Dado que estas tabelas são usadas para representar o código *Java*, é agora necessário modelar a semântica estática desta linguagem por forma a evitar instâncias irrealistas. Vejamos a utilização de factos para a atribuição de semântica da tabela `extendsReftype`, representada através da assinatura `tab_extendsReftype`. Esta é constituída por três relações: o filho (`id1`), o pai (`id2`) e o ficheiro onde se encontra (`cuid`).

```
sig tab_extendsReftype {
    id1 : reftype,
    id2 : reftype,
    cuid : compilationunit
}
```

As relações `id1` e `id2` são do tipo `reftype` que, neste caso, pode ser uma classe (`class`) ou uma interface (`interface`).

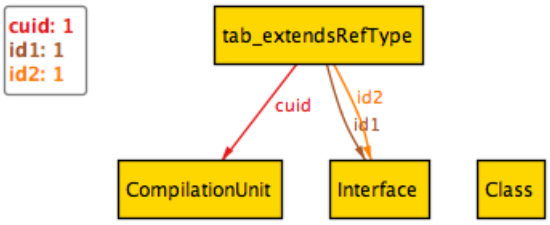

Figura 4.1: Instância gerada

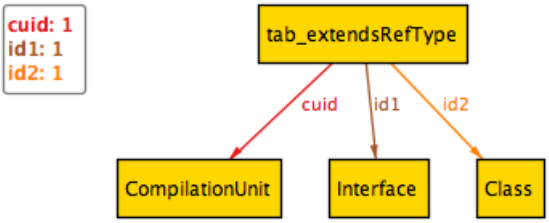

Figura 4.2: Instância gerada

A qualquer momento, no processo de modelação, existe a possibilidade de se gerarem instâncias do modelo. Duas dessas instâncias encontram-se nas Figuras 4.1 e 4.2. Na Figura 4.1, uma interface estende-se a si própria. Para evitar esta circularidade, podemos introduzir o seguinte facto:

```
no r : reftype | r in r.^(~pai)
```

Assim, não existe nenhuma classe ou interface que esteja no fecho transitivo (`^`) da relação filho (o inverso (`~`) da relação `pai`). A relação `pai` resulta da combinação das relações `id1` e `id2` da assinatura `tab_extendsRefType` e encontra-se declarada na seguinte função em *Alloy*:

```
fun pai : reftype -> reftype {
      ~id1.id2
}
```

Outras funções foram acrescentadas de modo a facilitar tanto a leitura como a elaboração de expressões. Apresentamo-las de seguida:

nomeC Associar uma classe ao seu nome. Também foram criadas funções semelhantes para *package* (`nomeP`), interface (`nomeI`), variável (`nomeF`), construtor (`nomeCr`) e método (`nomeM`)

```
fun nomeC : class -> string {
   ~( this/tab_classes <: id) .(this/tab_classes <: nodeName)
}
```

O uso do operador `<:` permite identificar a relação id da assinatura `tab_classes`: este processo é necessário porque existem várias relações com o mesmo nome em assinaturas diferentes.

packageC Associar uma classe à sua *package*. Também foi criada a função `packageI` para interfaces.

```
fun packageC : class -> package {
   ~(this/tab_classes <: id).parentid
}
```

variavel Associar classes/interfaces às suas váriáveis/campos.

```
fun variavel : reftype -> field {
   ~(this/tab_fields <: parentid).(this/tab_fields <: id)
}
```

metodo Associar classes/interfaces aos seus métodos.

```
fun metodo : reftype -> method {
   ~(this/tab_methods <: parentid).(this/tab_methods <: id)
}
```

constr Associar classes/interfaces aos seus construtores.

```
fun constr : reftype -> constructor {
   ~(this/tab_constrs <: parentid).(this/tab_constrs <: id)
}
```

tipoF Associar tipo à variável/campo. Também foram criadas funções semelhantes para os métodos (`tipoM`) e construtores (`tipoC`).

```
fun tipoF : field -> type {
   ~(this/tab_fields <: id).typeid
}
```

cuC Associar classes aos seus ficheiros. Também foram criadas funções semelhantes para os métodos (`cuM`), construtores (`cuCr`), interfaces (`cuI`) e variáveis (`cuF`).

```
fun cuC : class -> compilationUnit {
   ~( this/tab_classes <: id) .(this/tab_classes <: cuid)
}
```

importC Associar uma classe aos seus tipos de *import*. Também foi criada a função `importI` para interfaces.

```
fun importC : class ->  (type +  package) {
   (cuC.~( this/tab_imports <: cuid)).holder
}
```

Na Figura 4.2, uma interface está como subclasse de uma classe, o que é impossível em *Java*: uma classe (interface) só pode herdar o comportamento de outra classe (interface). Para incorporar no modelo esta restrição de tipos, é necessário introduzir os seguintes factos:

```
all c: class | c.pai in class
all i : interface | i.pai in interface
```

A necessidade destes factos é comprovada pelas seguintes afirmações presentes na especificação da linguagem *Java* [13]:

- "A ClassCircularityError is thrown at load time if a class would be a super-class of itself." (pág. 341)
- "An interface may be declared to be a direct extension of one or more other interfaces,(...)" (pág. 259) e "Each class (...) is an extension of (...) a single existing class." (pág. 173)

Uma pesquisa exaustiva da especificação da linguagem *Java* permitiu compilar um conjunto alargado de outros factos necessários à modelação da sua semântica estática, por exemplo:

- "The class Object is a superclass of all other classes." (pág. 47) Todas as classes descendem de `object`;

  ```
  class in *pai.object
  ```

- "Each class except Object is an extension of (that is, a subclass of) a single existing class." (pág. 173) Todas as classes menos a `object` são subclasses e só só são subclasses de uma classe.

  ```
  all c: class - object | one c.pai
  ```

- "The SimpleTypeName in the ConstructorDeclarator must be the simple name of the class that contains the constructor declaration; otherwise a compile-time error occurs." (pág. 240) Nome do construtor é igual ao da classe;

  ```
  all c : constructor | c.nomeCr = (constr.c).nomeC
  ```

- "If unique package names are not used, then package name conflicts may arise far from the point of creation of either of the conflicting packages." (pág. 170) Todas as *packages* devem ter nomes únicos;

  ```
  tab_packages <: nodeName in tab_packages lone -> string
  ```

- "The following character sequences, formed from ASCII letters, are reserved for use as keywords and cannot be used as identifiers : (...) short int byte long char float double boolean (...)" (pág. 21) Os identificadores (nomes) das classes, interfaces, variáveis e métodos não devem possuir as seguintes palavras reservadas;

  ```
  no (class.nomeC + interface.nomeI + field.nomeF + method.nomeM)
  & (char + int_ + double + float + boolean + long + short + byte)
  ```

  Estas oito palavras reservadas são do tipo `string`, tal como o `nomeC`, `nomeI`, `nomeF` e `nomeM`. O tipo `string` representa a sequência de caracteres. Em *Alloy* é necessário criar a assinatura `string`, e para cada palavra reservada é necessário criar outra assinatura que seja sub-assinatura de `string`. Por exemplo, a assinatura `char`, que só pode ter uma instância (`one`), é declarada da seguinte forma:

```
one sig char extends string {}
```

- "A primitive type is predefined by the Java programming language and named by its reserved keyword: short int byte long char float double boolean" (pág. 35) Estes são os oito tipos primitivos existentes na linguagem *Java*;

```
primitive.nomePr in
(char + int_ + double + float + boolean + long + short + byte)
```

No entanto, existem factos que pensamos ser verdadeiros mas para os quais não encontramos justificação na documentação, como o seguinte:

- Duas classes/interfaces só podem ter nomes iguais se estiverem em *packages* diferentes;

```
all disj c1,c2 : class | (id.c1).parentid = (id.c2).parentid =>
      (id.c1).nodeName != (id.c2).nodeName
all disj c1,c2 : interface | (id.c1).parentid = (id.c2).parentid =>
      (id.c1).nodeName != (id.c2).nodeName
```

4.3 Visualização de instâncias

Como vimos na secção anterior, o processo de especificação da semântica estática do *Java* em *Alloy* pode ser acompanhado pela validação visual das instâncias geradas pelo *Alloy Analyser*. Através deste processo, podemos encontrar prematuramente erros que se podem revelar difíceis de perceber se não foram detectados nesta altura. Para instâncias pequenas, como nas Figuras 4.1 e 4.2, as relações e assinaturas são facilmente identificáveis, mas isso já não se verifica para as instâncias geradas por um modelo mais complexo. Para facilitar a visualização, adoptamos a seguinte estratégia:

- Definir novas relações auxiliares a partir das existentes, reduzindo o número de relações necessárias para a compreensão das instâncias. Estas são criadas através da instrução `fun` em *Alloy*, como pudemos ver na secção anterior;

- Criar um tema, ou seja, atribuir cores, contornos e formas para as diferentes assinaturas/relações;

- Traduzir as instâncias geradas pelo *Alloy Analyser* para código *Java* real.

Vejamos a Figura 4.3: olhando para a instância `tab_extendsReftType` e respectivas relações `id1` e `id2`, podemos concluir que a class `Class0` é subclasse `Class1`, embora não seja fácil perceber esse facto rapidamente.

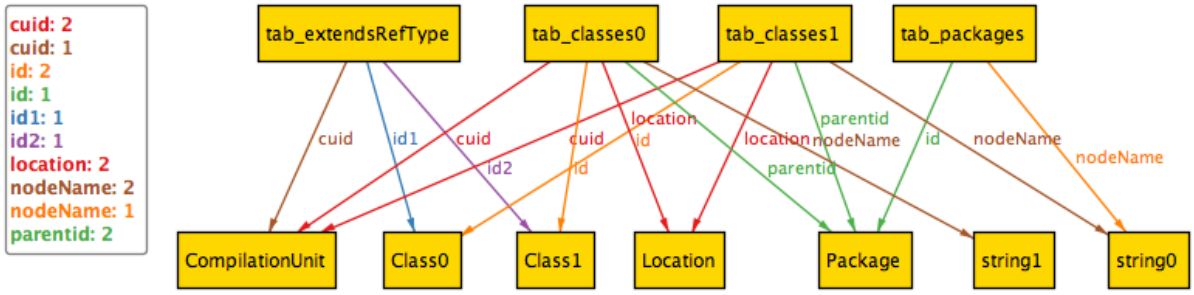

Figura 4.3: Instância gerada sem formatação

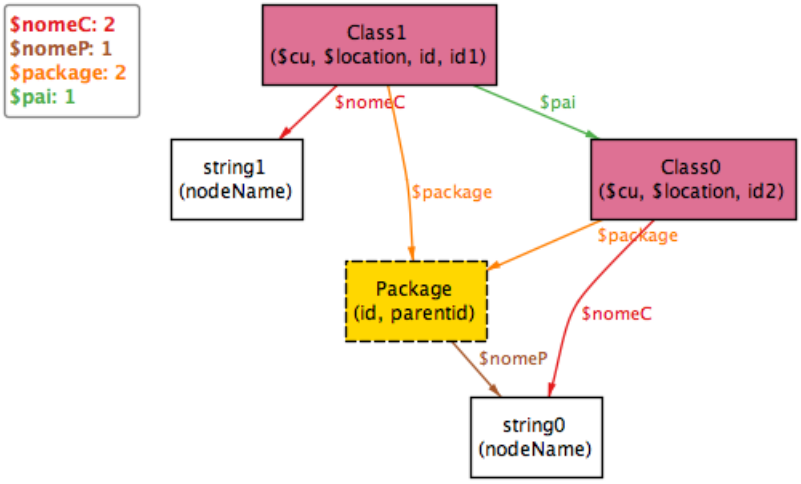

Figura 4.4: Mesma instância gerada que na Figura 4.3 mas com formatação

No tema criado, ocultamos algumas assinaturas (`tab_extendsRefType`, `tab_classes`, `tab_packages`) que passam a ser desnecessárias devido à criação das relações indicadas anteriormente. As assinaturas `Location` e `CompilationUnit` são menos úteis para a compreensão destas instâncias e são apresentadas como atributos de `Class` através das funções `cu` e `location`. Para as assinaturas restantes, foram aplicadas novas cores e contornos. Na Figura 4.4 fica evidente a utilidade desta estratégia. Agora é facilmente legível que a `Class0` é subclasse `Class1`, ambas estão na mesma `Package` e a `Package` e a `Class0` têm o mesmo nome.

Outra estratégia encontrada para facilitar a visualização de instâncias é a geração de código *Java*. A partir do modelo criado em *Alloy*, geramos instâncias, podemos analisá-las e produzir o código *Java* correspondente. Esta alternativa tem como principal vantagem a possibilidade de compilação do código *Java*, o que permite uma validação mais *segura* comparando com a validação visual.

Esta geração de código utiliza uma biblioteca *Java* que permite aceder a várias funcionalidades da ferramenta de verificação *Alloy Analyser*. Umas dessas funcionalidades é a geração de instâncias a partir de um ficheiro de texto com código *Alloy*. Usando esta biblioteca, temos acesso a todas as assinaturas, relações e funções, permitindo assim gerar um código a partir das instâncias do modelo.

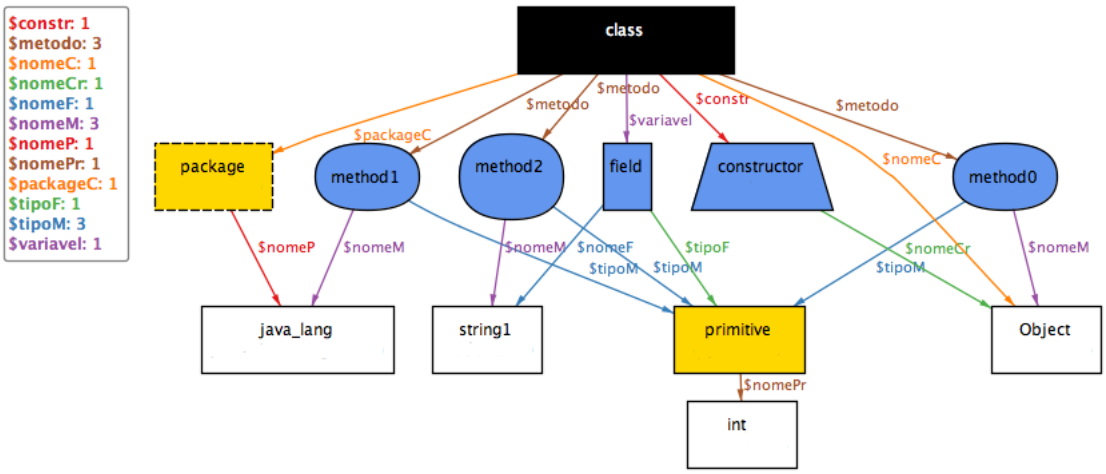

Figura 4.5: Instância gerada que foi traduzida para código *Java*

Na Figura 4.5, encontramos uma classe `Object` presente no *package* `java_lang`. Possui uma variável/campo `string1`, um construtor com o mesmo nome da classe (`Object`) e três métodos (com o nome: `Object`, `java_lang` e `string1`). Os métodos e a variável têm como tipo de retorno o tipo primitivo `int`. Como neste momento o modelo não tem representados os parâmetros e os diferentes tipos de instrução, o código gerado apresenta métodos e construtores sem argumentos e os métodos têm instruções de retorno por omissão. Recorde que palavras entre `/* */` são comentários e apenas servem para fazer a correspondência com a Figura 4.5.

```
package /*package$0_*/java_lang$0;
```

```
public class /*class$0_*/Object$0 {

     /*fields*/
     int /*Field$0_*/string1$0;

     /*constructors*/
     /*constructor$0*//*class$0_*/Object$0() {  }

     /*methods*/
     int /*method$1_*/java_lang$0() { return 0; }

     int /*method$0_*/Object$0() { return  0; }

     int /*method$2_*/string1$0() { return  0; }

}
```

Este código foi posteriormente compilado, o que nos leva a afirmar que o código gerado é estaticamente correcto. Por sua vez, permite ter um elevado grau de confiança nas instâncias geradas e, consequentemente, no modelo.

Capítulo 5

Tradução de .QL para Alloy

A base da linguagem de consulta *.QL* é o acesso à informação armazenada numa base de dados através de predicados *Datalog*. Estando esta componente modelada, como vimos no capítulo anterior, segue-se a modelação em *Alloy* dos componentes que caracterizam esta linguagem como orientada a objectos (classes, métodos e predicados).

5.1 Estratégia da tradução

Para elaborar consultas *.QL*, a ferramenta *SemmleCode* tem disponível uma biblioteca pré-definida com cerca de 250 classes e respectivos métodos e predicados num total de 6000 linhas de código *.QL*. Nesta biblioteca, é representada a linguagem *Java*, desde a hierarquia de classes até à comparação de variáveis em instruções.

A partir da biblioteca pré-definida, personalizámos uma nova biblioteca, reduzindo o número de classes para 13. Na Figura 5.1, podemos identificar essas classes. Isto permitiu:

- Descartar classes desnecessárias para o nível de abstracção pretendido, ou seja, retirar classes referentes a instruções, variáveis e outras especializações de classes;
- Evitar a herança múltipla existente na biblioteca, que não é facilmente traduzível para *Alloy*.

Elaborada esta nova biblioteca, seguiu-se a tradução das consultas e da biblioteca *.QL* para *Alloy*.

Vejamos a seguinte consulta:

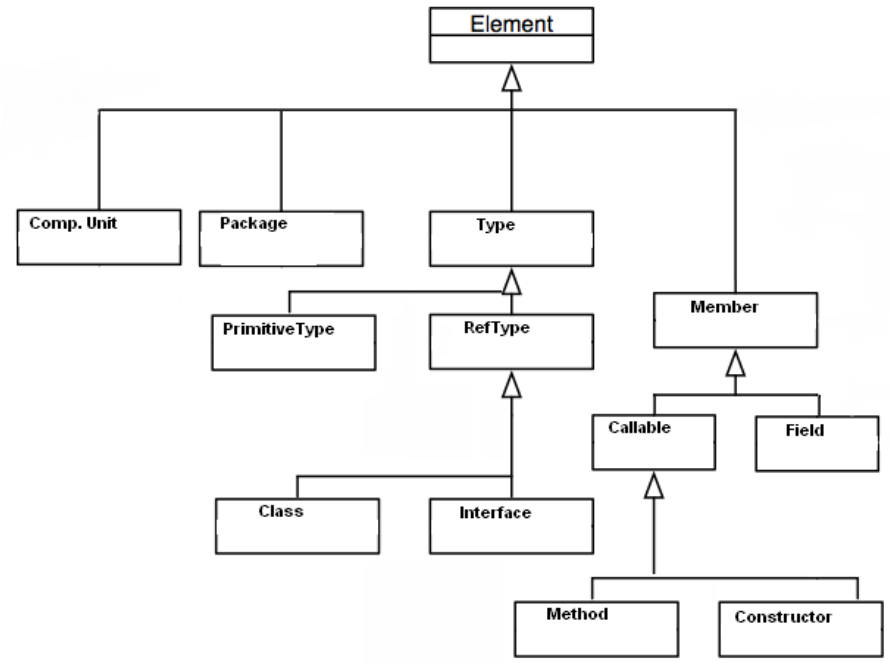

Figura 5.1: Diagrama de classes da biblioteca personalizada

```
from Class c
where c.fromSource()
select c, c.getName()
```

Em *Alloy*, para cada consulta é criado um predicado seguindo a abordagem definida em [7], onde os argumentos são traduzidos da cláusula `from` e da cláusula `select` e o corpo do predicado da cláusula `where`.

```
pred query [ c : Class, a : c, b : c.getName[ ] ] {
   isTrue[c.fromSource[ ]]
   a = c
   b = c.getName[]
}
```

O predicado *.QL* `fromSource` presente na cláusula `where` foi traduzido para a relação com o mesmo nome e invocada pelo predicado `isTrue`. O tipo `Bool` tem dois valores `True` e `False`, que correspondem ao verdadeiro e falso. Ao definir relações (`fromSource`) do tipo `Bool` e, no seu acesso, invocar o predicado `isTrue`, permite que estas relações se comportem como predicados. A decisão de utilizar relações para representar predicados declarados dentro de classes *.QL* permite que a tradução seja mais linear. Ao contrário

do *.QL*, onde uma classe pode possuir predicados, em *Alloy* uma assinatura não possui predicados.

A consulta lista todas as classes do código fonte, criadas pelo programador. A linguagem *.QL* representa as classes *Java* através de uma classe chamada `Class`. A `Class` herda o método `getName()` e o predicado `fromSource()` da classe `Element` que representa todos os elementos *Java* e declara o comportamento geral de cada elemento. Apresentamos de seguida parte do código *.QL* da classe `Element`:

```
/** An element is a node in the syntax tree that has a name */
class Element extends @element {

  /** does this element come from source code? this is useful to
       filter source elements when using on-demand population */
   predicate fromSource() {
      exists(CompilationUnit C | inCompilationUnit(this,C) and
                                 C.getExtension() = "java")
   }

   /** does this element have name name? */
   predicate hasName(string name) { hasName(this,name) }

   /** a name of this element */
   string getName() { this.hasName(result) }

}
```

É facilmente identificável o predicado `fromSource()` que verifica se o elemento se encontra num ficheiro (`CompilationUnit`) e se esse tem a extensão `java`. Já o método `getName()` acede ao predicado `hasName()`, que por sua vez, chama outro com o mesmo nome que se encontra declarado de seguida:

```
/** does element X have name Name? */
predicate hasName(@element X, string Name) {
     classes(X,Name,_,_,_) or
     interfaces(X,Name,_,_,_) or
     primitives(X,Name) or
     constrs(X,Name,_,_,_,_,_) or
     methods(X,Name,_,_,_,_,_) or
     fields(X,Name,_,_,_,_) or
     packages(X,Name)
}
```

Este predicado utiliza predicados *Datalog*, cuja modelação foi abordada no capítulo anterior.

O objectivo presente é modelar em *Alloy* estas classes, métodos e predicados *.QL* que representam a linguagem *Java*. Em seguida, mostramos parte o código *Alloy* necessário para a verificação da consulta exemplo.

```
open tables
open util/boolean

sig Element {
   element : tables/element
   hasName : string -> one Bool,
   getName : string,
   fromSource: one Bool,
}
sig Type extends Element {
   type : tables/type
}
sig RefType extends Type {
   reftype : tables/reftype
}
sig Class extends RefType {
   class : tables/class
}
sig CompilationUnit extends Element {
   cu : tables/cu,
   getExtension : string
}
```

Analisemos agora em detalhe o código *Alloy* apresentado. Em primeiro lugar são importados os módulos necessários para a modelação. O módulo `tables` possui as assinaturas e respectivos predicados das tabelas descritos no capítulo anterior. O módulo `boolean` é necessário para a representação do tipo de dados que representa a veracidade de uma expressão.

Seguidamente são modeladas as classes necessárias para consulta apresentada, neste caso a hierarquia a partir da assinatura `Class` (Figura 5.1), que inclui as classes `RefType`, `Type` e `Element` e as classes dependentes dos métodos e predicados destas (`CompilationUnit`). Para cada assinatura, é necessário associá-la ao correspondente tipo de coluna presente nas tabelas. Estes tipos, que são precedidos de `@`, já foram descritos na secção 2.1. Por exem-

plo, para a assinatura `Element` é criada a relação com o mesmo nome do tipo de coluna: `element`. Isto permite contornar a herança múltipla, visto ser impossível uma assinatura *Alloy* estender mais de uma assinatura. Em contrapartida, caso um predicado tenha um argumento do tipo `element` e tivermos uma instância `x` do tipo `Element`, o argumento terá de ser traduzido como: `x.element`, como veremos mais à frente.

Por fim, através de relações, é necessário representar os predicados/métodos necessários. Para cada método *.QL* da classe é criada uma relação. Para cada predicado *.QL* da classe é criada uma relação para `Bool`. O uso de relações para traduzir estes componentes deve-se ao facto de as relações permitirem uma tradução mais linear. Para cada predicado *.QL* sem classe é criado um predicado. A assinatura `Element`, para além da relação `element`, possui três relações:

hasName Esta relação de `Element` representa um predicado *.QL*. `string` é o tipo do argumento e `Bool` o tipo do resultado;

getName Esta relação de `Element` representa um método. Associa um `Element` a uma `string`;

fromSource Esta relação de `Element` representa um predicado. Como não tem argumentos, apenas associa um `Element` ao tipo de retorno `Bool`.

De seguida, é definido o comportamento de cada relação, através da adição de factos (um para cada relação):

```
fact {
   //methods
   (all x : Element , result : string | (result in x.getName
      <=>  isTrue[x.hasName[result]]))
}
fact hasName_Element {
   (all x : Element, name : string |
   (isTrue[x.hasName[name]])
      <=> (hasName[x.element, name]))
}
pred hasName [ X : tables/element , Name : string ] {
    some c : tabes/class, nodeName' :  string , parentid' :  tables/package ,
               cuid' :  tables/cu , location' :  tables/location |
               classes[ X, Name, parentid', cuid', location'  ] or ...
}
```

```
fact fromSource_Element {
   (all x : Element |
   (isTrue[x.fromSource[ ]])
      <=> (some C:CompilationUnit | inCompilationUnit[ x, C ] and
            (C.getExtension) in string_java))
}
```

O primeiro facto descreve o comportamento da relação `getName` através de uma equivalência (`<=>`). O resultado (`result`) é obtido através da verificação do predicado `isTrue` que é verdadeiro se o acesso à relação `hasName` da instância `x` com o argumento `result` retornar `True`. Como referido anteriormente, visto a relação `hasName` representar um predicado *.QL* da classe `Element`, aquando da sua utilização, é necessário invocar o predicado `isTrue` de modo a que a relação se comporte como um predicado.

O segundo facto define o comportamento da relação `hasName` que é referido no facto anterior. Através de uma equivalência esta relação é verdadeira (`isTrue`) se e só se o predicado `hasName`, descrito a seguir ao facto, for verdadeiro. Este predicado encontra-se incompleto e repare-se na invocação do predicado `classes` que foi definido na secção 4.1. Deve-se destacar também que, apesar da existência da relação `hasName` da assinatura `Element` e do predicado `hasName`, estes não se confundem.

Por exemplo, na Figura 5.2 `Class` está associada a `True` através da relação `hasName` pela instância `string` que é o nome da classe. A relação associa `Class` ao nome `Object` do tipo `string`. Esta relação é obtida através da relação `hasName`.

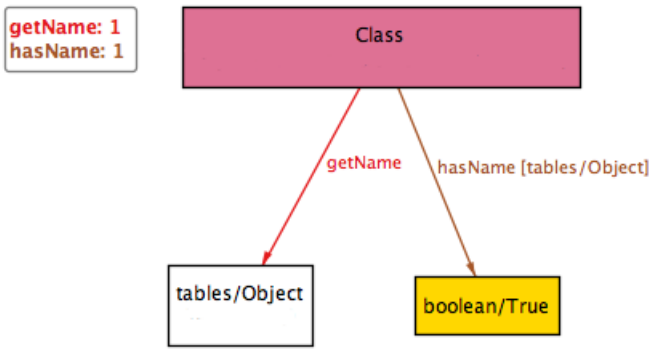

Figura 5.2: Instância gerada - relações `hasName` e `getName`

O facto `fromSource_Element` define o comportamento da relação `fromSource`. Esta relação é verdadeira (`isTrue`) se e só se existir uma `CompilationUnit` que contém o ele-

mento `x` e tiver extensão `java`. É necessário então modelar o comportamento da relação `getExtension` e do predicado `inCompilationUnit`, que são apresentados de seguida:

```
fact {
    ( all x : CompilationUnit , result : string |
        (result in x.getExtension)
            <=> (result in string_java or result in string_class ) )
}
pred inCompilationUnit [ X : tables/element , C : tables/cu ] {
    some  nodeName' :  string , parentid' :  tables/package ,
        location' :  tables/location |
        classes[ X, nodeName', parentid', C, location' ]  or ...
}
```

Por exemplo, na Figura 5.3 `Class` está associada a `True` através da relação `fromSource` porque existe um `CompilationUnit` em que a `Class` está contida e tem uma extensão `java_string`.

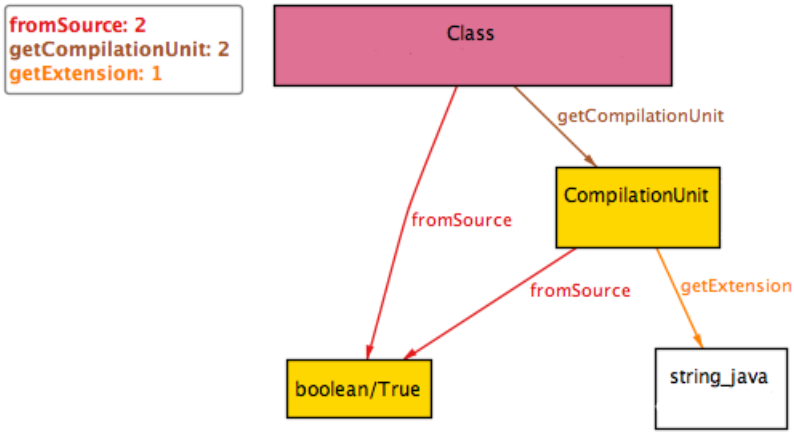

Figura 5.3: Instância gerada - relação `fromSource`

Recapitulando, vejamos a correspondência entre elementos da linguagem *.QL* e *Alloy*:

1. Para cada classe é criada uma assinatura;
2. Para cada tipo de coluna estendido por uma classe é criada um relação dentro da assinatura correspondente à classe;
3. Para cada método é criada uma relação e um facto. A relação é declarada na assinatura da classe correspondente e o facto descreve o seu comportamento, se um

método for redefinido por uma subclasse a sua tradução resulta apenas num facto e não numa relação e um facto. Este facto redefinirá o comportamento da relação herdada;

4. Para cada predicado da classe é criada uma relação para `Bool` e um facto. A relação é declarada na assinatura da classe correspondente e o facto descreve o seu comportamento;

5. Para cada predicado sem classe é criado um predicado.

5.2 Slicing

A modelação da totalidade da biblioteca tem as suas desvantagens:

- A ferramenta *Alloy Analyser* não foi desenvolvida para verificar propriedades sob um número elevado de assinaturas: a sua principal característica é a verificação através da abstracção de problemas;

- Para uma dada consulta, só uma pequena parte destas classes são realmente necessárias.

Surge então a necessidade de fazer *slicing* (traduzido para cortar ou fatiar), que consiste na redução de algo através da descartação de partes desnecessárias. A ideia é construir um modelo fazendo o *slicing* a partir das classes utilizadas na consulta. Recordemos a consulta:

```
from Class c
where c.fromSource()
select c, c.getName()
```

Esta consulta utiliza a classe `Class`, o predicado `fromSource` e o método `getName` para expressar o pretendido. A ideia agora é só seleccionar as classes a que `Class` está directa (através da hierarquia) e indirectamente (através de métodos e predicados) dependente.

Por exemplo, na nossa biblioteca, a classe `Element` conta com oito métodos ou predicados, o que equivale a outras oito relações. Dessas relações, apenas as `getName` e `fromSource` e respectivas relações dependentes são usadas pela consulta (`hasName`). Para seleccionar apenas os componentes dependentes da consulta, foi utilizado o seguinte processo que consiste em dois passos:

1. Popular duas estruturas de dados com as dependências entre classes e entre métodos ou predicados, retirados a partir da biblioteca;

2. Correr o algoritmo de *slicing* definido em [14] que, a partir das classes e métodos ou predicados da consulta, percorre a estrutura de dados seleccionando apenas as classes e métodos ou predicados dependentes.

Consultemos agora a Tabela 5.1 onde estão listados os resultados da geração de código *Alloy* para correr a consulta exemplo, com e sem o processo de *slicing*.

	Classes	Predicados e Métodos	Linhas de código gerado	Redução de código (%)
Sem *Slicing*	13	56	736	-
Com *Slicing*	5	6	90	88%

Tabela 5.1: Comparação entre diferentes níveis de *Slicing* para a consulta exemplo

A redução de classes e respectivos métodos e predicados necessários para modelar a consulta é considerável. Serão necessárias apenas cinco assinaturas e seis relações/predicados, o que corresponde a apenas 12 % do total de assinaturas necessárias para representar a nossa biblioteca.

Em suma, o *slicing* reduz significativamente o número de assinaturas e respectivas relações a modelar. Para a consulta dada como exemplo que depende da classe `Class`, do predicado `fromSource` e do método `getName` existe uma redução em cerca de 88%. Claro que para consultas que usam mais classes e métodos/predicados, essa redução não será tão eficaz.

5.3 Implementação

Nesta secção pretendemos enumerar e explicar as tecnologias usadas para implementar o verificador de consultas *.QL*.

Neste projecto foram usadas as linguagens *Java* e *Alloy*. Para *Java* utilizámos o ambiente de desenvolvimento *Eclipse*, enquanto que, para *Alloy*, a respectiva ferramenta de verificação.

O *projecto* criado pode ser divido em seis partes:

1. *Parsing* de *.QL*, das consultas e da biblioteca;

2. Tradução de *.QL* para *Alloy*, usando as regras descritas anteriormente;

3. Compilação do código *Alloy* gerado;

4. Execução da consulta no código *Alloy* gerado;

5. Geração de código *Java* caso resultem contra-exemplos da execução do passo anterior.

Nas próximas subsecções iremos percorrer as tecnologias e algoritmos usados para implementar cada um destes cinco passos.

5.3.1 *Parsing* e Tradução de *.QL*

Em primeiro lugar foi necessário desenvolver um analisador de sintaxe para a linguagem alvo deste projecto, o *.QL*. Ao existir um prévio conhecimento do gerador de analisadores sintácticos, *YACC* [30], que gera o código *C*, foi relativamente simples usar o seu homónimo para *Java*, chamado *BYACC/J* [6]. Os geradores de analisadores sintácticos são sempre combinados com geradores de analisadores léxicos. Para *C* existe o *FLEX* [10] enquanto para *Java* o *JFLEX* [16]. Para utilizar este gerador é necessário em primeiro lugar definir a gramática do analisador sintáctico pretendido. A gramática da linguagem *.QL* está bem documentada [25] e basta ser adaptada para a notação utilizada pelo *BYACC/J*.

Uma gramática é composta por [23]:

Símbolos terminais Símbolos básicos da linguagem, que definem o seu alfabeto, sendo reconhecidos através de analisadores léxicos;

Símbolos não terminais São classes sintácticas que definem conjunto de frases;

Símbolo inicial É um símbolo não terminal que representa a linguagem definida pela gramática;

Conjunto de produções São regras de sintaxe, onde uns símbolos são definidos à custa de outros.

Na linguagem de especificação usada no *BYACC/J*, temos o seguinte extracto da gramática:

```
Classes : Class
        | Classes Class
        ;
Class : CLASS ID Extends '{' Members '}'
        ;
Extends :
        | EXTENDS Parents
        ;
Members :
        | Member
        | Members Member
        ;
Member : Constructor
        | Predicate
        | Method
        ;
```

O símbolo inicial `Classes` representa o início da gramática que pretende analisar um conjunto de classes em *.QL*. Cada classe é definida por:

- Dois símbolos terminais, que por convenção são escritos com maiúsculas: `CLASS` representa a palavra reservada *class* e `ID` um identificador;
- Dois símbolos não terminais (`Extends` e `Members`), que definem os pais da classe e os membros da classe (construtores, predicados e métodos).
- Dois caracteres ('{' e '}').

O analisador léxico satisfaz a necessidade de reconhecer os símbolos não terminais da gramática. O gerador *JFlex* utiliza expressões regulares para especificar a sintaxe dos símbolos terminais. Vejamos a especificação dos símbolos não terminais `ID` e `CLASS`:

```
ID = {LETTER} ( {LETTER} | {DIGIT} | _ )*
LLETTER = [a-z]
ULETTER = [A-Z]
LETTER = {LLETTER} | {ULETTER}
DIGIT = [0-9]
CLASS = class
```

Um identificador `ID` é definido por uma expressão regular que começa por uma letra (`LETTER`) (maiúscula (`ULETTER`) ou minúscula (`LLETTER`)) e seguida de zero ou mais: letras, dígitos (`DIGIT`) ou o caracter _. A expressão regular que define `CLASS` é a sequência

dos caracteres `class`. Todos os símbolos terminais reconhecidos pelo analisador léxico são retornados ao analisador sintáctico.

Além da especificação da gramática e de expressões regulares, é necessário definir o comportamento aquando do reconhecimento de cada símbolo não terminal. Em *Java* são implementadas as regras da tradução de *.QL* para *Alloy* já descritas no início deste capítulo. Estas regras permitem a geração de código *Alloy*, que é posteriormente guardado num ficheiro. Neste ficheiro encontra-se:

- Importação de módulos necessários;
- Assinaturas, suas relações e predicados da biblioteca;
- Predicado correspondente à consulta;
- Comando para execução do predicado da consulta.

5.3.2 Compilação e execução de código *Alloy* e geração de código *Java*

O analisador sintáctico criado produz código *Alloy* para um ficheiro. O código deste ficheiro necessita de ser validado de alguma forma, tornando-se imperativo que o mesmo seja compilado antes de ser executado. É aqui que entra uma biblioteca que disponibiliza o acesso às funcionalidades do *Alloy Analyser* usando código *Java* [2]. Esta biblioteca permite:

- Analisar a sintaxe de um ficheiro de texto que contenha código *Alloy*;
- Verificar tipos;
- Executar comandos;
- Retornar mensagens de diagnóstico e resultados;
- Guardar o resultado como um ficheiro XML;
- Mostrar visualmente os resultados a partir do ficheiro XML.

Dando o nome do ficheiro onde se encontra o código *Alloy* gerado, é então analisada a sua sintaxe, a consistência de tipos, e por fim, a execução do comando referente à verificação de instâncias possíveis da consulta. Depois da execução do comando, caso seja encontrado algum contra-exemplo, os resultados são traduzidos para código *Java*. Este último passo já foi previamente descrito na secção 4.3.

5.4 Uso da ferramenta

O uso da ferramenta criada permite verificar consultas *.QL*, sendo realizada através da linha de comandos. Esta recebe dois argumentos:

- O nome do ficheiro que contém a consulta a verificar;
- O número `0` ou `1`, que permite ao utilizador, desligar ou ligar o *slicing*.

Supondo que o ficheiro `consulta.ql` possui uma consulta, vejamos o seguinte comando `java`, para correr a classe principal `Main` da ferramenta:

```
java Main consulta.ql 1
```

5.5 Verificação de consultas

Nesta secção iremos apresentar algumas consultas *.QL* e respectivo resultado da verificação, usando a ferramenta criada.

Recordemos a consulta presente na introdução deste documento:

```
from Class c, TypeObject o
where not c.getASupertype*() = o
select c
```

A consulta pesquisa todas as classes que são directa e indirectamente descendentes da classe `Object`. O seguinte predicado, traduzido a partir da consulta, é criado:

```
pred query[ c:Class,o:TypeObject, a :c ] {
   not c.*getASupertype[] in o
   a = c
}
```

Como era esperado, a verificação desta consulta não retornou nenhum contra-exemplo, portanto, não existe (para um *scope* definido) nenhuma classe `c` que não seja subclasse da classe `Object o`, directa e indirectamente.

Existe uma bateria de consultas da ferramenta do *SemmleCode* que constitui a sua análise pré-definida. Para isso, recordemos a seguinte consulta, já previamente introduzida na secção 2.1:

```
from RefType sub, RefType sup
where sub.getASupertype() = sup and
   sub.getName() = sup.getName() and
   sub.fromSource()
select sub
```

Esta consulta pesquisa classes e interfaces que tenham o mesmo nome que o seu super-tipo, e quando traduzida para Alloy resulta no seguinte predicado:

```
pred query[ sub:RefType,sup:RefType , a : sub] {
   sub.getASupertype[] in sup and
   sub.getName[] in sup.getName[] and
   isTrue[sub.fromSource[]]
   a = sub
}
```

Ao ser executado, no modelo resultante do *slicing*, é encontrada pelo menos uma instância que satisfaz o predicado. Quando traduzido para Java, este exemplo origina duas classes com o nome Object em packages diferentes. A primeira class Object encontra-se declarada na *package* `java_lang`:

```
package /*tables/package$1_*/java_lang$0;

public class /*tables/class$1_*/Object$0 {

}
```

A segunda classe está na *package* `string_java`:

```
package /*tables/package$0_*/string_java$0;

public class /*tables/class$0_*/Object$0 {

}
```

A verificação desta consulta permite então confirmar que é satisfazível.

Capítulo 6

Conclusão

O objectivo deste projecto consistia na validação de consultas *.QL* usando *Alloy*. Para a consecução deste propósito, foram realizadas as seguintes tarefas:

- Especificámos o modelo de dados do *SemmleCode* em *Alloy*, adicionando factos para capturar a semântica estática do *Java*;
- Criámos uma abordagem para traduzir um fragmento da linguagem de consulta *.QL* para *Alloy*;
- Criámos uma abordagem para verificar de consultas usando o *Alloy Analyser*, onde os exemplos gerados pela ferramenta são posteriormente traduzidos para código *Java*.

Neste projecto, foi demonstrado o poder da linguagem de modelação *Alloy*, que se revelou bastante eficaz na criação de modelos para posterior verificação através da ferramenta *Alloy Analyser*. Esta tem algumas limitações, mais em concreto o facto de não suportar um elevado número de assinaturas e relações. Para evitar esta limitação, utilizamos a técnica de *slicing*.

A arquitectura do projecto permite que, mesmo que se altere a biblioteca a partir do qual é traduzido o modelo do *Java*, se consigam verificar consultas. Este processo acontece porque o modelo do *Java* é construído a partir do modelo de dados: qualquer modelo que seja criado a partir daquele herda todas as restrições presentes no mesmo. A geração de código *Java* possibilitou a validação ou visualização dos resultados que se revelariam pouco legíveis caso apenas visualizássemos as instâncias no *Alloy Analyser*. Devido à extensa documentação do *Java* ainda não conseguimos reunir uma quantidade suficiente de factos que permitissem ter um modelo de dados mais fiável.

Como trabalho futuro, seria importante a integração deste projecto com o *Semmle-Code*, através de um *plugin* do ambiente de desenvolvimento *Eclipse*. A adição de herança múltipla suportada pelo *.QL* seria também fundamental para conseguir modelar a totalidade da linguagem.

Apêndice A

Gramática da linguagem .QL

```
script = { import }  { declaration }
import = "import" identifier "." { identifier }
declaration = classless-predicate
      | class
      | from-where-select
classless-predicate = "predicate" lowercase-identifier "(" { parameter } ")" "{"
            term
      "}"
parameter = type-reference identifier
class = "class" uppercase-identifier
            [ "extends" type-reference "," { type-reference } ] "{"
            { member }
      "}"
member =  constructor
      | predicate
      | method
constructor = uppercase-identifier "(" ")" "{"
            term
      "}"
predicate = "predicate" lowercase-identifier "(" { parameter } ")" "{"
            term
      "}"
method = type-reference lowercase-identifier "(" { parameter } ")" "{"
            term
      "}"
from-where-select = [ "from" { variable-declaration }  ]
                    [ "where" term ]
                    "select" { select-expression }
                    [ "order by" identifier [ "asc" | "desc" ] ]
variable-declaration = type-reference identifier
```

```
select-expression = expression [ "AS" identifier ]
term =   "(" term ")"
      | term "or" term
      | term "and" term
      | "not" term
      | "exists" "(" expression ")"
      | "exists" "(" { variable-declaration } "|" term ")"
      | "exists" "(" { variable-declaration } "|" term | term ")"
      | "forall" "(" { variable-declaration } "|" term ")"
      | "forall" "(" { variable-declaration } "|" term "|" term ")"
      | "forex" "(" { variable-declaration } "|" term "|" term ")"
      | term "implies" term
      | "if" term "then" term "else" term
      | expression comparison-operator expression
      | expression "instanceof" type-reference
      | classless-predicate-call
      | predicate-or-method-call
classless-predicate-call =  literal-identifier  [ "*" | "+" ] { literal-argument }
literal-argument = expression
      | "_"
predicate-or-method-call = expression "."
    lowercase-identifier  [ "*" | "+" ] { expression }
expression = "(" expression ")"
           | unary-arithmetic-operator expression
           | expression binary-arithmetic-operator expression
           | "(" type-reference ")" expression
           | constant
           | variable
           | predicate-or-method-call
           | aggregate
constant =  integer-literal
           | float-literal
           | string-literal
           | "true"
           | "false"
variable = identifier
           | "this"
           | [ type-reference "." ] "super"
           | "result"
aggregate = aggregate-function "("  { variable-declaration } "|"  [ term ]
          [ | expression ] ")"
          | aggregate-function "(" expression ")"
aggregate-function = "count"
                   | "sum"
                   | "avg"
```

```
                  | "max"
                  | "min"
type-reference = class-reference
           | column-type-reference
           | primitive-type-reference
class-reference = uppercase-identifier
column-type-reference = at-lowercase-identifier
primitive-type-reference = "int"
           | "float"
           | "string"
           | "boolean"
comparison-operator = "="
           | "!="
           | "<"
           | ">"
           | "<="
           | ">="
unary-arithmetic-operator = "+"
           | "-"
binary-arithmetic-operator = "+"
           | "-"
           | "*"
           | "/"

LEXICAL CONVENTIONS
lowercase-letter = a..z
uppercase-letter = A..Z
letter = uppercase-letter | lowercase-letter
digit = 0..9
identifier =  letter { letter | digit | "_" }
lowercase-identifier = lowercase-letter { letter | digit | _  }
uppercase-identifier = uppercase-letter { letter | digit  | "_"  }
at-lowercase-identifier =  "@" lowercase-letter { letter | digit | "_"  }
literal-identifier = lowercase-identifier | at-lowercase-identifier
```

Apêndice B

Gramática da linguagem Alloy

```
specification = [module] {open} {paragraph}

module = "module" name [ "[" ["exactly"] name {("," ["exactly"] num)} "]" ]

open = ["private"]  "open"  name  [ "[" {ref,} "]" ]  [ "as" name ]

paragraph = factDecl | assertDecl | funDecl | cmdDecl | enumDecl | sigDecl

factDecl = "fact" [name] block

assertDecl = "assert" [name] block

funDecl = ["private"] "fun" [ref "."] name "(" {decl,} ")" ":" expr block
funDecl = ["private"] "fun" [ref "."] name "[" {decl,} "]" ":" expr block
funDecl = ["private"] "fun" [ref "."] name                ":" expr block

funDecl = ["private"] "pred" [ref "."] name "(" {decl,} ")" block
funDecl = ["private"] "pred" [ref "."] name "[" {decl,} "]" block
funDecl = ["private"] "pred" [ref "."] name                  block

cmdDecl = [name ":"] ["run"|"check"] [name|block] scope

scope = "for" number                    ["expect" [0|1]]
scope = "for" number "but" {typescope,} ["expect" [0|1]]
scope = "for"              {typescope,} ["expect" [0|1]]
scope =                                 ["expect" [0|1]]

typescope = ["exactly"] number [name|"int"|"seq"]

sigDecl = {sigQual} "sig" {name,} [sigExt] "{" {decl,} "}" [block]
```

```
enumDecl = "enum" name "{" name  {("," name)}  "}"

sigQual = "abstract" | "lone" | "one" | "some" | "private"

sigExt = "extends" ref
sigExt = "in" ref {["+" ref]}

expr = "let" {letDecl,} blockOrBar
       | quant {decl,}    blockOrBar
       | unOp expr
       | expr binOp   expr
       | expr arrowOp expr
       | expr ["!"|"not"]? compareOp expr
       | expr ["=>"|"implies"] expr "else" expr
       | expr "[" {expr,} "]"
       |     number
       | "-" number
       | "none"
       | "iden"
       | "univ"
       | "Int"
       | "seq/Int"
       | "(" expr ")"
       | ["@"] Name
       | block
       | "{" {decl,} blockOrBar "}"

decl = ["private"] ["disj"] {name,} ":" ["disj"] expr

letDecl = name "=" expr

quant = "all" | "no" | "some" | "lone" | "one" | "sum"

binOp = "||" | "or" | "&&" | "and" | "&" | "<=>" | "iff" | "=>" | "implies"
         | "+" | "-" | "++" | "<:" | ":>" | "." | "<<" | ">>" | ">>>"

arrowOp = ["some"|"one"|"lone"|"set"]? "->" ["some"|"one"|"lone"|"set"]?

compareOp = "=" | "in" | "<" | ">" | "=<" | ">="

unOp = "!" | "not" | "no" | "some" | "lone" | "one" | "set"
        | "seq" | "#" | "~" | "*" | "^"

block = "{" {expr} "}"
```

```
blockOrBar = block
blockOrBar = "|" expr

name = ["this" | ID] {["/" ID]}

ref = name | "univ" | "Int" | "seq/Int"
```

Bibliografia

[1] Alloy. Alloy comunity. `http://alloy.mit.edu/alloy4`.

[2] Alloy. Java API. `http://alloy.mit.edu/alloy4/api.html`.

[3] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2):1–79, 2008.

[4] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[5] Stefan Brass, Christian Goldberg, and Alexander Hinneburg. Detecting semantic errors in SQL queries. Technical report, 2003.

[6] BYACC/J. Byacc/j java extension. `http://byaccj.sourceforge.net/`.

[7] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-oriented queries made easy. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 78–133. Springer, 2007.

[8] Oege de Moor, Mathieu Verbaere, and Elnar Hajiyev. Keynote address:.QL for source code analysis.

[9] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[10] Flex. Flex, a fast scanner generator. `http://dinosaur.compilertools.net/flex/index.html`.

[11] Pierre Genevès and Nabil Layaïda. A system for the static analysis of XPath. *ACM Trans. Inf. Syst.*, 24(4):475–502, 2006.

[12] Pierre Genevès and Nabil Layaïda. Deciding XPath containment with MSO. *Data Knowl. Eng.*, 63(1):108–136, 2007.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM.

[15] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[16] JFlex. JFlex - the fast scanner generator for java. `http://jflex.de/`.

[17] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[18] Pierre Kelsen and Qin Ma. A lightweight approach for defining the formal semantics of a modeling language. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 690–704, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Laks V. S. Lakshmanan, Ganesh Ramesh, Hui Wang, and Zheng Zhao. On testing satisfiability of tree pattern queries. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 120–131. VLDB Endowment, 2004.

[20] Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[21] Semmle Ltd. Semmle ltd. - company website with free downloads, documentation, and discussion forums. `http://semmle.com/semmlecode/`.

[22] Manizheh Montazerian, Peter T. Wood, and Seyed R. Mousavi. XPath query satisfiability is in ptime for real-world DTDs. In Denilson Barbosa, Angela Bonifati, Zohra Bellahsene, Ela Hunt, and Rainer Unland, editors, *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 17–30. Springer, 2007.

[23] João Saraiva. *Especificação e Processamento de Linguagens*. Universidade do Minho, 1995.

[24] Thomas Schwentick. XPath query containment. *SIGMOD Rec.*, 33(1):101–109, 2004.

[25] Semmle. .QL grammar. `http://semmle.com/semmlecode/ql-language-reference/syntax/grammar/`.

[26] Damien Sereni, Pavel Avgustinov, and Oege de Moor. Adding magic to an optimising datalog compiler. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 553–566, New York, NY, USA, 2008. ACM.

[27] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, New York, NY, USA, 1988.

[28] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. Symbolic query exploration. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 49–68. Springer, 2009.

[29] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[30] Yacc. Yacc: Yet another compiler-compiler. `http://dinosaur.compilertools.net/yacc/index.html`.