Daniel Rodrigues Pacheco Murta

A comparison between DSLs and GPLs for the implementation of unidirectional and bidirectional transformations

Novembro de 2014

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel Rodrigues Pacheco Murta

A comparison between DSLs and GPLs for the implementation of unidirectional and bidirectional transformations

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Alcino Cunha
Marco Perestrelo

Novembro de 2014

ACKNOWLEDGEMENTS

The Master's Degree is most certainly the culmination of a long path since I started as an undergraduate student in Computing Engineering at Universidade do Minho. The academic life comes to an end. In the beginning I had the opportunity to discover most of the areas in which Computer Science unfolds, which afterwards led me to choosing Formal Methods as one of the specialisation areas to opt for in the Master's first year.

I strongly believe that the future of Software Engineering is directly aligned with the MDE cornerstones: models and transformations. Nowadays, the industry tackles problems by investing in programmers inordinately. This happens because software is not modelled correctly. A much reasonable strategy should be to model more, so that problems are clearly understood and then apply well-known and fully experienced patterns to achieve intermediary goals, until the final result. As the idiom puts it "the wheel should not be reinvented" and, for that, one must apply known "recipes", which are encompassed by MDE as transformations.

My first contact with models as central artefacts of Software Engineering was precisely in one of the modules included in the Formal Methods specialisation I applied for. I always found models to be of high interest because they clarify what is by default blurry. My supervisor, Professor Alcino Cunha, was the lecturer of these classes and this is the first reason why I should thank him. He presented me with the basic concepts of modeling when I was attending his classes and, now, in this dissertation, the path leaned towards transformations in part because of him. I am very satisfied with the way my Master's Degree turned out to be because, in the first year I strengthened my knowledge on models and in the second year, during this dissertation, I had the opportunity to delve into transformations. Thus, I can say I covered pretty well the two main building-blocks of Software Engineering. This was, in part, because of Professor Alcino. Furthermore, he had always known how to guide me in this not so straight process and I am deeply thankful for that.

In addition, I would like to thank the Insurance company which not only gave me the opportunity of working on this thesis in an industrial environment but also gave me my first initiation into the software industry world. I consider this to be a decisive step in my professional career undoubtedly. In particular, I would like to express my very great appreciation to Marco Perestrelo, firstly, for accepting being my co-supervisor and lastly for always supporting the decisions taken in this thesis project. I would also like to express my gratitude to Abílio Pinto, for being a pleasure to work with and for the assistance provided. It was greatly appreciated.

Last but not least, I would like to thank my family and close friends for the unceasing support through, not only this thesis, but during all the periods of my life.

A COMPARISON BETWEEN DSLS AND GPLS FOR THE IMPLEMENTATION OF UNIDIRECTIONAL AND BIDIRECTIONAL TRANSFORMATIONS

Model-Driven Engineering (MDE) is a software development paradigm based on two building concepts: models and transformations. In an early stage, software is abstracted by models, which gradually evolve into the final product by the application of transformations. This dissertation focus essentially on transformations. Initially, a general categorisation of transformations is given by presenting the several aspects on which they can be distinguished. One of those aspects is *directionality*, where transformations are split into one of two categories: *unidirectional*; or *bidirectional*.

Then two distinct problems are presented, for which a transformation is proposed as a solution for each. For the first problem, a unidirectional transformation is recommended and for the latter a bidirectional transformation is recommended. For each transformation, two different implementations are conceived and discussed. One of the implementations is achieved by recurring to a Domain-Specific Language (DSL) specifically tailored for the specification of the transformation in case, and the other implementation is materialised by recurring to a General-Purpose Programming Language (GPPL).

The main aim of this thesis is to compare between DSLs and GPPLs as possible alternatives to specify model transformations. This comparison is done by taking into consideration both the unidirectional and bidirectional case scenarios, and by having as comparison guidelines performance and maintainability.

COMPARAÇÃO ENTRE DSLS E GPLS PARA A IMPLEMENTAÇÃO DE TRANSFORMAÇÕES UNIDIRECIONAIS E BIDIRECIONAIS

MDE é um paradigma de desenvolvimento de software assente em duas pedras basilares: modelos e transformações. Numa primeira fase, o software é abstraído através de modelos, que depois evoluem gradualmente na direcção do produto final pela aplicação de transformações. Inicialmente é apresentada uma categorização geral do conceito de “transformação” no âmbito do MDE, apresentando os vários fatores nos quais as transformações se distinguem umas das outras. Um destes factores é designado de *direcionalidade*, segundo o qual as transformações podem ser unidirecionais ou bidirecionais.

Esta tese apresenta dois problemas distintos, para os quais uma transformação é proposta como solução para cada. No caso do primeiro problema, recomenda-se uma transformação unidirecional, enquanto que para o segundo problema é proposta uma transformação bidirecional. Para cada transformação proposta, duas implementações diferentes são concebidas e discutidas. Uma das implementações é conseguida recorrendo a uma linguagem de domínio específico talhada especificamente para a implementação da transformação em causa, e a outra implementação é obtida através do desenvolvimento de um programa através de uma linguagem de programação genérica.

O principal objetivo desta dissertação é comparar entre as linguagens de domínio específico e as linguagens genéricas como duas alternativas na especificação de transformações entre modelos. Esta comparação é feita tendo em consideração os cenários de estudo unidirecional e bidirecional, sendo que os termos de comparação usados foram a noção de *performance* e manutenção das implementações.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

AGG	Attributed Graph Grammar.
ATL	ATLAS Transformation Language.
BDO	Business Domain Object.
CIM	Computation-Independent Model.
DSL	Domain-Specific Language.
DSML	Domain-Specific Modeling Language.
EISC	European Insurance Software Company.
GPL	General-Purpose Language.
GPML	General-Purpose Modeling Language.
GPPL	General-Purpose Programming Language.
IDE	Integrated Development Environment.
MBE	Model-Based Engineering.
MDA	Model-Driven Architecture.
MDD	Model-Driven Development.
MDE	Model-Driven Engineering.
MDSE	Model-Driven Software Engineering.
MIC	Model-Integrated Computing.
OCL	Object Constraint Language.
OMG	Object Management Group.
OWL	Web Ontology Language.

PIM Platform-Independent Model.

PSM Platform-Specific Model.

QVTO Query/View/Transformation operational.

QVTR Query/View/Transformation relational.

SQL Structured Query Language.

UML Unified Modeling Language.

VBDO View Business Domain Object.

W3C World Wide Web Consortium.

Part I

INTRODUCTORY MATERIAL

1 INTRODUCTION

This thesis was developed in the context of an European Insurance Software Company (EISC). This company is specialised in the insurance industry, offering a vast software solution for its clients to manage the insurance products they make available on the market and their affiliated clients. The software solution sold by the company had its development first initiated several years ago, enough for it to be considered, nowadays, a so-called *legacy system*. In order to surpass this, the company has opted to perform a modernisation on its system by executing a migration strategy, guided by MDE concepts.

The final goal of the company is to have a development structure entirely based on the MDE principles. Rather than starting to code right away, one will create models of the system, which will then be used as input to generate automatically the actual code of the system. In spite of being a bold objective, adopting these automation procedures can be very rewarding for a software company because, not only does it accelerate the development process, but also because generating code implies reducing the probability of existing "bugs" on it, since part of what used to be the developers' work is shifted to generators, and "bugs" mostly exist due to human error.

For this goal to be achieved, several steps need to be addressed. First of all, for the models of the system to exist, previously there needs to exist at least one modeling language to specify those models. Such languages can either be a General-Purpose Modeling Language (GPML), that basically can be applied to model any domain whatsoever, or they can be a Domain-Specific Modeling Language (DSML), that was created to target a specific domain. In the case of this EISC, several DSMLs were implemented with the purpose of modeling different aspects of the system. One of those languages was designed to model business objects in particular, in this case related to the insurance domain, such as policies, accidents, etc. This language was called the Business Domain Object (BDO) language, and has the .bdo extension. The idea behind the BDO language is to have a simple way of specifying business entities, which afterwards are used to generate their respective pieces of code, being Java classes, interfaces, etc. Nevertheless, there are two issues related to this language:

1. Global Perception

   Each BDO file is a specification of a certain business object. One of the features made available by the BDO language is *cross references*. Thus, when specifying a business object, it is actually possible to state, for instance, that the object has an attribute which type points to another business object defined externally, in another BDO file. This is a cross reference. The problem about this feature is that it does not provide a global perception of the cross references between all the BDOs. Thus, by the time the system evolves by specifying more and more business objects, at some point, the density of the "network" between the objects (resulting from the cross references) can be quite impressive and, therefor, a very useful information, which is not perceived. Nevertheless, at the same time, the BDO files for themselves do not express this network perception in any way, since each holds a single business object.

2. Suitability for insurance experts

   Being a DSL created specifically to model business objects, it would be natural to assume that the BDO language should be an easy one to use for experts on the business in question, in this case the insurance business. Nonetheless, in spite of being possible to specify typical facts about business objects for an insurance expert, such as the fact that a policy must carry a holder of that policy, an expiration date, etc, specifying a business object with the BDO language also involves having to deal with other concepts that an insurance expert is not so much acquainted with. For instance, it is possible to define interfaces and inheritance assertions which are object-oriented programming concepts. This is a problem because, on the one hand, insurance experts are not able of specifying business objects themselves without recurring to the assistance of the company's consultants for the programming related specifications and, on the other hand, the consultants will also have problems performing the same task because, in this case, they lack the insurance expertise.

In order to solve these two problems, one solution for each one was proposed. Taking into consideration the Global Perception problem, the solution proposed was defining a transformation that takes all the current existing BDOs as parameter and merges them in a single model, which can be graphically visualised, so that the network perception is easily assimilated. The output target language chosen was World Wide Web Consortium (W3C)'s Web Ontology Language (OWL) World Wide Web Consortium (2004). Despite being a DSL for ontologies, the reason for choosing OWL specifically, and not another language, was the fact that Protégé[1] is a free graphical visualiser for models specified in OWL, which was precisely what was necessary. This is a unidirectional transformation because it is not our interest to update the OWL files after them being generated from the BDOs - the OWL files are generated and then simply given as input to Protégé for the visualisation. Bearing this in mind, the signature of this transformation should be something like:

$$BDO^* \rightarrow OWL$$

As for the suitability issue, the problem lies in the fact that the BDO language encompasses some aspects other than the ones typically related with the insurance domain. Those aspects are the ones which the insurance experts are not able to specify individually. The solution for this problem was

1 http://protege.stanford.edu/

defining a *view* for the BDOs, which only presents the concepts in the BDO that the insurance experts are able to deal with. The view can be regarded as a submodel of its corresponding BDO, in the sense that every information present in the view is replicated in the BDO, with a different syntax. The final purpose of the views is to let the insurance experts deal with them, editing as they wish, and then automatically propagate the changes to their corresponding BDOs, so that the experts only have to deal with the domain aspect of the BDOs. This solution involves defining a bidirectional transformation: one direction to generate the BDO's corresponding view; and the other direction to refactor the original BDO according to the possibly changed view. The following expression denotes the type of the bidirectional transformation

$$BDO \leftrightarrow VBDO$$

where View Business Domain Object (VBDO) corresponds to a language implemented in order to specify the BDOs' views.

To sum up, this thesis can be resumed in the definition of two transformations. In order to define these transformations, two different approaches were followed for each one: in one approach, a DSL was used to specify the transformation; and in the other a General-Purpose Language (GPL) was used with the same purpose. The final goal was to establish a comparison between the approaches followed for the implementations of both transformations. Thus, in the end, four implementations were produced: two for each transformation. The DSLs chosen for defining the transformations were MDE standards, namely, the ATLAS Transformation Language (ATL) Jouault and Kurtev (2006) for the unidirectional transformation, and the Query/View/Transformation relational (QVTr) language Object Management Group (2011) for the bidirectional transformation. The GPL used for specifying both transformations was Java.

The following chapter (Chapter 2) presents the state of the art of this thesis. Firstly, the topic of legacy systems is addressed, followed by a brief presentation of the MDE concept. In the MDE presentation, a brief description of software transformations is given, which serves as introduction to a more detailed section only dedicated to that point. In that section, unidirectional and bidirectional transformations are defined and the focus is in the several approaches accepted nowadays by the community to specify these two types of transformations. Some examples of software transformations are given in that section, namely, one Xtend Eclipse (2013a); Bettini (2013) transformation and one Boomerang Bohannon et al. (2008) transformation. Finally, the chapter ends with a section dedicated to languages, where DSLs and GPLs are defined and compared with an use case.

The core of this dissertation begins in Chapter 3, where the two problems addressed in this thesis are clearly explained. Chapter 4 addresses the solutions proposed in order to solve the two problems described previously. The solution for each problem can be resumed in the specification of a transformation, as mentioned before. Bearing this in mind, for each problem, firstly the solution is explained and then the two implementations for that solution are presented and compared in the end. ATL and QVTr were the DSLs used to specify the transformations as mentioned before and, consequently, they are introduced further, so that the implementations are easily understood.

Chapter 5 is reserved for the conclusions and ends by identifying possible directions for future work.

2

STATE OF THE ART

This chapter is reserved for the state of the art of this thesis. “Legacy systems” is the first topic to be addressed. Then, MDE is presented as a possible solution to modernise a legacy system. Two concepts are paramount in MDE: models and transformations. The section which follows the MDE presentation is reserved solely for addressing transformations, taking into consideration both unidirectional and bidirectional transformations. Finally, the chapter ends with a section dedicated to languages, by presenting and comparing DSLs and GPLs.

2.1 LEGACY SYSTEMS

Ian Sommerville states in his famous ”Software Engineering” book (Sommerville, 2007, Chapter 26) the following:

> “Some organisations still rely on software systems that are more than 20 years old. Many of these old systems are still business-critical. (...) These old systems have been given the name legacy systems.”

Legacy systems represent a real problem for some organisations. On the one hand, they are and always were their revenue generator, on the other hand, keeping them stale as they are today will not make that trend sustainable. Legacy systems have been developed in the past using languages/technologies nowadays considered obsolete. This represents, for itself, a potential problem in the future because obsolete technologies always reach a point where their support ceases to exist, which is concerning for companies. In addition, for a “fresh” software engineer, who has just finished his/her academic degree, being assigned the task of programming using such tools can be very demanding, considering the steep learning curve which he/she will have to overcome. This is true because object-oriented programming is the standard these days, like Figure 1 shows, and obsolete technologies normally do not follow current standards. Furthermore, legacy systems typically display some issues concerning documentation, which makes the acquaintance to them even harder for someone unfamiliarised Sommerville (2007). These are the main reasons why legacy systems are not sustainable in the long term: a person who is actually capable of “diving into the code” is most likely only one who have been implementing it from the very beginning or someone with some expertise in the language used, since the context knowledge transfer is definitely, not a straightforward process to go through. People with some knowledge of the legacy system's used technologies tend to decrease to zero in the future. This is a very concerning fact for a company, because it basically means that it becomes dependent on its senior collaborators for its business to keep making profit.

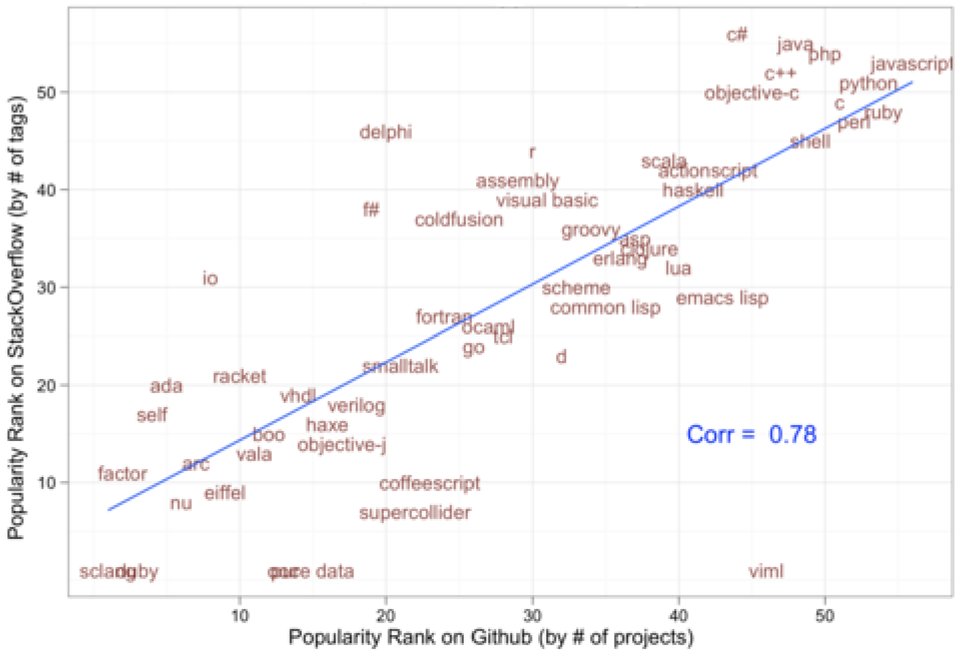


Figure 1.: Programming languages popularity Yusuf

System disruption is rarely the solution for this problem. The cost of implementing everything all over again and abandoning the current system is simply unbearable in useful time. This is true because legacy systems are enormous pieces of code, with millions of lines of code, configuration files, etc. Furthermore, one cannot simply leave behind a well behaved code, considering both safety and functionality as parameters, which took years to implement and perfect and, at the moment, is being used by several clients. At the end of the day, what really matters to companies is the fact that, for the moment, their systems work as they are supposed to. Being able of reproducing/implementing that same behaviour all over again from scratch, using up to date technologies, is an endeavouring and mainly time consuming task.

A progressive migration strategy is usually the solution opted for in order to modernise a legacy system. The difference between migrating and starting over is the fact that migrating means re-implementing a system having the legacy one as basis, whereas starting over means repeating the entire development process from the beginning, including requirements analysis, development, etc. The main parameters based on which one should establish a comparison between migration and disruption are "time consumption" and "complexity", as stated in Table 1:

	Time Consumption	Complexity
Disruption	✗	✓
Migration	✓	✗

Table 1.: Comparative analysis between Disruption and Migration as strategies to modernize legacy systems

As long as one knows what the system is expected to accomplish, modernising legacy systems by re-implementing them from the beginning can be considered an easy task in terms of complexity. The problem with this approach is mainly the time needed to execute it. The task of re-compiling the entire requirements of the system and then implementing is very time-consuming for legacy systems. The opposite happens with migration. Migration is a matter of identifying two platforms (the source and the target platforms) and the mapping function between them. Having specified the mapping, then migration is "simply" about applying the mapping, which can sometimes be achieved with automation. If, in fact, one is able of performing the mapping automatically, then the migration is much faster than the disruption approach. The problem with migration is precisely the mapping function - generally, it is not immediate the specification of such function. When one has two migrate, for instance, a J2E application to the .NET platform, the mapping is not necessarily hard to accomplish, because the two platforms involved share the same nature - both are object oriented. However, when the source and target platforms are not similar in any way, migration can be very difficult to perform. This is unfortunately what usually happens with legacy systems' migration.

2.2 MODEL-DRIVEN ENGINEERING

"*Model-driven engineering* (MDE) is a discipline in software engineering that relies on models as first class entities and that aims to develop, maintain and evolve software by performing model transformations" (Mens and Van Gorp, 2006, Chapter 1). Sometimes, MDE is also referred as Model-Driven Software Engineering (MDSE), which is the acronym used by Ian Sommerville, which can be defined "as a methodology for applying the advantages of modeling specifically to software engineering activities" (Brambilla et al., 2012, Chapter 2.1). Generally speaking, defining a methodology implies specifying the following aspects: concepts; notations; process; and tools.

In the context of MDE, the core concepts are models and transformations, as emphasised by Equation 1 Brambilla et al. (2012):

$$\textbf{Models} + \textbf{Transformations} = \textit{Software} \tag{1}$$

Models are usually defined concisively as "system abstractions" Czarnecki and Helsen (2006), in the sense that they represent a certain reality, just focusing on the important information for the observer. Thus, models are naturally very useful because they present the reality in a clearer way and, at the same time, they are supposed to behave accordingly to the system, which is useful to test certain scenarios without facing the risks of doing so in reality. For example, “civil engineers create (...) models of bridges to check structural safety since modeling is certainly cheaper and more effective than building real bridges to see under what scenarios they will collapse.” (Czarnecki and Helsen, 2006, Chapter 2). One important fact to know about models is they always conform to a specific (meta-)model, like Figure 2 indicates.

Transformations are the other core element of MDE. Transformations are actually defined at the meta level, and then applied at the level below, upon instances that conform to those (meta)models Brambilla et al. (2012). Applying a transformation can be visualised as moving from a diagram like the one in Figure 2 to a similar one. Section 2.3 is entirely reserved to address the topic of transformations, for their significant part in this thesis.

Having both models and transformations been specified, transformations are applied over models, new models are generated and so on, until, eventually, the final result is reached (the software). This final result can be itself a model, which is to be executed by an interpreter specially designed for that purpose, or it can be, directly, a specific platform’s runnable code, such as Java, for instance.

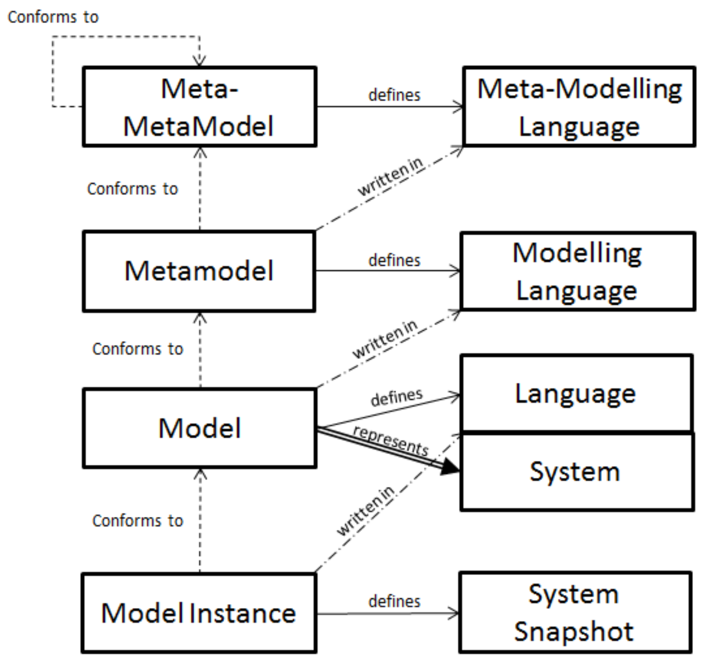


Figure 2.: Model abstractions scheme

As Figure 2 indicates, in order for a model (or model instance) to be specified, at least one language is necessary. Just like one needs a programming language to write a program, in the context of MDE, one needs a modeling language to specify a model. Languages are the notation building block of the MDE methodology. When opting for modeling languages, in this case, two possibilities are always available: GPLs; or DSLs. Like the name suggests, GPLs are supposedly applied to any domain whatsoever. For example, Java is a GPPL and Unified Modeling Language (UML) Object Management Group (b) is a GPML. On the other hand, DSLs are languages specifically designed to handle a particular domain. For example, if one has the need of specifying graphs, creating a language to define graphs would be a possible solution. That language would be a DSL.

Observing Figure 2 once again, four abstraction levels are displayed. This is not always necessarily true. Metamodeling is basically the process of "modeling a model". While, in theory one could define infinite levels of metamodeling, it has been shown, in practise, that metamodels can be defined based on themselves Brambilla et al. (2012), just like with the Meta-Metamodel in Figure 2. Thus, just like four abstraction levels are presented in this case, three could also be - in that case, the Metamodel would conform to itself, closing the chain.

Finally, two parts remain to define the MDE methodology: the process; and the tools. Equation 1 does not tell what kind of models (and in which order, at what abstraction level, etc.) need to be defined depending on the kind of software to produce. That's when the model-driven process of choice comes to play Brambilla et al. (2012). Model-Driven Architecture (MDA)[1] Frankel (2003), Model-Integrated Computing (MIC) (Gokhale et al., 2002, Chapter 3) Sztipanovits and Karsai (1997) and Software Factories Greenfield and Short (2004) are instantiations of MDE where the process is clearly defined. Section 2.2.1 exposes MDA. Finally, an appropriate set of tools/frameworks is also needed to make MDE feasible in practise. The tools are mainly necessary to define the models and transformations as well as to compile or interpret in order to execute them and produce the final software artefacts. Bearing this in mind, the tools typically used are Integrated Development Environment (IDE)s for language usage or creation to specify the models and transformations, and compilers or interpreters for the execution.

2.2.1 *Model-Driven Architecture*

In the previous section, MDE was presented as a methodology or a paradigm where model-driven techniques are applied towards software development. Apart from that, MDE can be regarded as an abstraction, for which instantiations can be derived. This is the case with the MDA, the particular vision of Model-Driven Development (MDD) proposed by the Object Management Group (OMG) and thus relies on the use of OMG standards" (Brambilla et al., 2012, Section 2.2). In order not to get confused with acronyms, Figure 3 is presented:

1 http://www.omg.org/mda/specs.htm

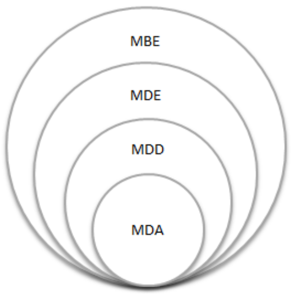


Figure 3.: Model-driven acronyms diagram Brambilla et al. (2012)

- Model-Based Engineering (MBE) - Engineering paradigm oriented (not driven) by models. Using models simply as blueprints for developers would be considered MBE, for instance;
- MDE - Engineering paradigm driven by models, encompassing aspects other than development, such as reverse engineering, model-based testing, etc;
- MDD - "Development paradigm that uses models as the primary artifact of the development process" (Brambilla et al., 2012, Section 2.2) (solely);

Thus, MDA can be regarded as a subset of MDD. This means that the concepts are the same - models and transformations. The only aspect added by MDA is the specification of three particular levels of abstraction, so that one can categorise more clearly the models created during the development process (Brambilla et al., 2012, Chapter 4.2):

- "Computation-Independent Model (CIM) - The most abstract modeling level, which represents the context, requirements, and purpose of the solution without any binding to computational implications. It presents exactly what the solution is expected to do, but hides all the IT-related specifications, to remain independent of it and how a system will be (or currently is implemented). The CIM is often referred to as a business model or a domain model (...);
- Platform-Independent Model (PIM) - The level that describes the behaviour and structure of the application, regardless of the implementation platform. (...) The PIM exhibits a sufficient degree of independence so as to enable its mapping to one or more concrete implementations platforms;
- Platform-Specific Model (PSM) - Even if it is not executed itself, this model must contain all required information regarding the behaviour and structure of an application of a specific platform that developers may use to implement the executable code.

In MDA, the core activity is the conceptualisation phase, which is the way analysis is conducted: first, requirements are codified as PIM (or even CIM), an then the actual

> design of the application must be performed. This typically produces a PSM. In turn, the PSM is transformed into the running code, possible requiring additional programming (partial generation)."

The standards used when it comes to the languages to specify the models and transformations are not fixed by MDA. In spite of the OMG being responsible for languages such the UML Object Management Group (b), adopting MDA does not imply adopting UML as well. MDA is neutral with respect to this decision.

2.3 TRANSFORMATIONS

Transformations are, alongside models, the core concept of the MDE methodology, as stated in Section 2.2. Models abstract the system being modelled in different manners, depending on the level of abstraction projected for each model. Transformations are responsible for the transitioning between models, i.e. for switching between levels of abstraction as the modeller pleases. As a metaphor, one can regard MDE as a microscope, where models are the several objectives available, and the act of applying a transformation corresponds to shifting between objectives, to increase or decrease the resolution/abstraction. These are the main applications of transformations: "synthesis of a higher-level, more abstract, specification (...) into a lower-level, more concrete one" Mens and Van Gorp (2006); and reverse engineering, which "is the inverse of synthesis and extracts a higher-level specification from a lower-level one" Mens and Van Gorp (2006).

The first distinction to be made regarding transformations generally is between model and program transformations. The distinction is not as clear as one would wish because the boundaries between them vary according to how one defines "model" and "program". A model transformation is any transformation where at least one of its "sides" (input or output) includes models. Bearing this mind, parsing can be regarded as a model transformation, as some text file is read and transformed into some digital representation of the same in memory (model). Taking this definition into account, for those who actually consider programs to be models as well, because they take them as simple versions of the machine-level code, everything is a model transformation. For those who do not include programs in the models universe, program transformations are transformations between programs possibly specified in distinct programming languages. At the same time, some people actually believe that the difference between model and program transformations lies in the frameworks for operating with those transformations. According to Mens and Van Gorp (2006), "perhaps the most important distinction between the current approaches to program transformation and model transformation is that the latter has been targeted for a particular set of requirements including (...) traceability (...) and multi-directionality of transformations. While these requirements could also be the subject of program transformation approaches, they are typically not considered by program transformation systems."

In the context of MDE, model transformations are the ones to consider. Model transformations can be classified according to several aspects, most of them being directly related with the input(s) and output(s) the transformations operate with:

- Output Czarnecki and Helsen (2006)
  - Model to Model Transformations' output is a model. In this case, the transformation operates in-memory both with the source and target models;
  - Model to Text Transformations' output are simply strings. Thus, these transformations operate by pushing strings into a text stream;
- Cardinality Czarnecki and Helsen (2006)
  - Transformations mapping among multiple models are typically called n-way transformations. They can be one-to-many, many-to-one, etc. In the case of the many-to-one transformations, these ones are usually called model weaving or model merging also;
- Source/Target Metamodels Mens and Van Gorp (2006)
  - Endogenous transformations are mappings between models conforming to the same metamodel. These transformations are also called rephrasings. Endogenous transformations can be classified even further in terms of the number of models involved:
    - If there is only one model involved, i.e. the source and the target models are the same, the transformation is in-place;
    - If the there is at least one input and output models, the transformation is out-place.
  - Exogenous transformations are transformations between models conforming to different metamodels. These ones are always out-place.
- Source/Target Abstraction Levels Mens and Van Gorp (2006)
  - Horizontal transformations are applied between source and target models at the same level of abstraction. Refactoring is an example of an endogenous horizontal transformation;
  - Vertical transformations are applied between different levels of abstraction. Code generation synthesis is an example of a vertical transformation.
- Directionality Czarnecki and Helsen (2006)
  - Unidirectional transformations are executed in one direction only, as the name suggests. In this case, the target model is produced based on the source model, invariantly;
  - Bidirectional transformations are executed in two directions, so the target and the source models are able of switching parts, depending on the direction chosen. Some languages allow the specification of bidirectional transformations with single rules, that serve both directions, whereas with other languages one has to specify two unidirectional transformations. Section 2.3.2 addresses bidirectional transformations with more depth.

To finalise this topic, a brief categorisation on how to specify model transformations remains to be given. So far it has only been presented the concept of model transformations and their several derivations. Nevertheless, considering the goals stated for this thesis in Section 1, the actual specification of transformations is a matter of high importance. There are approaches for defining unidirectional transformations and approaches for defining bidirectional transformations. Bearing this in mind, the categorisation to be given will be divided between unidirectional and bidirectional transformations, by discussing this topic both in Sections 2.3.1 and 2.3.2, respectively. Besides this, these sections also address unidirectional and bidirectional transformations with more depth, due to their considerable importance in this dissertation.

2.3.1 *Unidirectional*

Unidirectional transformations are functions of type $S \longrightarrow T$, that accept models conforming to metamodel S (the source) and produce models conforming to T (the target). Unidirectional transformations are transformations that can be applied in one direction only and invariantly, which is directly defined by the way the transformation itself has been specified. There are several approaches one can use to specify unidirectional transformations. Czarnecki and Helsen (2006) suggested a categorisation for these approaches, and the following is based on it.

The "easiest" approach to understand is normally the Direct-Manipulation approach, where one basically defines the transformation by recurring to some GPPL, such as Java. This approach is used for model to model transformations - the correspondent one for model to text transformations is called the Visitor-Based approach. In both approaches very little is given to the developers. Normally the developers only have an object-oriented way of navigating along the input model and the rest has to be coded manually. The only reason why this approach can be regarded as the "easiest" is because the developer does not have to learn any concepts about it, since most of the work is expected to be delivered by him/her. Thus, as long as the developer knows some GPL, the Direct-Manipulation approach is possible. However, in the perspective of the actual workload the developer may have to deal with, maybe this approach is worse than others that implement some handy concepts like rule scheduling, tracing, etc Czarnecki and Helsen (2006). For instance, Structure-Driven approaches for model to model transformations, normally implement rule scheduling, so that the developer only has to concern with providing the actual transformation rules. The downside of these approaches is the overhead associated with getting acquainted with these concepts and their implementation supplied by the framework. The Structure-Driven approach is based on two phases, where in the first one the structure of the target model is created and, in the final one, that structure is mounted, by setting attributes and references. OptimalJ[2] framework provides an approach like this to specify transformations.

Template-Based approaches are another possibility to consider, both for model to model and model to text transformations. This approach is usually more common with model to text approaches because

2 http://www.nomagic.com/getting-started/our-philosophy.html?id=11

templates suit very nicely text as output. The template is basically the output of the transformation with some gaps containing metacode to access information from the source model. Thus, the gaps exist to fill in the template with the info that is dependent on the source models being passed as input to the transformation. In the case of model to model transformations, templates are normally expressed in the concrete syntax of the target language. An example of a framework which allows the template approach is Xtend Eclipse (2013a). Later, a transformation using this template-based approach is given as an example.

The Operational approach is another one which should be considered. Normally one says that the Operational approach "competes" with another one called the Relational approach because their paradigms are somehow contrasting. Operational approaches focus on how to map from source to target, so one is expected to specify imperative rules stating the steps in order to do so. On the other hand, Relational approaches focus on the what aspect, by specifying when the source and the target models are synchronised. It is important to mention that of these two approaches, the only one targeted for unidirectional transformations is the Operational one: the only reason for presenting the Relational approach at this moment is precisely because of its contrasting paradigm in comparison with the Operational approach. Query/View/Transformation operational Object Management Group (2011) is a specific framework implementing the operational approach.

The last approach to be mentioned is one drawn on the theoretical work on graph transformations, named the Graph-Transformation-Based approach. This approach operate with rules composed by one left-hand side (LHS) and one right-hand side (RHS), and the transformations are executed by matching the rules' left-hand sides in the source model and replacing them in the target model by the respective right-hand sides. The particularity of this approach is the fact that the rules are specified with graphs. Attributed Graph Grammar (AGG) Taentzer (2003) and Fujaba[3] are examples of frameworks implementing this approach.

Finally, in spite of having presented all these approaches separately, some frameworks opt to merge some of them, resulting in Hybrid approaches. ATL, for example, is the Eclipse standard for unidirectional model transformations and provides a fully declarative mode, an hybrid mode and a fully imperative mode.

The following sections present two simple examples of unidirectional transformations, each one implemented following one of the approaches mentioned above.

Xtend example

This section presents an example of a transformation specified using one of the approaches mentioned above. In this case, the template-based approach was the one chosen. Figure 4 shows a model expressed as a UML class diagram. The model represents a simplified version of object-oriented classes. `Classes` contain `Attributes` and `Comments` and may extend another `Classes` (one at most). The `children` association, for each `Class`, encompasses every other `Class` which extends that

3 `http://www.fujaba.de/`

Class. The Comment entity represents a special type of comments that exists for javadoc tools. Classes can be marked as abstract.

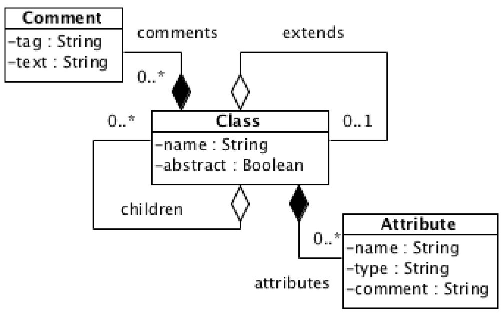


Figure 4.: Class metamodel

In this case, the transformation to specify is one which generates the actual Java code associated to a Class. Thus, the transformation is a model to text one. In order to specify the transformation, the template-based approach was chosen, as mentioned before, and the framework used in order to do so was Xtend. Listing 2.1 illustrates the transformation.

```
def transform(Class c) '''
        <<IF c.comments!=null && c.comments.size!=0>>
            /**
            <<FOR cm : c.comments>>* @<<cm.tag>> <<cm.text>><<ENDFOR>>
            */
        <<ENDIF>>
        public <<IF c.abstract>>abstract <<ENDIF>>class <<c.name>> <<IF c.extends!=null>>
            extends <<c.extends.name>> <<ENDIF>>{
            <<FOR a : c.attributes>>
                <<IF a.comment!=null && a.comment.length!=0>>//<<a.comment>><<ENDIF>>
               private <<a.type>> <<a.name>>;
            <<ENDFOR>>

            <<IF !c.abstract>>public <<c.name>> () {}<<ENDIF>>

            //Getters
            <<FOR a : c.attributes>>
                public <<a.type>> get<<a.name.toFirstUpper>> ()
                { return <<a.name>>; }
            <<ENDFOR>>

            //Setters
            <<FOR a : c.attributes>>
                public void set<<a.name.toFirstUpper>> (<<a.type>> a)
                { this.<<a.name>> = a; }
            <<ENDFOR>>
        }
    '''
```

Listing 2.1: Model to text transformation expressed in Xtend

As described in Section 2.3, the template presented in Listing 2.1 is basically the output of the transformation (code in blue, not inside <<>>) with some gaps containing the metacode to access the source model given as input. In Xtend, the gaps are expressed between <<>>.

Considering the `MotorizedVehicle` class presented in Figure 5 as input for the transformation, the result given would be something like the text presented in Listing 2.2.

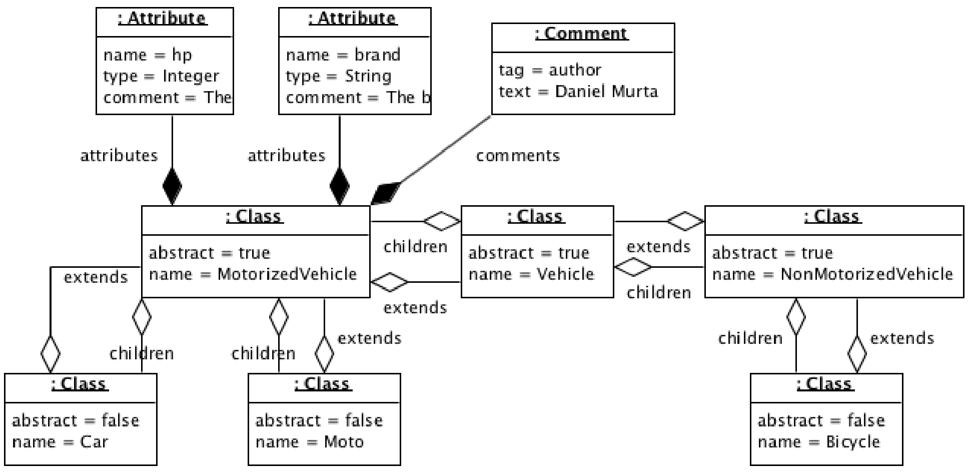


Figure 5.: Instance of the Class metamodel in Figure 4

```
/**
* @author Daniel Murta
*/
public abstract class MotorizedVehicle extends Vehicle {
    //The Brand of the vehicle
    private String brand;
    //The horse power of the vehicle
    private Integer hp;

    //Getters
    public String getBrand ()
    { return brand; }
    public Integer getHp ()
    { return hp; }

    //Setters
    public void setBrand (String a)
    { this.brand = a; }
    public void setHp (Integer a)
    { this.hp = a; }
}
```

Listing 2.2: Result of the transformation in Listing 2.1 applied to the intance in Figure 5

QVTo example

This section encloses the second example to illustrate the approaches for specifying unidirectional transformations. In this case, the operational approach is the one being addressed. The transformation to specify is a model to model one and the language used to specify it was QVTo.

The source model of this transformation is the same as the previous transformation (Figure 4). The target model is specified once again by an UML class diagram (Figure 6), however, in this case, it describes the structure of a `Tree`, which has a `node` (name) and can have multiple `Trees` as branches. There are two types of `Trees` that basically express their nodes' shape: `SquareTrees`; and `CircleTrees`. These two types were defined to be able to map two types of classes: abstract `Classes` result in `CircleTrees` and non abstract `Classes` are mapped to `SquareTrees`.

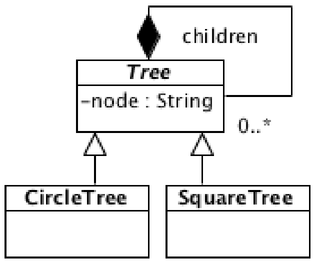


Figure 6.: Class Hierarchy metamodel

Bearing this in mind, the idea of the transformation in this case is to map the class hierarchy that exists in the source model due the the `extends` association, into the tree structure of the target model. Taking this into account, the target model can be regarded as a submodel of the source one, since it presents nothing more than the information regarding to the `extends` association of the source model. The actual QVTo transformation is illustrated in Listing 2.3.

```
transformation ModelToModel(in source:Class, out target:ClassHierarchy);

main() { source.objectsOfType( Class )->map classToTree();}

mapping Class::classToTree() : Tree
disjuncts Class::classToSquareTree, Class::classToCircleTree
{/* No body */}

mapping Class::classToSquareTree() : SquareTree
when {!self.abstract}
{    node := self.name;
     children := self.children->map classToTree();}

mapping Class::classToCircleTree() : CircleTree
when {self.abstract}
{    node := self.name;
     children := self.children->map classToTree();}
```

Listing 2.3: Model to model transformation expressed in QVT Operational

In QVTo, one specifies mappings, which are rules that indicate how the elements of the source model are to be transformed into elements of the target model. For instance, in line 7, we can see a mapping stating that a `Class` should be converted into a `Tree`. Furthermore, it is easy to understand that every `Class` is either mapped to a `SquareTree` or a `CircleTree`: abstract `Classes` result in `CircleTrees`; and non abstract `Classes` result in `SquareTrees`. This happens because the mapping rule `classToTree` in lines 7-9 delegates to the other two mappings - `classToSquareTree` (lines 11-16) and `classToCircleTree` (lines 18-23) - through the `disjuncts` keyword. Each `Class` originates a `Tree`, which `node` equals the class `name`. The tree "connected" structure is guaranteed by the line

```
children := self.children->map classToTree();
```

which sets the `Classes` extending, below in the tree.

Applying this transformation to the instance of the previous case study in Figure 5 would result in the instance of the ClassHierarchy model presented in Figures 7 and 8.

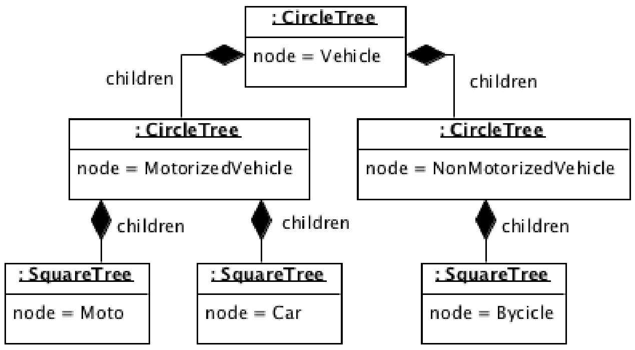


Figure 7.: Result of the transformation illustrated in Listing 2.3 applied to the Figure 5

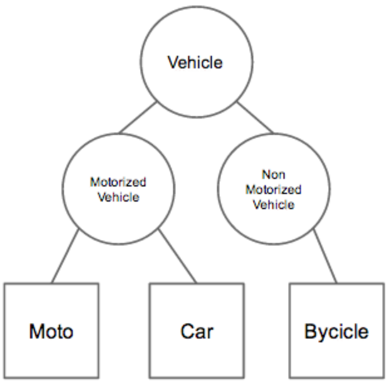


Figure 8.: Alternative view of Figure 7

2.3.2 *Bidirectional*

In the unidirectional paradigm, transformations only occur in one direction - from the source to the target. This happens because updates are only supposed to occur in the source side of the transformation. Thus, every time the source is updated, the target is simply regenerated. A typical example of this scenario is when some information is extremely complex to understand and one needs an abstraction of that information just to be able of displaying it (not updating) in a more user-friendly version. In that case, a unidirectional transformation suffices.

However, when both the source and target of a transformation must co-exist and sometimes evolve independently, the unidirectional panorama no longer serves. In this case, not only source updates must be propagated but also target updates must be. Taking this scenario into consideration, bidirectional transformations are required. Bidirectional transformations have a signature like $S \longleftrightarrow T$, however, in contrast with unidirectional transformations, S and T switch parts between each other depending on the direction the transformation is executed.

Nowadays, there are three standard approaches for defining bidirectional transformations, according to Pacheco (2012). The approaches are presented in Table 2.

$\begin{tikzcd} S \arrow[r, bend left, "to"] \arrow[r, phantom, "\rightleftharpoons"] & T \arrow[l, bend left, "from"] \end{tikzcd}$	Mappings	Involves the specification of two unidirectional transformations: *to* and *from*. None of them considers its source's pre-state to specify the transformation;
$\begin{tikzcd} S \arrow[rr, bend left, "get"] & \rhd & T \\ & T \times S \arrow[ul, "put"] \arrow[ur, "\pi_1"'] & \end{tikzcd}$	Lenses	Involves the specification of two unidirectional transformations: *get* and *put*. *Put* considers its source's pre-state to specify the transformation, which is visible in its signature since its domain is the pair $(T \times S)$ instead of being only T, like with Mappings;
$\begin{tikzcd} & S \times T \arrow[dl, "\pi_1"'] \arrow[dr, "\rhd"] & \\ S & \gtrless & T \\ & S \times T \arrow[ul, "\lhd"] \arrow[ur, "\pi_2"'] & \end{tikzcd}$	Maintainers	Involves the specification of two unidirectional transformations: $\lhd$ and $\rhd$. Both of them consider its source's pre-state to specify the transformation;

Table 2.: Standard bidirecional approaches

Some of these approaches consider pre-state so that they are able of restoring information. Considering the lenses approach Foster et al. (2007) for instance, typically, the target of lenses is an abstraction of the source, meaning it contains less information than the source. Bearing this in mind, when the *get* transformation is applied for a given s, some information is lost on the way. To be able of restoring this information, the *put* transformation attaches the source's pre-state to its domain $(S \times T)$, so that it is able of fetching that lost information that, at the moment, only exists in the pre-state. The *put* is said to be stateful. This is precisely where the approaches vary between each other.

These approaches for themselves are usually insufficient for an end user because, normally, certain properties must be respected for the transformations to make some sense. When such properties are respected, the transformations are said to be *well behaved*, and those are the ones which are normally wanted. One property that must always be respected is Stability, which basically guarantees that if the target/source of the transformation does not change after applying the forward/backward transformation, then the backward/forward transformation shall return the original source/target.

In the case of mappings to be well behaved, normally, the transformations to and from must respect the properties illustrated by Equation FromTo and Equation ToFrom.

$$from(to(s)) = s \tag{FromTo}$$

$$to(from(t)) = t \tag{ToFrom}$$

When these two conditions are respected, one actually realises that S and T must be isomorphic, and *from* and *to* are the bijections witnessing that isomorphism. In this case the mapping is symmetric. When only one of the properties is respected, the mapping is said to be asymmetric, and, in that case, it is also considered well behaved nonetheless. XSugar Brabrand et al. (2008) and biXid Kawanaka and Hosoya (2006) are two examples of languages based on bidirectional mappings.

Considering lenses, for them to be well behaved, once again, only two properties must not be violated. These properties are commonly referred to as GetPut and PutGet for the lenses case solely.

$$put(get(s), s) = s \tag{GetPut}$$

$$get(put(t, s)) = t \tag{PutGet}$$

Focal Foster et al. (2007) and Boomerang Bohannon et al. (2008) are two languages specifically designed to tackle bidirectional transformations with the lenses approach as basis.

Finally, in the case of maintainers, which were introduced by mai, the rules necessary for them to be considered well behaved are expressed by the conjunction of equations BowTie and Compass. However, normally it is also necessary to define a consistency relation R, so that the transformations are able of recovering consistency. This relation basically establishes what sources (s) are related to what targets (t) - $R(s, \triangleright(s, t))$ and $R(\triangleleft(s, t), t)$ - which for lenses and mappings, is defined implicitly. When relation R is indeed defined, the well behaviour rules transform simply into $R(s, t) \Rightarrow \triangleleft(s, t) = s \wedge \triangleright(s, t) = t$. JTL Cicchetti et al. (2011) and QVTr Object Management Group (2011) are two languages designed to address bidirectional transformations following the maintainers approach.

$$\triangleright(\triangleleft(s, t), t) = t \tag{BowTie}$$

$$\triangleleft(s, \triangleright(s, t)) = s \tag{Compass}$$

As a last topic, two aspects of bidirectional transformations remain to be addressed. So far, after having presented the several approaches for defining bidirectional transformations, one fact has been true for each one of them: two unidirectional transformations (forward and backward) always exist for the approach to work ($to/from$, put/get and $\triangleleft/\triangleright$). In spite of being two transformations to specify, bidirectional platforms normally require the specification of a single artefact, through which that two transformations are generated, by applying a bidirectionalisation strategy. This is very useful because it means that one only has to maintain one resource, not two. Nevertheless, it is always possible to specify the two transformations separately, in an ad-hoc fashion. In that case, the developer is responsible for making sure the transformation is well-behaved, while bidirectionalisation techniques provide such property "for free".

Besides this, bidirectional transformations are about propagating updates from source to target and vice-versa. Bearing this in mind, in order to implement a framework following one of the approaches mentioned in table 2, there must be a way of representing updates. Concerning this question, normally two approaches are followed. State-based frameworks take updates simply as the output of a given edit sequence, whereas operation-based frameworks represent updates as a sequence of pre-defined edit actions that may be taken to perform the update, such as creating, deleting, renaming, etc. Thus, the difference between these two approaches is that in the second one, one knows exactly how the source was updated, whereas in the first, one has to observe the output given in order to try to "guess" what has been changed.

This section had the intention of addressing briefly bidirectional transformations. Next is presented an example of a bidirectional transformation using one of the approaches mentioned till now. In this case, the transformation example will be expressed using the Boomerang language.

Boomerang example

Boomerang is "a bidirectional programming language for ad-hoc, textual data formats" (Foster and Pierce, 2009, p. 5). In other words, Boomerang is a bidirectional language whose source and target formats are both restricted to strings. Bearing this in mind, it is correct to say that Boomerang only makes possible the specification of *text* transformations, not model ones.

The approach on which Boomerang is supported is *lenses*, which means that, in order to specify one transformation, the *get* and the *put* components must always be specified. Nevertheless, with Boomerang, only the *get* needs to be specified: the *put* is automatically derived from the *get* specification with a particular bidirectionalisation strategy.

In this case, the example to illustrate Boomerang is related to browsers' history. When one "navigates the Internet" every browser keeps track (unless told not to) of the several web pages one has visited. That tracking results in a compiled list such as the one in Listing 2.4.

```
Sunday, 31/08/2014, 09:41 PM, http://www.seas.upenn.edu
Saturday, 30/08/2014, 09:18 PM, http://www.springer.com
Friday, 29/08/2014, 10:17 AM, http://mei.di.uminho.pt/
Thursday, 28/08/2014, 09:20 AM, http://www.nytimes.com
Thursday, 28/08/2014, 09:10 AM, http://www.publico.pt
```

Listing 2.4: Browser history example

An alternative format for this information could be something like the list in Listing 2.5. In this case, the week days were omitted, which is possibly reasonable because the complete date is already given, and some delimiters were changed: the date and the time are now separated by an hyphen (-); and the web page's URL is preceded by >, instead of a comma (`,`);

```
31/08/2014 - 09:41 PM > http://www.seas.upenn.edu
30/08/2014 - 09:18 PM > http://www.springer.com
29/08/2014 - 10:17 AM > http://mei.di.uminho.pt/
28/08/2014 - 09:20 AM > http://www.nytimes.com
28/08/2014 - 09:10 AM > http://www.publico.pt
```

Listing 2.5: Browser history example alternative format

If these two formats were to be synchronised between each other, so that one could modify one of them and then automatically propagate the changes made to the other, one could use the Boomerang bidirectional transformation shown in Listing 2.6.

```
1  let DAY : regexp = "Sunday" | "Monday" | "Tuesday" | "Wednesday" | "Thursday" | "Friday" | "
       Saturday"
2  let DATE : regexp = [0-9]{2} . "/" . [0-9]{2} . "/" . [0-9]{4}
3  let TIME : regexp = [0-9]{2} . ":" . [0-9]{2} . " " . ("AM" | "PM")
4  let URL : regexp = [A-Za-z0-9/:.]+
5
6  let line : lens =
7      del DAY . del ", " .
8      DATE . del ", " . ins " - " . TIME . del ", " . ins " > " . URL
9
10 let history : lens = "" | line . (newline . line)*
```

Listing 2.6: Boomerang transformation

Lenses are expressed in Boomerang by mixing regular expressions with specific commands recognised by Boomerang (`ins`, `del`, `.`, etc). Besides this, as mentioned before, with Boomerang it is only necessary to specify the *get* transformation. In this case, the *get* is a function receiving text following the format in Listing 2.4 and returning text in the format of Listing 2.5.

Taking this into account, firstly, the regular expressions were declared in order to recognise week days (line 1), dates (line 2), time (line 3) and URLs (line 4). Then, between lines 6 and 8, the lens for a single history occurrence was specified following the next steps:

1. Delete week day, which makes sense because it does not appear in the target format;
2. Delete `", "`, because the target format does not start with that;
3. Maintain the date, because the target format starts with that;
4. Replace `", "` for `" - "`;
5. Maintain the time, because it follows the date in the target format;
6. Replace `,` for `>` ;

7. Maintain the url;

Finally, in line 10, the complete lens for a sequence of history occurrences is defined, and given the name `history`. Having specified the lenses, it is possible to test if they have the desired behaviour. To check if the application of the *get* to the input in Listing 2.4 returns the input in Listing 2.5, one could execute `test (history.get h1) = h2`, after assigning string `h1` and `h2` to Listings 2.4 and 2.5 values, respectively. The result returned by Boomerang would be `true`.

We could also opt to test the *put* component by executing the code in Listing 2.7. In this case, we are updating string `h1` (Listing 2.4) based on string `h2`. As we can see, one of the updates made was stating that, in the first history occurrence, instead of visiting `http://www.seas.upenn.edu`, `http://www.google.com` was visited (line 3). The result returned by Boomerang is shown in Listing 2.8.

```
1  let h2 : string =
2  <<
3  31/08/2014 - 09:41 PM > http://www.google.com
4  30/08/2014 - 10:00 PM > http://www.springer.com
5  29/08/2014 - 10:17 AM > http://mei.di.uminho.pt
6  28/08/2014 - 09:20 PM > http://www.nytimes.com
7  27/08/2014 - 09:10 AM > http://www.publico.pt
8  >>
9
10 test (history.put h2 into h1) = ?
```

Listing 2.7: Execution of the *put* component

As we can see, each line was updated, by overriding it with the content of the `h2` string, and maintaining the content of `h1` (Listing 2.4), which is not mapped in `h2` (the days). This eventually lead to the last line (Listing 2.8 line 6) being "conceptually wrong" because the date was updated from the 28*th* of August to the 27*th* and, nonetheless, the week day was kept as `Thursday`. The specification does not take this detail into account.

```
1 Test result:
2 "Sunday, 31/08/2014, 09:41 PM, http://www.google.com
3 Saturday, 30/08/2014, 10:00 PM, http://www.springer.com
4 Friday, 29/08/2014, 10:17 AM, http://mei.di.uminho.pt
5 Thursday, 28/08/2014, 09:20 PM, http://www.nytimes.com
6 Thursday, 27/08/2014, 09:10 AM, http://www.publico.pt"
```

Listing 2.8: Result of the execution of the *put* component

To sum up, it is important to mention that the specification provided for this transformation was a simplified one. There are some details that this specification does not consider that would make it more complex. For instance, it is not possible to add a new history line in the updated target and expect that line to be included in the source artefact after applying the *put*. For that, a third component of lenses, called *create*, would have to be specified also.

2.4 LANGUAGES

Generally speaking, languages exist so that one is able of communicating. In the context of Computer Science, it is possible to regard several kinds of languages. Programming languages, for instance, are designed to communicate instructions to computers, by specifying programs. Depending on the scope of a language, it can be labelled as a DSL or a GPL. As the name suggests, DSLs are languages specifically tailored to address a certain domain. For instance, ATL is a DSL specifically designed to specify unidirectional transformations. UML, on the other hand, is normally viewed as a GPL to model any domain whatsoever. The separation between DSLs and GPLs is not as clear, though. For instance, UML is normally considered a GPML, nevertheless, it does not seem ridiculous to regard UML as a DSL oriented to the specification of mainly object-oriented software systems. Targeting a language as a domain-specific one or a general-purpose one depends on the scope of the domain one considers, and that is a subjective matter.

The following subsections address this topic, presenting GPLs and DSLs and their contrast. To do so, a particular case study has been considered: a bibtex database. The goal is to present a simplified model of a database to hold bibtex entries using GPLs and DSLs, so that the two perspectives can be compared. The languages used were Structured Query Language (SQL) (DSL), and Alloy and UML (GPLs). Bearing this in mind, it is valid to say that the case study focus on modeling languages in particular. Nevertheless, the conclusions drawn are naturally extended to any kind of language.

A modeling language is a language that enables the creation of models. Models abstract a problem or a system, filtering unnecessary details. In terms of the "anatomy", a model normally encompasses two concepts: building blocks; and constraints. The building blocks are basically the entities that make sense in the context of the system/problem being modelled. For example, when modeling a light bulb, a possible building block would be the tungsten filament. Constraints are the rules that model instances must respect to conform to the model. Once again considering the light bulb example, a natural constraint of the model would state that a light bulb has one and just one tungsten filament.

The are two reasons for addressing this topic at this point. Firstly, as mentioned in Section 1, a unidirectional and a bidirectional transformations were to be specified in this thesis. For specifying both of them, two alternatives were considered: in one of those, a DSL is used; and in the other, a GPL is used. Lastly, throughout this thesis, it was also necessary to choose between using a GPL or implementing a new DSL to specify the BDOs views, as explained in Section 1.

2.4.1 *Domain-specific Languages*

> "A Domain-Specific Language is simply a language that is optimized for a given class of problems, called a domain. It is based on abstractions that are closely aligned with the domain for which the language is built. Specialised languages also come with a syntax suitable for expressing these abstractions concisely. (...) Assuming the semantics of these

abstractions is well defined, this makes a good starting point for expressing programs for a specialised domain effectively." (Voelter, 2013, Chapter 2.2)

The main idea is that a DSL is a language specifically designed to handle problems on a certain domain. Bearing in mind the case study specified previously, the goal is to model a database do hold bibtex references. SQL was the DSL chosen in this case, which clearly bounds the model to the relational paradigm. SQL is a DSL specifically designed to specify database schemas and queries over them. The SQL model created for the case study is presented in Listing 2.9.

The specification of a model using SQL encompasses the creation/alteration of tables with the definition of constraints over them and the definition of queries to obtain data from those tables. By looking at the SQL model, one can notice that there are two types of references modelled - articles and books - because of the tables `Article` and `Book` (lines 9-24), which primary keys point to the table `Reference` (lines 14 and 23). The `Reference` table basically contains the attributes which are common between `Books` and `Articles`. Furthermore, it is also possible to notice that `References` and `Authors` are related between themselves by a M-M relation, because of table `ReferenceAuthor` (lines 32-38), which has two foreign keys pointing to the `Reference` and the `Author` tables. Every `Reference` (`Article` or `Book`) must always have a title and a year, which is clear because of the `not null` constraints.

The main conclusion to draw from this SQL model is the fact that the syntax used to define the model is completely oriented to the database domain. People acquainted with the database domain can easily understand a model specified with SQL, because they are aware of the concept of primary key, foreign key, inner join, etc. For instance, besides creating tables, the model also creates a view called `AuthorsBooks` (lines 40-46) which is an easy way to obtain the title of every book written by a specific author, identified by the `author_id` attribute. This would be an easy guess for a database expert because it is common knowledge that a (non-materialised) view is basically an alias for a query. Then, understanding the query is a matter of analysing the SQL code, which would quickly lead to the right assumption because of the inner join operator between content provided by the `Author` and `Reference` tables.

```
create table Reference {
    ref_id varchar(1024) not null,
    title varchar(255) not null,
    year int not null,
    primary key (ref_id),
    constraint chk_year check (year > 0)
}

create table Article {
    article_id varchar(1024) not null,
    journal varchar(255) not null,
    pages varchar(255),
    primary key (article_id),
    foreign key (article_id) references Reference(ref_id)
}

create table Book {
    book_id varchar(1024) not null,
    publisher varchar(255) not null,
    series varchar(255),
    edition int,
    primary key (book_id),
    foreign key (book_id) references Reference(ref_id)
}

create table Author {
    author_id int not null,
    name varchar(255) not null,
    primary key (author_id)
}

create table ReferenceAuthor {
    ref_id varchar(1024) not null,
    author_id int not null,
    primary key (ref_id, author_id),
    foreign key (ref_id) references Reference(ref_id),
    foreign key (author_id) references Author(author_id)
}

create view AuthorsBooks as
select authors.author_id, books.title
from ReferenceAuthor as authors
inner join (select ref_id, title
            from References as r
            where r.ref_id in (select book_id from Book)) as books
on authors.ref_id = books.ref_id
```

Listing 2.9: Bibtex model specified in SQL

2.4.2 *General-purpose Languages*

A GPL is one which is broadly applicable across application domains, lacking specialised features for a particular domain, in contrast with DSLs. When the language is suited for modeling and is at the same time a GPL, GPML is the acronym normally used instead.

The following sections present two specific GPMLs: UML and Alloy. As mentioned before, in order to present these two languages, a model of a database for bibtex entries was specified using both of them. The goal is to reproduce the model specified before with SQL, now using Alloy and UML.

UML & OCL

UML Object Management Group (b) is a GPML for the development of software systems. Despite this definition, UML is widely considered, not a single language, but a suit of languages, most of them with graphical syntax. It comprises a set of different diagrams for describing a system from different perspectives: 7 different diagrams can be used for describing the static (structural) aspects of a system, while other 7 can be used for describing the dynamic aspects Brambilla et al. (2012). One of the most used UML diagrams is the class diagram, which is the one chosen to model the case study. UML for itself sometimes is not powerful enough to enable the specification of certain constraints of a model. Object Constraint Language (OCL) Object Management Group (a) is a language adopted by OMG to fill this gap.

Taking this into account, Figure 9 is the class diagram modeling the case study described before and Listing 9 presents the additional OCL constraints over the class diagram.

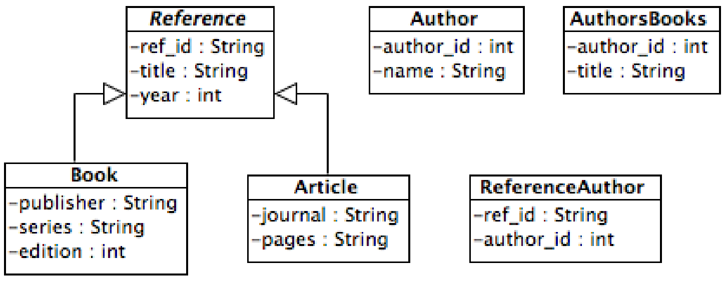


Figure 9.: Bibtex model Class diagram

The first difference in comparison to the SQL model presented before in Section 2.4.1 is the fact that it is no longer clear that the entities represented in this UML model are actually tables. In SQL, by using the syntax `create table`, one could easily understand that `Reference`, for instance, was a database table. With this UML class diagram, since it is a GPL model, the entities modelled are classes, which can be regarded as a synonym for an abstract concept, which lacks objectivity. Nevertheless, now it is clearer that `Book` and `Article` are types of references, because of the extension arrows, which in the SQL model were modelled with foreign keys. Besides this, what used to be specified in a single model is now divided in two: the class diagram; and the OCL constraints (in Listing 2.10). Every OCL constraint is defined with a specific class as context, stating to which objects the constraint is to be verified against.

```
 1 --References must have ref_id, title as well as year defined as attributes
 2 context Reference
 3 inv inv1 : (not self.ref_id.oclIsUndefined()) and (not self.title.oclIsUndefined())
 4
 5 --The ref_id attribute must be primary key of References
 6 context Reference
 7 inv inv2 : Reference.allInstances()->one(ref | ref.ref_id = self.ref_id)
 8
 9 --Articles must have the attribute journal defined
10 context Article
11 inv inv3 : (not self.journal.oclIsUndefined())
12
13 --Books must have the attribute publisher defined
14 context Book
15 inv inv4 : (not self.publisher.oclIsUndefined())
16
17 --Every author must have an id as well as a name
18 context Author
19 inv inv5 : not self.name.oclIsUndefined()
20
21 --The author_id attribute must be primary key of Author
22 context Author
23 inv inv6 : Author.allInstances()->one(a | a.author_id = self.author_id)
24
25 --The (ref_id,author_id) pair must be primary key for every ReferenceAuthor
26 context ReferenceAuthor
27 inv inv7 : ReferenceAuthor.allInstances()->one(ref |
28     ref.ref_id = self.ref_id and ref.author_id = self.author_id)
29
30 --The ref_id attribute must be foreign key for every ReferenceAuthor
31 context ReferenceAuthor
32 inv inv8 : Reference.allInstances()->exists(ref | ref.ref_id = self.ref_id)
33
34 --The author_id attribute must be foreign key for every ReferenceAuthor
35 context ReferenceAuthor
36 inv inv9 : Author.allInstances()->exists(a | a.author_id = self.author_id)
```

Listing 2.10: OCL constraints for the UML Class diagram in Figure 9

To sum up, most of the concepts and constraints that were clearly defined with SQL are now hard to understand and to define, using UML. For instance, in order to define `ref_id` as the primary key of the `Reference` table, invariant two was written (lines 6-7). Comparing invariant two and the analogous integrity constraint in SQL, it is clear that stating `primary key(ref_id)` is certainly a lot more efficient in terms of readability and easiness. In addition, another clear difference in terms of expressivity of UML in comparison to SQL is the fact of having to specify the `AuthorsBooks` entity in order to be able of reproducing the view with the same name specified in the SQL model. This view was specified using a `select` operator in the SQL model. In this case, the associated constraint to define the view was not even specified because of the obvious difficulties in order to do so using OCL.

Alloy

Alloy Jackson (2012) is another GPML, somewhat different from other mainstream languages, because of its embodied model checking feature. Not only does Alloy let one define a model, but it also generates model instances, to see if the specified model is consistent. This process is done with the help of the Alloy Analyser. "The Alloy Analyzer translates constraints to be solved from Alloy into boolean constraints, which are fed to an off-the-shelf SAT solver" Jackson (2012). Listing 2.11 represents the same case study modelled previously with UML and OCL, now in the Alloy notation.

```
abstract sig Reference {
    //Every reference must have a ref_id attribute, which identifies it
    ref_id : one CharSequence,
    //Every reference must have one title
    title : one CharSequence,
    //Every reference must specify the year of creation/publication
    year: one Integer}
sig Book extends Reference {
    publisher: one CharSequence, series : lone CharSequence,
    edition: lone Integer}
sig Article extends Reference {
    journal: one CharSequence, pages : lone CharSequence}
sig Author {
    author_id : one Integer, name : one CharSequence}
sig ReferenceAuthor {
    ref_id : one CharSequence, author_id : one Integer}
sig AuthorsBooks {
    author_id : one Integer, title : one CharSequence}
sig CharSequence {} sig Integer {}

fact bibtexRules {
    //Cons1 - The ref_id attribute is primary key of references
    all cs : CharSequence | lone (Reference<:ref_id).cs
    //Cons2 - The author_id attribute is primary key of authors
    all i : Integer l lone (Author<:author_id).i
    //Cons3 - The ref_id and author_id must be primary key of ReferenceAuthor
    all cs : CharSequence, i : Integer | lone ((ReferenceAuthor<:ref_id).cs &
                                (ReferenceAuthor<:author_id).i)
    //Cons4 - The ref_id attribute must be a foreign key pointing to References
    ReferenceAuthor.ref_id in Reference.ref_id
    //Cons5 - The author_id attribute must be foreign key pointing to Author
    ReferenceAuthor.author_id in Author.author_id
}
```

Listing 2.11: Alloy model Bibtex entities

One way of understanding Alloy models is to assume that, fundamentally, a model contains entities and relations between them. Entities are declared with the `sig` keyword, whereas relations are declared "inside" entities, like `ref_id` (line 3) which is a binary relation of type: `Reference` → `CharSequence`. In this case, since the objective is to model a bibtex database, reproducing the SQL model, Alloy entities represent database tables, implicitly, which is not as elucidative as the SQL straight-forward notation. Alloy models also have constraints. In order to specify primary keys, for example, the constraints in lines 34-40 were specified. In spite of being a more compact way of specifying primary keys than the format used with OCL, it is still not as efficient as the SQL notation. In addition, the constraints declaring the mandatory attributes for every reference were declared directly "in-line" with the relations, such as the statement in line 7 which forces every `Reference` to have a specific year defined. Once again, it has been opted not to define the `AuthorsBooks` view.

Nevertheless, there are actually some constraints that one would define easily with Alloy. Supposing, for instance, that it was decided to model cross-references. A cross-references is a way of a reference to point to another:

```
sig Reference {
   ...
   crossref : lone Reference
}
```

A typical constraint that one would wish to define would prevent a particular reference from cross-referencing itself. This constraint would be defined this way in Alloy:

```
all r : Reference | r not in r.^crossref
```

This is a constraint which does not simply evaluate if a reference points directly to itself - it evaluates if all the references pointed through transition by the original one do not contain the original one. It is a recursive constraint. Such constraint cannot be defined easily with SQL.

As mentioned before, Alloy allows model checking, generating model instances, provided that the model given as input is consistent. In this case, by running the model in Listing 2.11, one of the instances provided by the Alloy Analyser is the following:

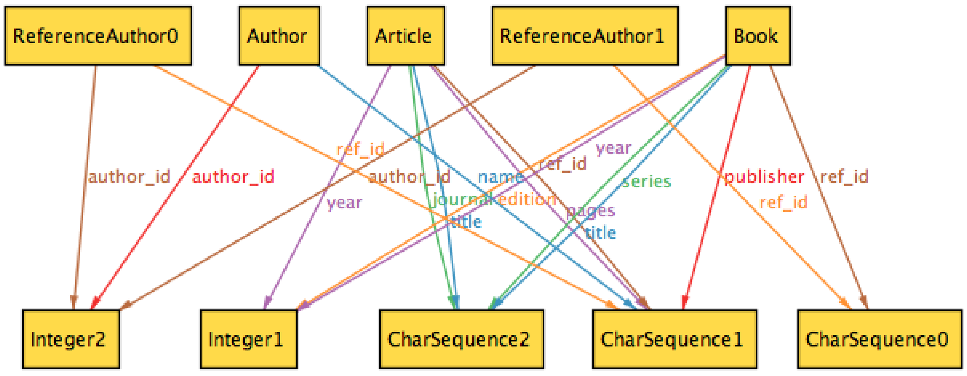


Figure 10.: Model instance provided by Alloy Analyzer

This is Alloy's graphical notation. In this case, the model instance includes one `Book` and one `Article`, both from the same `Author`. This is clear because, in this case, `CharSequence0` corresponds to the book's `ref_id` and `CharSequence1` is the article's `ref_id`, and, at the same time, these two `CharSequence`s are pointed by two different `ReferenceAuthor`, which also point to the same `Author`.

Strings were modelled in Alloy by defining the `CharSequence` entity, which makes the model instances provided not as realistic as one would wish (since no real strings are actually used, like "John Murray"). It was modelled this way because, in spite being possible to use Strings in Alloy, one can only use them in a limited way.

Part II

CORE OF THE DISSERTATION

3 THE PROBLEM

As mentioned earlier, this thesis was developed in the context of an EISC, which, at the moment, is in the middle of a modernisation process. This company sells a software system for insurers to manage their insurance package offers and clients. In spite of being one of the top products in the market for this purpose, the system needs an upgrade, since it was developed several years ago, enough for it to be considered a legacy system. This modernisation process aims precisely to upgrade the system, by migrating it to current technologies.

The modernisation process is being driven by MDE guidelines. The idea is to map the legacy code into models, that will afterwards be used as input to generate the new code. In this case, Java was the chosen target platform for the modernisation. The final goal is to have, not only a totally migrated and equally functional system comparing to the legacy one, but also to adopt an entirely new development process guided by MDE. At that point, the development will be based on specifying models and then applying transformations to generate the underlying code.

For this dashing goal to be achieved, several steps need to be addressed progressively. First of all, for the models of the system to exist, previously there needs to exist at least one modeling language to specify those models. In the case of this EISC, several DSMLs were implemented with the purpose of modeling different aspects of the system. One of those languages was designed to model business objects in particular, in this case related to the insurance domain, such as policies, accidents, etc. This language was called the BDO language.

The problem addressed during this thesis is divided in two subproblems, both of them based on the BDO language: the Global Perception problem; and the Suitability for Insurance Experts problem. Since these problems are related but, even so, separate, two subsections are presented below in order to address them more accurately. In addition, firstly, another subsection is dedicated to introduce further the BDO language.

3.1 THE BUSINESS DOMAIN OBJECTS LANGUAGE

The EISC defines the BDO language as a DSL created for specifying business entities. In this case, the business entities are strictly related to the insurance domain, consequently, policies, receipts... are legitimate examples of such entities. Nevertheless, the most correct assumption for this language would be to regard it as a GPL which was created specifically to model insurance entities. There are no specific elements in the BDO language that clearly bind it to a particular domain such as the insurance one, thus, the BDO language is general enough for it to be applied to any domain whatsoever and, in this case, the domain is the insurance one.

Files with the .bdo extension are files using the BDO language. The final goal for these files is to use them as input to generate their respective pieces of Java code, being classes, interfaces, etc. This is the reason why the BDO language leans towards the object-oriented paradigm somehow, as explained below.

The BDO language follows the XML syntax. The respective metamodel which the BDO language derives from is given in Appendix .1. In order to describe more properly what one is able of specifying by using the BDO language, Listing 3.1 illustrates an example of a .bdo file. By observing the Listing, one is able of basically identifying the skeleton of a Java class:

- The respective package is given in line 6 (`com.i2s.tut.pas.policy`);
- The name of the class is given in the root xml element, by the `name` attribute (`Policy`), in line 5;
- A super class is also identified in the `classDef` element by the attribute `extends` (`GISCfgDataObj`), in line 12;
- Constructors are defined inside the `classDef` element;
- Attributes are defined by each XML element with the `element` tag (for instance, the `produtoRef` attribute of type `ProdutoRef` in line 20);

In addition to this, the BDO language also allows the inclusion of `imports`, the specification of `interfaces` for the entity to implement and, finally, the definition of `extramethods`, which are short pieces of code to be included/injected in the final Java class to be generated. This is how a BDO normally looks like. As mentioned before, now it is clear that BDOs are really oriented towards the object-oriented paradigm, because they basically allow the definition of classes.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<businessobject
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.i2s.pt/schemas/i2s/devtools/generators/bdo/
        i2sbusinessobject.xsd"
    name="Policy">
    <package>com.i2s.tut.pas.policy</package>
    <import>com.i2s.utils.ProdutoReference</import>
    <interfaceDef>
    </interfaceDef>
    <interfaceInternoDef>
    </interfaceInternoDef>
    <classDef extends="GISCfgDataObj">
        <constructor>
            <parm tipo="ProdutoReference" name="prodRef" element="produtoRef"/>
        </constructor>
        <inconsistenteignorables>
            <ignore>POS_REFSUPERAPOLICE</ignore>
        </inconsistenteignorables>
    </classDef>
    <element name="produtoRef" type="ProdutoReference" write="false" />
    <element name="codMoradaPagamento" type="long" read="abstract"/>
    <element name="dataEmissao" type="Data" />
    <element name="duracaoPagamentoAutomatico" type="boolean" />
    <element name="fracionamento" type="Fracionamento" />
    <element name="mesDePagamento" type="int" />
    <element name="mesVencimento" type="int" />
    <element name="currency" type="I2SCurrency" />
    <element name="numApolice" type="int" />
    <element name="refSuperApolice" type="RefApolice"/>
    <element name="regraCalculo" type="int" />
    <element name="titular" type="Titular" />
    <element name="utilizador" type="String" />

    <extramethod>
        <extra>char APOLICE_EM_VIGOR = 'L';</extra>
    </extramethod>
    <extramethod>
        <extra>boolean isApoliceEmitida();</extra>
    </extramethod>
    <extramethod>
        <extra>I2SCurrencyAmount getSaldoPremioCobrado(Data dataEfeito);</extra>
    </extramethod>
</businessobject>
```

Listing 3.1: `Policy.bdo` file

3.2 THE GLOBAL PERCEPTION PROBLEM

As mentioned in the previous section, the BDO language is used to specify insurance entities/classes. At the same time, the BDO language makes available a quite useful feature designated as "cross references", which allows to reference external BDO files inside a BDO file. For instance, by looking at the Policy BDO in Listing 3.1, the first attribute defined is named `produtoRef` and its type is `ProdutoReference` (line 20). In this case, the `type` of the attribute is actually the `ProdutoReference` entity defined in another (external) BDO file, located in the package `com.i2s.utils.ProdutoReference` (because of the `import` element declared in line 7). This is a cross reference.

The problem is that, as the system evolves with the specification of more and more BDO files, as the need for defining new entities surges, a *global perception* of the system is completely miss regarded by each BDO, which is natural since each only encloses a single entity. By global perception one means amongst other concepts:

- The size of the system, in terms of the number of BDOs specified, for example;
- The density of the network of BDOs, due to cross references, which sheds light on valuable information such as:
  - The most referenced BDO in the system;
  - The BDO with the highest number of cross references;
  - Isolated BDOs;
  - etc;

This is useful information for the EISC, which the BDOs completely lack. The best way to perceive this global perception would be to represent the entire network panorama of the BDOs in a clear graphical way, so that with a single visual look one would have a slight grasp of the global perception.

3.3 THE SUITABILITY FOR INSURANCE EXPERTS PROBLEM

Supposedly being a DSL created specifically to model insurance objects, it would be natural to assume that the BDO language should be an easy to use one for insurance experts. Nonetheless, there are some reasons why this is not the case:

- Just the fact that the BDO language is an XML language accounts negatively for the insurance experts. Unlike software engineers who are accustomed with the XML syntax, common people usually do not appreciate XML, either because they associate it with programming and computers or because they actually do not know XML and simply do not like its visual aspect;

- As mentioned before, in spite of being a language which allows the specification of entities common to the insurance domain, the BDO language is mainly a language good for specifying object-oriented classes, which involves some concepts which insurance experts are not supposed to be acquainted with, such as interfaces, inheritance, methods, etc.

The second bullet is the crucial reason for the dislike, because even if the insurance experts get used to the XML notation, there will always exist those concepts in the language which they will not be able of specifying for themselves without recurring to the assistance of the software company's consultants. On the other hand, the software company's consultants will also have problems creating BDOs as, in their case, they lack the insurance expertise. This is a common problem in Software Engineering, which Brambilla et al. (2012) designates as "the gap between business requirements and IT implementation".

4 CONTRIBUTION

Now that the Problem chapter was addressed, the Contribution chapter follows as the one where the solutions for those problems stated are presented. Since there are basically two separate subproblems, this chapter will also be divided in those two parts: firstly the Global Perception problem is addressed; and then the Suitability for Insurance Experts is addressed.

For each one of these problems a solution was proposed, for which two implementations were conceived. The final goal is to compare the two implementations given for each problem so that the best one is chosen reasonably. Bearing this in mind, in the following sections, each problem is reintroduced briefly, the respective solution is explained, the respective two implementations are described and a final comparison between the two implementations is given.

4.1 THE GLOBAL PERCEPTION PROBLEM

The global perception problem alludes to the fact that BDO files only encompass a single entity perspective and, since the global system is composed of several BDOs, a more global vision of the system is something that is useful and, at the same time, BDOs for themselves do not express this point in any manner. BDOs form a network between themselves which the global perception view would elucidate better.

4.1.1 *Solution*

The solution proposed in order to solve the Global Perception problem was to merge all the current existing BDOs in a single model that could be graphically visualised. Thus, in this case, the solution is basically a transformation that receives several BDOs as input and generates a single model, which unifies all the BDOs, and for which there exists a good graphical renderer to display the desired network.

Taking this into consideration, the first point to be addressed after proposing this solution was choosing the target platform for the transformation to be specified. For this point specifically, the goal was to find a language to specify models with a graphical visualiser already implemented, so that implementing one from scratch would not be necessary. After some research, the choice was made: the OWL language. OWL is also an XML language, endorsed by the W3C, used for specifying ontologies. Its respective schema is in Appendix .2. An ontology can be regarded as a domain model, because it describes the entities that exist in some domain and how they relate between each other. The difference of ontologies in comparison to domain models is the fact that, in the case of ontologies, instances of the defined entities are also enumerated/included in the ontology definition. Considering, for instance, an ontology describing Humanity. In that case, "Man" and "Woman" would obviously be two entities and, at the same time, "Jessica Alba" could be declared as an instance of the "Woman" entity, for example. That's something that models normally do not consider. Nonetheless, in this case, the interest does not lie specifically in the OWL language itself. It lies in Protégé[1], which is a free editor/renderer of OWL ontologies.

In terms of visual features, Protégé offers very good graphical visualisers, because not only are they appealing, but also because they allow an incremental visualisation of the ontologies. This last feature is particularly good when the ontologies are very extensive, because, in these cases, observing the ontology all at once is normally not useful at all. This is precisely the case with the transformation to be defined. In this case, the transformation is supposed to merge several BDOs in one single ontology. In order to test the transformation, about three hundred BDOs were included - this results in a considerably large ontology to display. These were the reasons for opting for OWL. Figure 11 illustrates a screenshot of Protégé's graphical visualiser (by clicking in the "+" signs in each box, one is able of expanding the model).

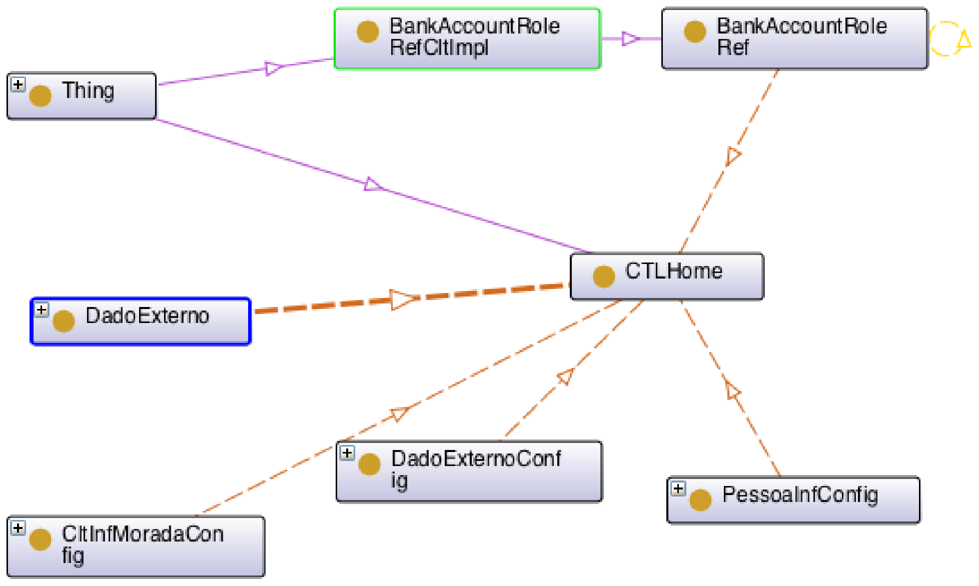


Figure 11.: Screenshot of Protégé's graphical visualiser

1 http://protege.stanford.edu/

Now that both the source and target of the transformation are clear, it is possible to classify the transformation according to the taxonomy defined in section 2.3:

- Output

  Since both the input and output of the transformation are XML files, one could actually opt to implement a transformation from **Text to Text**, using XSLT. In that case, this would not be a Model Transformation. Nevertheless, opting for that road is normally not a good choice, because **Text to Text** transformations are based on concrete syntax, which is something that is supposed to be flexible over time. In MDE there are two concepts: *abstract syntax*; and *concrete syntax*. The abstract syntax is basically the model concept itself, while the concrete syntax defines how instances conforming to the model (abstract syntax) should be represented (text/graphically). So, for one abstract syntax there can exist more than one concrete syntax. Bearing this in mind, normally it is preferable to have **Model to Model** transformations, because those are based on the abstract syntax both in the source and in the target. Consequently, the transformation to implement in this case should also be a Model to Model one;

- Cardinality

  This is a model merging transformation, since it is from N BDOs into 1 OWL model;

- Source/Target Metamodels

  This is an exogenous/translation transformation because the input conforms to the BDO language and the output conforms to the OWL language;

- Source/Target Abstraction Levels

  This transformation is a vertical one, because its target is an abstraction of the source. In this case, the interest of the target resides only in parts of the BDOs (source), which are the following:

  - The package of the BDO, identified by the value of the `package` attribute inside the XML elements with the `businessobject` tag;
  - The name of the BDO, identified by the value of the `name` attribute inside the XML elements with the `businessobject` tag;
  - The name of the possible class which the BDO extends, identified by the value of the `extends` attribute inside the XML elements with the `classDef` tag;
  - All the attributes of the BDO, identified by the XML elements with the `element` tag. Inside these `element` XML elements, the interest is in:
    - The `name` attribute;
    - The `type` attribute;

- Directionality

  This is an unidirectional transformation because the interest is in the direction that takes N BDOs into 1 OWL ontology. The other way round is not necessary, because the OWL files are supposed to be used just for visualisation, not for update.

Having completely characterised the transformation, the final step was the concrete implementation. In this case, the goal was to compare two different approaches for the specification of the transformation. One of the approaches uses a DSL to specify the transformation and the other uses a GPL. The DSL chosen was ATL, which is a language designed specifically to specify unidirectional model transformations. ATL was chosen because it is a MDE de-facto standard for unidirectional transformations and because it is strongly embedded in the Eclipse framework, which is the preferential IDE used by the EISC in question. Java was the GPL chosen. Java is a Touring-complete programming language, which accounts for its general-purpose categorisation.

In order to go through with any of the two implementations, it was necessary to have a way of "navigating" the BDO files in memory, to be able of defining the transformation. One way of doing this would be to use a general XML parser, but this option would not be good because of its generality. An XML general parser offers an API where concepts like Node, Attribute, Child... are defined - XML concepts. In this case, the input of the transformation are models specified using the BDO language, which defines its very own concepts, like `classDef`, `element`, etc. A good API for navigating the BDOs would be one where these concepts were defined, not general concepts, where one needs to check afterwards if they correspond to a specific one.

To solve this problem, it was decided to shift from XML into XMI, or, in other words, to change from XML Schema metamodels into Ecore metamodels. This is designated as a shift between *technical spaces*. Mens and Van Gorp (2006) defines *technical space* as "a model management framework containing concepts, tools, mechanisms, techniques, languages and formalisms associated to a particular technology.". The technical spaces involved in this case are the XML and the Ecore technical spaces. Taking this into consideration, it is important to clarify that both implementations are based in Ecore, despite being implemented with different specification languages. Migrating from XML into Ecore is easily done because the Eclipse framework offers converters between those two formats and, by the time the Ecore is obtained, obtaining the API to navigate over the BDOs is done automatically, by code generation (once again with Eclipse).

To sum up, Figure 12 illustrates a schema of the complete mechanics of the complete transformation, from the BDO XML files in the input into the OWL xml file in the output.

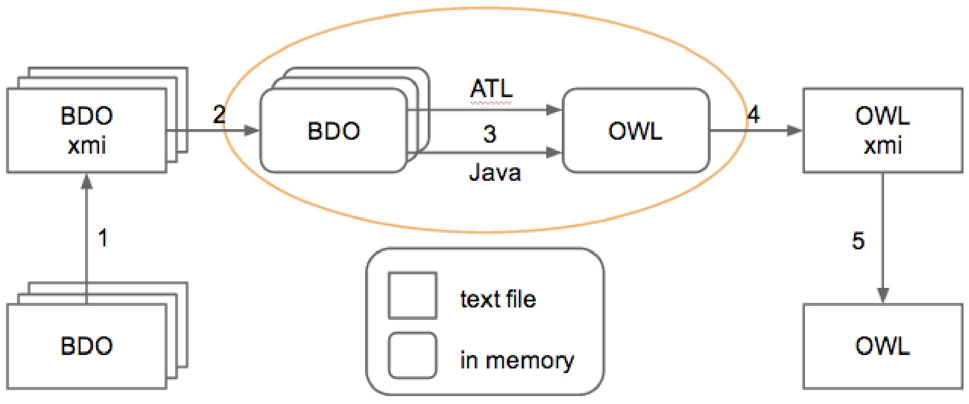


Figure 12.: Schema of the general transformation solution for the Global Perception problem

The specific transformation in which we are interested in is illustrated by the arrow number 3 - all the remaining transformations/arrows are complementary for the complete transformation:

- Arrow 1 - Text to Text transformation to transform BDOs in the XML notation into the XMI expected notation by Ecore's default parser;
- Arrow 2 - Parsing;
- Arrow 4 - Model to text transformation to persist the model in text (xmi notation);
- Arrow 5 - Text to Text transformation to convert from XMI into the Protégé's expected OWL notation.

As expected, Figure 12 confirms that the transformation to specify is a model to model many-to-one transformation and that both implementations "begin" with **Ecore** BDO instances, despite being supported by different languages (ATL/Java).

The following sections present the two implementations conceived for the solution explained above. Nevertheless, before the implementations, a brief introduction of OWL is given.

4.1.2 *OWL*

OWL stands for *Web Ontology Language* and, as the name suggests, it is a language developed for the specification of ontologies. The best way to explain OWL is to understand the following sentence taken from World Wide Web Consortium (2004):

> A *class* defines a group of *individuals* that belong together because they share some *properties*.

The words emphasised in the quote represent the main concepts an ontology is formed by: classes; properties; and individuals. A class is a group of individuals, as the quote puts it. For instance, one could define the class "Living Being" to describe every living being that exists in the world. Classes are organised in an hierarchical way, so that one is able of specifying what subclasses a specific class derives into. Considering the example proposed, the "Living Being" class could derive into the subclasses "Human" and "Animal", to separate between humans and animals, precisely. Every class defined in an OWL ontology eventually extends the "Thing" class, which is a default one. This means that "Thing" is the class in the top of the hierarchy. In addition, there exists another default class in OWL, named "Nothing", which is the lowest class in the hierarchy since it extends every class invariantly.

Properties are relations between classes or from classes to primitive types. There are two types of properties. Object properties establish a relation between classes, whereas data properties establish a relation between classes and some primitive type supported by OWL, such as integers, strings, etc. Thus, if, for instance, one would like to define a property to state the nationality of a "Human", one of two ways could be used. One could define a data property named "nationality", which domain would be the "Human" class and which range would be strings, for one to write the nationality in words. Alternatively, one could define the "Country" class and create an object property named "is from" between the "Human" and "Country" classes.

Finally, individuals are the concrete instances that compose classes. For instance, "Jessica Alba" could be declared as an individual of the "Human" class. In order to state her nationality:

- If the first solution was the one used, one could simply assign the string "north-american" to her, in the "nationality" property;
- If the second solution was the one opted for, firstly it would be necessary to declare "USA" as an individual of the "Country" class and then relate "Jessica Alba" to "USA" by the "is from" object property.

Individuals are the main difference of ontologies in comparison to models, generally. Models normally only encompass classes and properties (relations) - ontologies add individuals. To finalise, Figure 13 represents Protégé's visual rendering of the 'Living Beings" ontology that has been used as example. In this case, the nationality was modelled with the object property "is from" (yellow arrow).

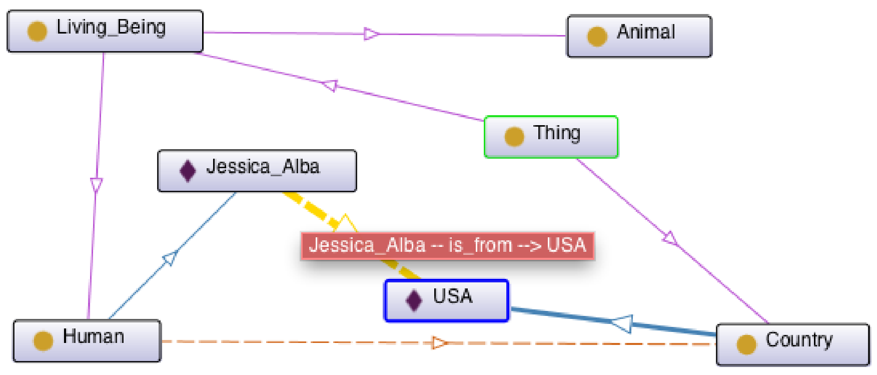


Figure 13.: “Living Beings” ontology graphical visualisation

4.1.3 *ATL Implementation*

This ATL implementation follows the hybrid approach, defined in section 2.3.1. In this case, the defined ATL rules are declarative, but those rules are aided by some ATL helpers, which are basically blocks of imperative code, like we will see later. This is why, in this case, the implementation is said to follow the hybrid approach. Before explaining the actual ATL implementation, an ATL introduction is given.

ATL Introduction

ATL is a DSL tailored for the specification of unidirectional model transformations. Nowadays, it is recognised as the standard of Eclipse for unidirectional transformations. This section aims to provide the reader a short introduction of ATL, focusing on the concepts of the language that were used to implement the transformation to solve the Global Perception problem.

ATL transformations can be executed using one of two modes: the normal mode; and the refinement mode. With the normal mode, the developer specifies the way the target model is generated from source model elements, by specifying ATL rules. Using this mode for specifying *refactorings*, where only short parts of the source are modified, can be very tiresome since the developer, not only has to define the rules for the modifications to occur, but also has to define rules to guarantee that the source elements that are supposed to be preserved are in fact copied to the target. The refining mode exists precisely for this kind of situations, preserving non modified source elements in the target, so that only the refactoring rules have to be defined. The execution mode we used to implement the transformation was the normal one.

A typical ATL transformation has the structure illustrated by Listing 4.1. A module name must always be declared (line 4) and after that the signature of the transformation (line 6). The first two lines of the transformation are two comments stating the location of the input and output Ecores, and they bind the strings `input` and `output`, in this case, to each location. These two strings are then used in the transformation signature. In this example, the transformation is supposed to receive two instances of the `Meta1.ecore` model, which are labelled `IN1` and `IN2`, and generate one instance of the `Meta2.ecore`.

```
-- @path input=/Metas/Ecores/Meta1.ecore
-- @path output=/Metas/Ecores/Meta2.ecore

module transformationMod;

create OUT: output from IN1: input, IN2: input;

--helpers
--body
```

Listing 4.1: Typical structure of an ATL Transformation

One annoying problem of ATL is the fact that, in the signature of a transformation, it is necessary to literally enumerate each input and output the transformation is supposed to operate with. If the transformation to specify is supposed to receive one thousand source instances all conforming to the same model, it would be necessary to write:

```
from IN1: model, IN2: model,...,IN1000: model
```

This can be an obstacle when one cannot infer, statically, the expected number of inputs the transformation is supposed to receive. In this case, the number must be calculated somehow, and then, “on the fly”, the transformation must be specified and compiled to be executed afterwards. Having a statement to specify that a transformation is supposed to receive *several* inputs of some type (something like `IN: model`*) would be much more flexible.

After declaring the signature of the transformation, the concrete rules to generate the target model(s) are defined. Rules are declarative blocks of code, which can be supplemented by helpers. Helpers are imperative blocks of code that can be defined to do some kind of useful calculation to be used in some rule. Helpers can be regarded as the object-oriented methods of ATL transformations and are defined according to the following scheme:

```
1 helper [context context_type]? def :
2 helper_name(parameters) : return_type =
3 exp;
```

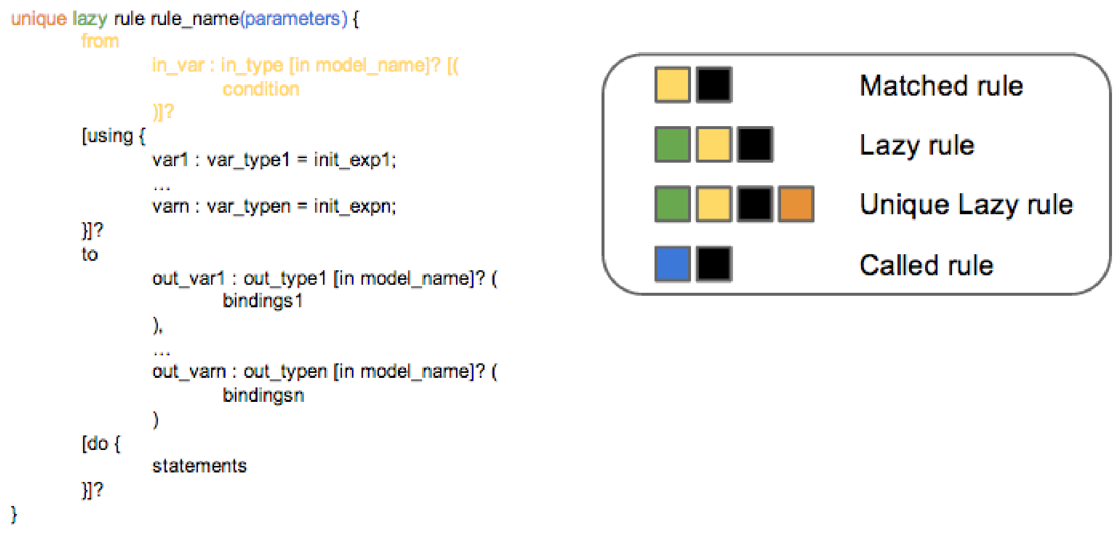
An helper is always identified by a name (line 2) and can have a context (line 1), which is the type of elements on which the helper can be invoked. If no context is given, the helper has a global context and is called differently. In addition, an helper always has a return type and can have or not some parameters. The body of the helper is an imperative block of code, supporting *If*s and *For*s as control statements as well as OCL expressions.

Last but not least, taking into consideration the rules, there are four kinds of rules:

- Matched Rules - These rules are called automatically by the execution engine. In this case, a source pattern is defined for the rule, stating which source elements are to be matched and target elements are specified, stating what elements in the target are to be generated and how to do so. For each source elements that match a matched rule source pattern, that rule is called and the respective target elements are generated;

- Lazy rules - These rules are similar to Matched Rules, but these ones have to be explicitly called to be executed. The execution engine will not called them automatically. Thus, basically, for these rules to be used, the developer must explicitly invoke them in the specification (code) of other rules;
- Unique Lazy Rules - Lazy rules, when called over a certain element, always instantiate a new target element. In the case of unique lazy rules, if the rule has been called before for some element, when called again for that element, instead of instantiating a new target element like lazy rules do, the target previously created by the rule when called is used;
- Called rules - These rules are similar to lazy rules, in the sense that both of them need to be called explicitly. The difference is in the fact that, with called rules, source patterns are not specified - only target elements are. Thus, called rules are useful to generate target elements that must always exist regardless of the source given. Besides this, called rules also allow the specification of parameters to pass to the rule upon invocation;

Bearing this in mind, Figure 14 presents a *merged* template of the structure of each type of rule defined above.



```
unique lazy rule rule_name(parameters) {
          from
                    in_var : in_type [in model_name]? [(
                              condition
                    )]?
          [using {
                    var1 : var_type1 = init_exp1;
                    ...
                    varn : var_typen = init_expn;
          }]?
          to
                    out_var1 : out_type1 [in model_name]? (
                              bindings1
                    ),
                    ...
                    out_varn : out_typen [in model_name]? (
                              bindingsn
                    )
          [do {
                    statements
          }]?
}
```

Figure 14.: Template with the structure of all ATL rules types

The *from* and *to* sections are used to declare the source patterns and the target outputs, respectively. The *using* section is where local variables are declared and the *do* section is a zone for some imperative code, typically to update global variables of the transformation.

ATL Transformation

This section explains the ATL transformation specified to solve the Global Perception problem. In order to do so, the explanation will be focused essentially on the transformation rules, and less on the helpers to supplement those rules. Futhermore, to make the explanation easier, it will be based on a particular example of the application of the transformation: Listing 4.2 represents the BDO given as input; and Figure 15 represents the OWL ontology returned (rendered by Protégé), after applying the transformation.

```
<businessobject name="HistoricoPensao">
    <package>com.i2s.gis.pensao.historicoPensao</package>
    <classDef extends="GISDataObjImplSer">
        <constructor>
            <parm tipo="GISHome" name="home" element="home"/>
            <parm tipo="Pensao" name="pensao" element="pensao"/>
        </constructor>
    </classDef>
    <element name="pensao" type="Pensao" read="true" write="true"/>
    <element name="numeroSequencia" type="String" read="true" write="true"/>
    <element name="tipoPensao" type="int" read="true" write="true"/>
    <element name="salarioAnual" type="I2SCurrencyAmount" read="true" write="true"/>
    <element name="numeroPensionista" type="int" read="true" write="true"/>
    <element name="utilizador" type="Utilizador" read="true" write="true"/>
    <element name="dataCriacao" type="Data" read="true" write="true"/>
</businessobject>
```

Listing 4.2: BDO given as input to the ATL transformation

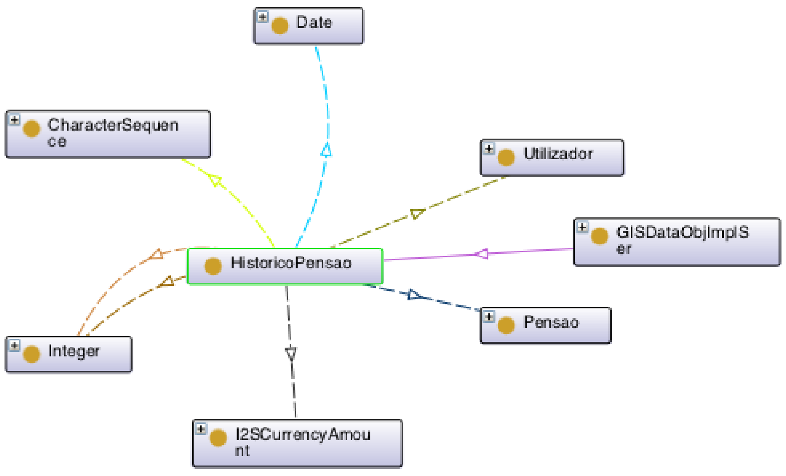


Figure 15.: OWL ontology returned by the ATL transformation

Beginning with the transformation header, as illustrated by Listing 4.3, the transformation receives several BDOs as input, which conform to the `bdo.ecore` model and generates one OWL instance, conforming to the `owl.ecore` model. In this specific example, there are a total of 328 BDOs declared as input of the transformation. Every time this number changes, a new transformation with the same body as this one, but a signature slightly different by adding more BDOs as input, must be generated, as explained in the previous section. The BDO and the OWL Ecores were both obtained by converting their XML Schemas into Ecore.

```
--@path bdo=/ThesisProject/Ecores/bdo.ecore
--@path owl=/ThesisProject/Ecores/owl.ecore

module bdo2owl;

create OUT: owl from IN1: bdo, IN2: bdo, IN3: bdo, IN4: bdo ...
```

Listing 4.3: Transformation header

Shifting focus into the actual body of the transformation, 4 transformation rules were specified along with 15 helpers. The first rule to be executed by ATL's execution engine is the rule named `rule1`, illustrated in part by Listing 4.4. `rule1` is a Called Rule with the `entrypoint` keyword attached to it and that is why it is the first rule to be executed. This rule was specified in order to guarantee that all the *primitive* and *default* types were generated in the target side, since those have to exist by default. The final idea of this transformation is to be able of seeing in Protégé the relations between BDOs, which exists by two ways: either a BDO has an attribute (`element` tag) which `type` points to another BDO; or a BDO `extends` another BDO. Nevertheless, some BDOs also have attributes pointing to primitive types, such as Integers, Booleans, etc (see, for example, line 11 in Listing 4.2). To also be able of displaying those attributes in Protégé, all the primitive types must be declared in the target, the same way BDOs are. The same happens with the default types, that basically are global BDOs recognised across the system, in spite of not being given as input in the transformation. For example, the `GISDataObjImplSer` BDO, which is extended by the `HistoricoPensao` BDO (line 3 Listing 4.2) is a *default* BDO.

```
entrypoint rule rule1() {
  to
    ...,
    string : owl!OWLClass (
      uriRef <- Sequence{stringURI},
      label <- Sequence{stringLabel}
    ),
    stringURI : owl!URIReference (uri <- stringUriRef),
    stringUriRef : owl!UniformResourceIdentifier (
      name <- thisModule.ontologyName() + '#' + 'CharacterSequence'
    ),
    stringLabel : owl!PlainLiteral (
      lexicalForm <- 'CharacterSequence'
    ),
  ...
```

Listing 4.4: Part of the Called Rule `rule1`

The part of the rule1 which is covered by Listing 4.4 is responsible for the string primitive type to be generated. The same procedure is done for every other primitive and default type to declare, in this rule. The string primitive type is mapped into an OWLClass, which uri attribute "points" to the expression[2]:

```
thisModule.ontologyName() + '#' + 'CharacterSequence'
```

and which label "points" to CharacterSequence. This actually means that, instead of calling String, it was decided that there would exist an OWLClass labelled CharacterSequence which represents the string primitive type that exists in the bdo.ecore. For example, by looking at Listing 4.2, in line 10, there is an attribute of type String. On the other hand, in Figure 15, that same attribute now points to CharacterSequence (light green arrow), which is possible, in the first place, because rule1 declared the CharacterSequence OWLClass.

By observing the actual code of the rule, one realises that the label and uriRef attributes of the OWLClass are not strings directly. Taking the uriRef attribute as example, it points to a URIReference, which uri attribute points to a UniformResourceIdentifier, which, in this case, points directly to the actual string (PlainLiteral) by the name attribute. This happens because the OWL's Ecore is very "verbose". Finally, in lines 5 and 6, the Sequence keyword is used. This happens because, in these cases, the attribute type in the respective Ecore model is expecting not one but several entities. Consequently, in that cases, these Sequence "wraps" must be declared.

The following rule which deserves some attention is the BusinessObject2Class Matched Rule. This is probably the most important rule because it is the one which maps BDOs into OWLClasses. Since this is a Matched Rule, it is automatically called by ATL's execution engine for each Business objectComplexType found, which is the type that wraps each BDO, in the bdo.ecore. The rule is illustrated in Listing 4.5. By observing the rule, it is possible to realise that, just like in rule1, the uriRef and label attributes are defined for the OWLClass, mapping the BDO in this case. The uriRef attribute ends up pointing to the string formed by the expression

```
thisModule.ontologyName() + '#' + bo.package.trim() + '.' + bo.name.trim()
```

which corresponds to the ontology prefix followed by the *fully qualified name* of the BDO class (package + name). The trim is used just to make sure that blank spaces are suppressed. The label attribute simply corresponds to the name of the BDO (bo.name.trim()). This name is the one which is displayed by Protégé's renderer.

The "primary key" of every OWLClass is the uriReference - two OWLClasses with the same URI are the same class. However, in this case, two more attributes are defined for the OWLClass: the superClass attribute; and the propertyForDomain attribute. The superClass holds, unsurprisingly, the OWLClass of the BDO which the matched BDO extends, in case it does. The URI of the OWLClass is calculated by calling the classURI helper (line 20). The propertyForDomain

2 thisModule.ontologyName() is a call to an helper with global context, which in this case, returns the default name of the ontology to be generated (every URI must be prefixed by the ontology name).

holds the several attributes the BDO can have, which are mapped as `OWLObjectPropertys` in the target.

```
 1 rule BusinessObject2Class {
 2   from
 3     bo : bdo!BusinessobjectComplexType
 4   to
 5     c : owl!OWLClass (
 6       uriRef <- Sequence{ur},
 7       label <- Sequence{className},
 8       superClass <- Sequence{sc},
 9       propertyForDomain <- bo.element->collect(p | thisModule.Property2ObjectProperty(p,bo))
10     ),
11     sc : owl!OWLClass (
12       uriRef <- Sequence{superURIRef},
13       label <- Sequence{slabel}
14     ),
15     slabel : owl!PlainLiteral (
16       lexicalForm <- bo.classDef.extends.testString().typeName()
17     ),
18     superURIRef : owl!URIReference (uri <- surid),
19     surid : owl!UniformResourceIdentifier (
20       name <- thisModule.ontologyName() + '#' + thisModule.classURI(bo.classDef.extends,bo)
21     ),
22     ur : owl!URIReference (uri <- urid),
23     urid : owl!UniformResourceIdentifier (
24       name <- thisModule.ontologyName() + '#' + bo.package.trim() + '.' + bo.name.trim()
25     ),
26     className : owl!PlainLiteral (lexicalForm <- bo.name.trim())
27 }
```

Listing 4.5: `Businessobject2Class` ATL rule

Considering the `propertyForDomain` attribute, the binding specified in the rule is different from the others viewed until now (line 9). In this case, the `propertyForDomain` is defined by recurring to an OCL expression, which basically iterates over each `Property Type` element in the `element` attribute of the BDO and explicitly calls the Called Rule `Property2ObjectProperty`, which returns an `OWLObjectProperty`, as indicated in the previous paragraph. Each `Property Type` element corresponds to one of the attributes of the BDO, such as the one in line 11 of Listing 4.2. This OCL expression basically allows to transform from a `Sequence` of `PropertyTypes` in the source into a `Sequence` of `OWLObjectPropertys` in the target. One could think that an alternative way of specifying this “mapping” would be to simply state `propertyForDomain <- bo.element` and define the `Property2ObjectProperty` rule, not as a Called rule but as a matched rule on `PropertyType`, because, in that case, the rule would be implicitly called for each `bo.element`. This is not the case because the `Property2ObjectProperty` rule receives one more argument besides the `PropertyType` one, which makes the matched rule impossible because those do not support arguments.

The `Property2ObjectProperty` rule transforms `PropertyType`s, from the `bdo.ecore`, into `OWLObjectProperty`s, from the `owl.ecore`, for which the following attributes were specified:

- The `uriRef` attribute, which is the URI of the property, in this case;
- The `domain` attribute, which is the `OWLClass` of the BDO which the property belongs to;
- The `range` attribute, which is the `OWLClass` conforming to the property's type (primitive or a reference to another BDO);
- The `label` attribute, which is the name of the property;
- The `propertyRestriction` attribute, which is used to hold the property's multiplicity, which can either be `[1]`, `[0..1]` or `[0..*]`.

For instance, considering the BDO's attribute in line 11 of Listing 4.2, in this case, that `PropertyType` was converted into an `OWLObjectProperty` which domain is the `HistoricoPensao` `OWLClass`, which range is the `Integer` `OWLClass`, which `label` is `tipoPensao` and which `propertyRestriction` points to the string `[1]`.

To finalise the explanation of the ATL transformation, a brief description of the utility given for the defined helpers is advisable. The defined helpers were essentially for two reasons:

1. Calculate the BDOs' attributes multiplicity

   The multiplicity has to be calculated by analysing the `type` string of each attribute of the BDO being matched. This analysis is done with an helper and, from it, the following decision tree is conducted:

   - If the attribute's `type` turns out to be a primitive type:
     - If the attribute's `type` string ends with "[]" then it is the `[0..*]` multiplicity;
     - If not, it is the `[1]` multiplicity;
   - If not:
     - If the attribute's `type` string ends with "[]" then it is the `[0..*]` multiplicity;
     - If not, it is the `[0..1]` multiplicity;

2. Determine the BDO's URI

   When referencing a BDO in an attribute `type` or by extending some BDO, the reference can be given in two ways:

   - If the *fully qualified name* is given, obtaining the URI from that point is a matter of concatenating strings;

- If not, it is actually necessary to obtain that name. This is done by an helper, by looking up all the given BDOs in the input of the transformation (with OCL).

Figure 4.6 illustrates the `multiplicity` helper, which matches quite well the decision tree explained just above in point 1.

```
helper def: multiplicity(t : String): String =
  if t.isPrimitiveDatatype() and t.endsWith('[]') then
    '[0..*]'
  else if t.isPrimitiveDatatype() then
    '[1]'
  else if not t.endsWith('[]') then
    '[0..1]'
  else
    '[0..*]'
  endif
  endif
  endif;
```

Listing 4.6: `multiplicity` helper

4.1.4 *Java Transformation*

This section aims to explain the second implementation planned to specify the transformation to solve the Global Perception problem. This implementation follows the direct-manipulation approach defined in section 2.3.1, because it was defined using a GPML - Java. As mentioned before, the input and output models are specified in Ecore, which allows a proper navigation with Java over the BDO instances in the input, as well as a proper instantiation of the OWL instance to generate as output.

Since the ATL implementation has already been explained in section 4.1.3 and this one is semantically the same, in this case, the explanation will be shorter by comparing with its ATL version respective parts in each case.

The transformation was implemented as a Java class. It could have been implemented *statically*, but it was decided to go with the object-oriented way. This basically means that the implementation works by instantiating an object instance which then is able of performing the actual transformation. In this case, the inputs and outputs of the transformation are not as clearly perceived as they were in the ATL transformation, because of ATL's domain-specific syntax. Nevertheless, by looking at the attributes of the class, in Listing 4.7, one is able of identifying the input and output.

```
private Resource owl;
private String owlPath;
private List<BusinessobjectComplexType> bdos;
private StatisticsLine stats;
```

Listing 4.7: Java transformation class attributes

The output of the transformation is the `owl` attribute. This attribute is of type `Resource` because Ecore instances are always "wrapped" by an EMF Resource. Thus, this resource holds the OWL instance to result from the transformation. The `bdos` attribute corresponds to the several BDOs given as input to the transformation. The BDO's `Resource` is not present because we are only interested in reading the BDOs content, not updating it. In this case, the `List` (line 3) allows the specification of an undetermined number of BDOs as input, which was not possible with ATL. The remaining attributes are the `path` where to save the output of the transformation - the EMF Resource - and the `stats` is declared to allow measuring the execution times of the transformation, which were useful for section 4.1.6. The `BusinessobjectComplexType` class is one of the classes generated automatically from the BDO Ecore to allow the in-memory manipulation/navigation of the BDOs.

Taking into consideration the actual transformation code, the method invoked to trigger the transformation is illustrated in Listing 4.8.

```
1 public void transform () throws IOException {
2         generateDefaultClasses();
3         for (BusinessobjectComplexType bdo : bdos)
4             businessObject2Class(bdo);
5         owl.save(Collections.EMPTY_MAP);
6 }
```

Listing 4.8: Java transformation trigger method

If one remembers the ATL transformation just explained, there are some clear similarities in this case. In the ATL transformation, the first rule to be executed was the `entrypoint` rule, which was used to generate all the primitive and default types' `OWLClasses` in the target. In this case, the method `generateDefaultClasses` (line 2) is the first line of code to be executed and does exactly the same. Then, a `for` looping structure is used to iterate over all the input BDOs to transform them into `OWLClasses`, by invoking the `businessObject2Class` method. This is precisely what the Matched Rule `BusinessObject2Class` does implicitly in ATL, because of ATL's execution engine.

Figure 4.9 presents part of the `generateDefaultClasses` method. In the case of this Java method, creating the primitive and default types is done by iterating over two static variables containing the names of those types, and then by invoking the methods `createPrimitiveType` and `createDefaultType`. These two methods, instantiate `OWLClasses` just like the `rule1` rule in Listing 4.4 does. `rule1` replicates the pattern in Listing 4.4 for each default and primitive type to be declared - that is why it is the most extensive rule. The `generateDefaultClasses` behaves the same way, in a more modular form because of the looping structures (lines 5 and 9).

```java
private void generateDefaultClasses() {
    ...
    //Generate primitive types
    Set<String> aux = new HashSet<String>(getPrimitiveTypes().values());
    for (String pt : aux)
        createPrimitiveType(pt);

    //Generate default types
    for (String dt: getDefaultTypes())
        createDefaultType(dt);
}
```

Listing 4.9: Java's `generateDefaultClasses` method

Finally, the last Java method to address is the `businessObject2Class` one, which maps each BDO (`BusinessobjectComplexType`) into an `OWLClass`. The corresponding ATL rule is the `BusinessObject2Class` Matched Rule. Listing 4.10 presents the Java method. The first line of code creates the `OWLClass` with the `createClass` method (line 2), which receives the value for the `label` attribute in the first parameter and the URI value in the second parameter of the BDO. By observing these two values, one can confirm that they are the same in the ATL transformation. For instance, in the case of the `label` value, in Java, the expression which represents it is `bdo.getName().trim()` (line 3 Listing 4.10) whereas in ATL is `bo.name.trim()` (line 26 Listing 4.5).

Continuing to observe the Java implementation in Listing 4.10, between lines 7 and 13, the code inside the `If` statement guarantees that, if necessary, the `superclass` attribute is assigned just like in the ATL version (line 8 Listing 4.5). To determine the super BDO's URI an auxiliary method called `getClassURI` (line 10) was specified, similarly to the `classURI` helper in ATL. Lastly, the `for` loop in the end of the Java method is there to make sure the all the attributes (`PropertyType`) of the BDO, residing in the `element` tag, are mapped into `OWLObjectPropertys`, like rule `Property2ObjectProperty` does in ATL. Besides this, the `createAnnotationProperty` is a method invoked for each attribute of the BDO that generates an annotation tag, for one to be able of specifying the attributes' multiplicity as a comment. This procedure was also carried out in the ATL transformation by a Matched Rule called `Property2Annotation`.

```java
private void businessObject2Class(BusinessobjectComplexType bdo) {
    OWLClass c = createClass(
            bdo.getName().trim(),
            getDefaultOntologyName() + "#" + bdo.getPackage().trim() + "." + bdo.getName().
                trim()
            );

    if (bdo.getClassDef() != null) {
        String extendsStr = bdo.getClassDef().getExtends().trim();
        if (extendsStr != null && !extendsStr.isEmpty()) {
            OWLClass sc = createClass(getClassSimplename(extendsStr), getClassURI(extendsStr
                ,bdo));
            c.getSuperClass().add(sc);
        }
    }

    for (PropertyType pt : bdo.getElement()) {
        OWLObjectProperty objprop = createObjectProperty(pt,c,bdo);
        c.getPropertyForDomain().add(objprop);
        createAnnotationProperty(pt);
    }
}
```

Listing 4.10: Java's `businessObject2Class` method

The idea to convey in this section was that there are several similarities in the ATL and Java implementations.

4.1.5 *Transformation Output*

In this section, the output of the transformation, using both implementations, is presented. There were a total of 328 BDOs given as input to the transformation, which resulted in the following OWL output model.

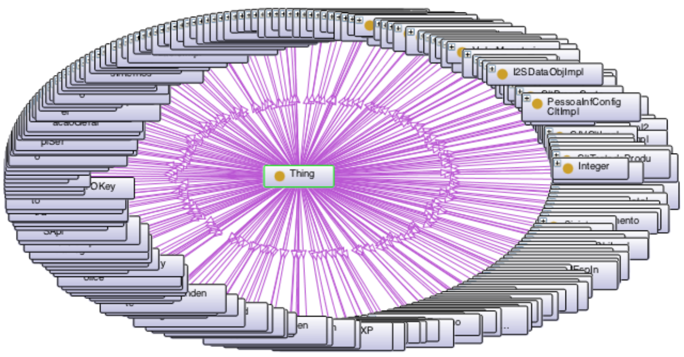


Figure 16.: Full output model

Figure 16 illustrates the complete OWL output model. In this case, all the BDOs are being displayed. As mentioned previously, this is not a useful display of the system because BDOs are overlapping each other, making it confusing. The `Thing` box in the centre of the diagram is the `OWLClass` in the top of the hierarchy as explained when introducing OWL in section 4.1.2.

Nevertheless, this is not the only view Protégé has to offer. It is possible to display only a specific BDO and, incrementally expand from there. Figures 17 and 18 represent this expansion process.

Figure 17.: The `Sinistro` BDO `OWLClass` collapsed

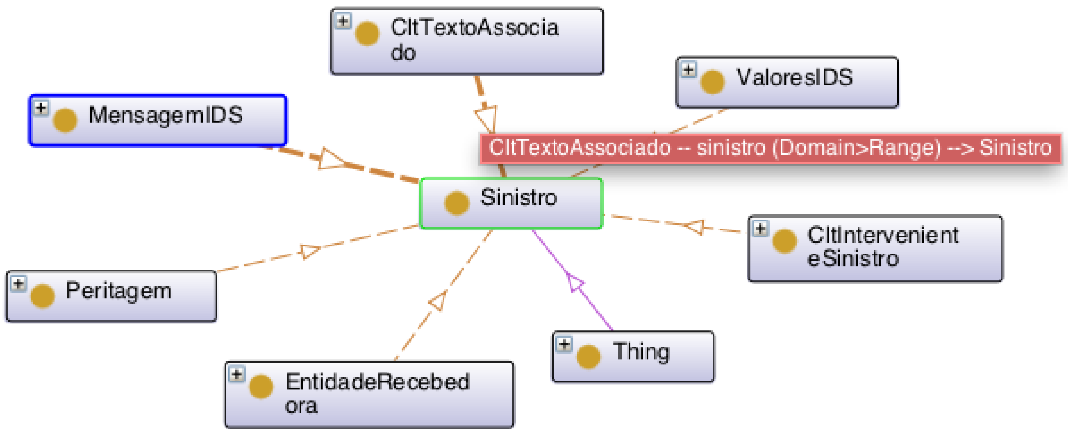
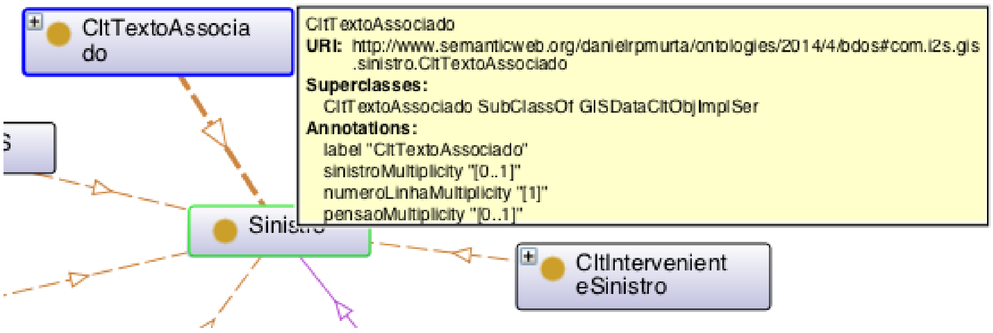


Figure 18.: The `Sinistro` BDO `OWLClass` expanded

The network in figure 18 is obtained after clicking in the + sign in figure 17. In this case, the `Sinistro` BDO has been expanded. After expanding, it is possible to realise that there are several BDOs with attributes pointing to the `Sinistro` one. The `CltTextoAssociado` BDO is an example, where, by hovering the mouse over the arrow connecting it with the `Sinistro` BDO, one realises that the attribute connecting the two BDOs is named `sinistro`. If we opt to hover the `CltTextoAssociado` BDO, is is possible to check, amongst other information, the multiplicity of the `sinistro` attribute, as depicted in figure 19.



CltTextoAssociado
URI: http://www.semanticweb.org/danielrpmurta/ontologies/2014/4/bdos#com.i2s.gis.sinistro.CltTextoAssociado
Superclasses:
CltTextoAssociado SubClassOf GISDataCltObjImplSer
Annotations:
label "CltTextoAssociado"
sinistroMultiplicity "[0..1]"
numeroLinhaMultiplicity "[1]"
pensaoMultiplicity "[0..1]"

Figure 19.: `Sinistro` BDO extra information

The multiplicity is [0..1]. In addition, it is also possible to understand that the `CltTextoAssociado` BDO extends the `GISDataCltObjImplSer` BDO. Once again, by expanding `CltTextoAssociado`, one is able of visualising that extension relation graphically also. In this case, the arrow illustrating the extension is purple, shown Figure 20.

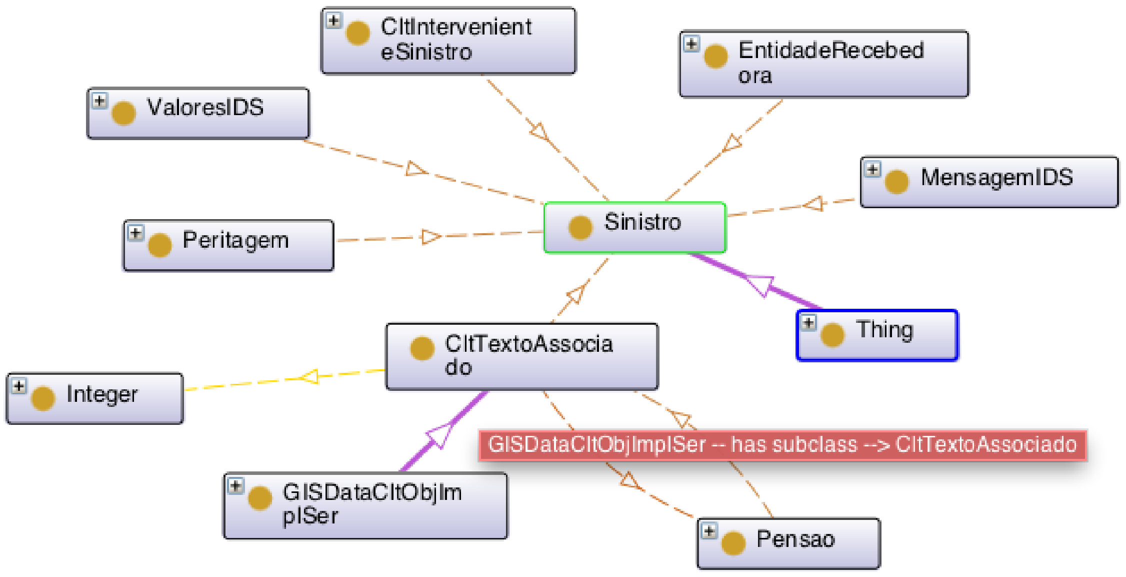


Figure 20.: The `CltTextAssociado` BDO `OWLClass` expanded

With this kind of output model, we believe the Global Perception problem is solved.

4.1.6 *Comparisson*

Having finished the two implementations planned for the solution proposed for the Global Perception problem, the comparison was the last point to address. The comparison was made so that a sustainable decision could be made regarding what is the best implementation for the transformation which had to be implemented. In order to perform this comparison, defining the comparison guidelines was the first point addressed. By "comparison guidelines" it is basically meant, in other words, what is it supposed to be compared between the two approaches. Two guidelines were thought as relevant:

1. **Performance** - One obvious and very important term of comparison should be the performance of both implementations. In this case, the goal was trying to understand which one was faster;
2. **Maintainability** - Maintainability can be defined as the easiness with which one is able of maintaining and evolve each implementation in the future. This term is directly related with questions like "Is it easy to for someone to understand the implementation at first sight?", "Would it be easy to modify the implementation in the future in order for it to meet some new requirement?", etc.

Taking into consideration the performance parameter and bearing in mind that the transformation, in this case, receives several BDOs to merge into one OWL model ($BDO^* \longrightarrow OWL$), two test cases were elaborated:

- *Test 1* - One BDO at a time

  This test was conceived to be able of understanding the relation between the performance of each implementation regarding the size of the BDOs. The size of each BDO was defined in two ways:

  – The actual size of the BDO file in Kilobytes (KB);

  – The number of attributes the BDO contains, which corresponds to the number of XML elements with the tag `element` the BDO contains;

- *Test 2* - Groups of BDOs at a time

  Since the transformation is supposed to operate with several BDOs, not one typically, this test case was considered to analyse how the performance of each implementation varies depending on the number of BDOs passed as input.

For *Test 1*, each implementation was executed for each available BDO, one at a time, and the execution times were registered for each different size. Since there are some sizes shared between different BDOs, in those cases, the time registered would be the average of the times registered for each of the BDOs sharing the size. This resulted in the charts in figure 21.

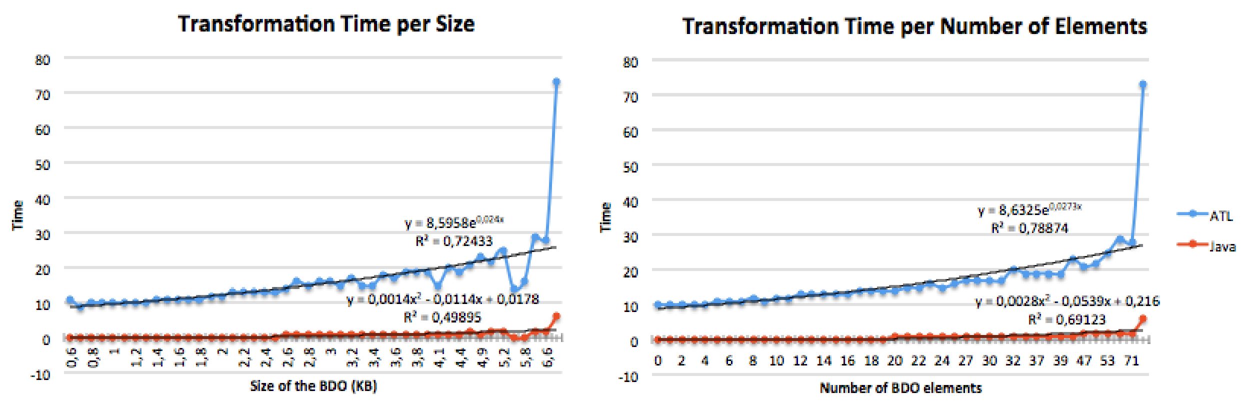
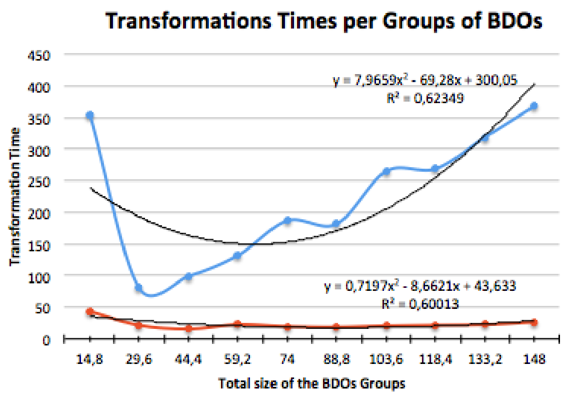


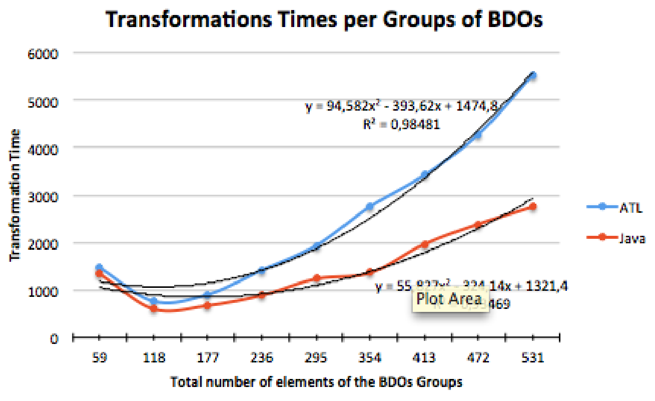
Figure 21.: ATL and Java Transformations' execution times by size charts

In both cases, the ATL transformation was best fitted by an exponential trendline, whereas the Java transformation was best fitted by a polynomial one. Nevertheless, the R^2 measure, which is a number that indicates how well data fits a statistical model, was more convincing in the chart with the number of elements as the *x-axis*. This is somehow expected, because, as mentioned previously when explaining the Java and ATL implementations, one of the procedures that both implementations have to deal with is finding the URI of referenced BDOs, in attributes/`elements`, for instance. This is the "heaviest" process of the transformation because it involves searching in the entire set of BDOs given as input. Thus, it is logical that the more attributes/number of `element` tags a BDO contains, more times this process will have to be called. This is why the number of elements in each BDO is probably the best indicator to regard. This is related to a possible explanation for the "inconsistent" points in both curves in the left hand side of Figure 21. These points have 5.6 and 5.8 as *x-axis* coordinates and they are considered "inconsistent" because they do not follow the curve trendline. After observing the correspondent BDOs it was verified that both of them, in spite being large files comparing to most BDOs, do not have an equally big number of attributes. For instance, the BDO corresponding to the size 5.2 KB has 53 attributes, while the BDO corresponding to the size 5.6 KB only has 12.

Taking into consideration *Test 2*, in this case, the implementations were tested with groups of BDOs. The size of the group being tested in each iteration varied linearly - it started with a group were the combined size of the BDOs were 14.8 KB, and from then on, the size kept being incremented with that value, by including more BDOs in the group. The BDOs composing each group were chosen randomly. After measuring the execution times, the charts in figure 22 were obtained.



(a) A subfigure



(b) A subfigure

Figure 22.: ATL and Java Transformations' execution times by groups size charts

In this case, both transformations were best fitted with a polynomial trendline and once again the R^2 measure was much better when considering the number of elements as the size parameter.

Bearing this in mind, when considering the performance term of comparison, the Java implementation is proven to be better than the ATL one. In both tests, the Java implementation was fitted by a polynomial function, whereas the ATL transformation is exponential in the first test. Furthermore, the Java transformation curve always was clearly below the ATL one, which means that Java was more efficient than ATL.

Shifting focus to the Maintainability comparison, the evaluation is not so linear to address. Evaluating for instance how easy it is to modify an implementation in order for it to meet some new requirement is not linear because it is not easy to formulate tests pointing on that direction. A similar analysis like the one made for Performance, now for Maintainability, was left out of the scope of this dissertation, in spite of being regarded as a valuable guideline of comparison. Heitlager et al. (2007) provides a set of measures for estimating maintainability on the basis of a system's source code. One of those measures is *volume*, which can be measured by counting, for instance, the number of lines of code of the transformations, in this case. Taking into account this measure, the Java implementation ended up with a total of 401 lines of code whereas the ATL implementation ended up with 573 lines of code. Consequently, it is legitimate to say that the Java implementation outweighs the ATL one regarding this point. Nevertheless, bearing in mind that ATL is a DSL created for the specification of transformations and Java is a GPML, the ATL transformation probably is a better choice in terms of maintainability. As long as one takes the overhead of learning ATL, understanding and specifying an ATL transformation should be easier than doing the same with Java code. Besides this, the 401 lines of code declared for the Java implementation only take into account the actual Java class coded to implement the transformation. However, for running that code, it was necessary to implement some auxiliary code like the one which loads the BDOs that are then fed to the implementation. This type of code is completely given "for free" when opting to use ATL.

To sum up, the final decision regarding what implementation to choose to implement the transformation is not a direct one. If the transformation is expected not to evolve in the future, Java should be the best way to implement. However, if maintainability is an important concern, in spite of being less efficient, ATL should be considered as a valid choice because it is a much more oriented implementation towards its purpose than Java, making it easily maintainable in the long term. Nevertheless, if performance is definitely a key factor, then Java is the implementation to opt for.

4.2 THE SUITABILITY FOR INSURANCE EXPERTS PROBLEM

The Suitability for Insurance Experts problem exists because the BDO language encompasses more concepts than the ones an insurance expert is able to deal with. This results in insurance experts not being able of using the BDO language themselves without having to ask for help in order to specify those concepts they are not acquainted with.

4.2.1 *Solution*

Views were the solution opted for this problem. The main idea is to be able of converting a BDO into a different format, referred as *view*, which the insurance experts can deal with and edit as they wish, having then the modifications made be automatically propagated into the view's corresponding BDO, so that both artifacts remain consistent/synchronised. Thus, basically, a BDO's view corresponds to a filtered version of the BDO, where only the domain concepts which the insurance experts are supposed to deal with are present.

In practical terms, this solution corresponds to a bidirectional transformation, because, in the one hand, we are interested in the transformation that generates a BDO's view and, on the other hand, we are interested in propagating the changes made in the views into the BDOs. As a matter fact, by the time the transformation is bidirectional, updates in both sides of the transformation are possible, because the transformation guarantees that those are propagated into "the other side". This is exactly the case because BDOs will continue to exist and will be maintained by the company's software engineers and the views will also exist and will be used by the insurance experts, so updates will exist on both sides of the transformation.

When speaking about bidirectional transformations, the idea of Source and Target becomes dependant on the direction chosen to execute the transformation. Bearing this in mind, it is preferable to mention the "domains" the transformation processes. One of those domains is clearly BDOs, which are specified by the BDO language. The other domain is not so clear by now because, in order to be able of specifying the BDO's views, firstly it is necessary to have a language to do so. Once again, choosing between a DSL and a GPL was necessary. In this case, since the views were supposed to be edited by insurance experts, a DSL sounded as the best solution. The DSL should have a textual syntax and, at the same time, it should resemble "normal" text, not programming code. Taking this into account, the DSL could not be an XML language, certainly. In order to “have” a DSL like this, one of two options could be used: one could try to search for a language with those features, already implemented; or could opt for creating a new DSL, from scratch. The second option was the one chosen, and Xtext Eclipse (2013b) was the framework used for it.

There were two reasons for opting for Xtext when defining the new DSL:

1. Xtext is an Eclipse framework specifically implemented for the specification of DSLs. It works by having the user specify the grammar of the language first and then a representative Ecore model of that grammar is automatically generated, for the *parsing* process to be possible. This is very useful because it basically means that the views' *abstract syntax* (model) is specified in Ecore, just like the BDO language, which allows the transformation implementations to work in the Ecore *technical space*, just like what happened with the unidirectional transformation. This is very convenient.

2. When one uses a language in some IDE, there are some features that the user just takes for granted because of their utility. Automatic completion, error quick fixing... are some of those features. Xtext allows a very easy implementation of these features.

Taking this into account, the view's language was specified in Xtext and it was given the extension **vbdo**, from “View Business Domain Object”. Thus, the signature of the bidirectional transformation to specify is depicted by the type $BDO \longleftrightarrow VBDO$.

It is now possible to classify the transformation according to the taxonomy defined in section 2.3, just like what was done with the unidirectional transformation:

- Output

  This is a *Model to Model* transformation for the same reasons specified before in section 4.1.1. Since both domains of the transformation conform to models specified in Ecore, to which we have access, it makes more sense to have model to model transformations, instead of model to text, for instance.

- Cardinality

  This is a *one-to-one* model transformation because for each BDO the transformation returns one corresponding view and vice-versa.

- Source/Target Metamodels

  This is an *exogenous/translation* transformation because the domains involved do not conform to the same model;

- Source/Target Abstraction Levels

  Since the BDOs' views are subsets of BDOs, containing only parts of the correspondent BDOs, they can be regarded as abstractions of BDOs. Taking this into account, the transformation is vertical.

- Directionality

  This is a bidirectional transformation for the reasons stated before.

The following sections present the two implementations defined for the bidirectional transformation. Nevertheless, before the implementations, a brief explanation of the VBDO language is given. In contrast to how the unidirectional transformation was specified, in this case, the first implementation to be explained is the one implemented with the GPL (Java). The second implementation to be explained was defined in QVTr, which is a DSL designed for the specification of bidirectional transformations. Considering this, once again, the decision between using a GPL or a DSL surged, however, in this case, the case study is a bidirectional transformation.

VBDO language

This section aims to describe the VBDO language, so that one understands how the BDOs will be mapped. For that, parts of the grammar of the language are illustrated in Listing 4.11. The full grammar is in Appendix .3. A grammar specified in Xtext is composed of rules that indicate how to write something valid using the language. For example, by looking at the `Field` rule (lines 30-32), we can conclude that

```
- Many object(s), here referring to Bytes
```

is a valid specification of a `Field` element. This is because the `Field` rule (lines 30 and 31) starts precisely with the `-` sign, followed by the specification of a `Multiplicity`, which can be a `SpecificMultiplicity`, a `ManyMultiplicity`, a `PossiblyOneMultiplicity` or a `OneMultiplicity` (lines 36-41). In this example, the multiplicity chosen was the `ManyMultiplicity` because the word that follows the `-` sign is `Many`, as defined by the `ManyMultiplicity` rule (line 38). Then, after the multiplicity, an `InlineAlias` follows, which forces the specification of a `name` for the `Field` (line 34), which is `object` in this case, possibly followed by `(s)` to denote the plural, followed by `, here referring to` and ending with a `PrimitiveType`, a `DefaultType` or a reference to an `Entity` (because of the square brackets), as illustrated in line 34. In this case, the field ends with `Bytes`, which is one of the primitive types declared in the grammar (line 19).

```
1  grammar org.xtext.domainmodel.DomainModel with org.eclipse.xtext.common.Terminals
2
3  generate domainModel "http://www.xtext.org/domainmodel/DomainModel"
4
5  Model:
6    ('Model:' | 'model:') name = QualifiedName
7    (imports = Imports)?
8    entities += Entity+
9  ;
10
11 Imports:
12   ('Read:' | 'read:') imports += Import+
13 ;
14 Import:
15   '+' importedNamespace = QualifiedNameWithWildcard
16 ;
17
18 enum PrimitiveType:
19   CharacterSequence | Integer | Real | Date | Boolean | Character | Bytes
20 ;
21 enum DefaultType:
22   CTLObjBaseImpl2 | GISCfgDataObjImplSer | GISDataCltImpl |GISDataCltObjImplSer |
23   GISDataObjImplSer | GIVCltImpl | GIVObjImpl | I2SCltImpl | I2SCurrencyAmount |
24   I2SDataObjImpl | I2SDataObjSerializableImpl | I2SHome
25 ;
26
27 Entity:
28   article = ('A' | 'An' | 'a' | 'an')? name = ID ('is a' (superEntity = [Entity |
         QualifiedName] | superDefault = DefaultWrapper) and = ('and')?)? ('has:' fields +=
         Field+)?
29 ;
30 Field:
31   '-' multiplicity = Multiplicity inlineAlias = InlineAlias
32 ;
33 InlineAlias:
34   name = ID (plural='(s)')? ', here referring to' (type = PrimitiveTypeWrapper | default =
         DefaultWrapper | entity = [Entity | QualifiedName])
35 ;
36 Multiplicity:
37   {SpecificMultiplicity} number = INT |
38   {ManyMultiplicity} ('Many' | 'many') |
39   {PossiblyOneMultiplicity} ('Possibly one' | 'possibly one') |
40   {OneMultiplicity} ('One' | 'one')
41 ;
```

Listing 4.11: VBDO's language grammar

By looking at the grammar, it is easy to realise the concepts from the BDO language we are interested in:

1. Each file specified using the VBDO language starts with the specification of the BDO's package - line 6. `QualifiedName` is a special string recognised by Xtext;

2. Then, following the package are, possibly, the imports indicated in the BDO - line 7. “Possibly” because of the `?` sign, which is plausible because BDOs do not necessarily have to declare imports;

3. Following the imports is the actual BDO specification, which in the VBDO language corresponds to the `Entity` concept - line 8. In this case, more than one `Entity` is expected, because of the + sign, which may be odd because each BDO file only encompasses one BDO. In this case, the other entities correspond to local BDOs referenced by the main BDO. This happens because, there are cases of BDOs which make references to other BDOs that actually do not exist anywhere (these BDOs are clearly inconsistent). In these cases, Xtext throws an error stating that the references are not valid. In order for this error not to appear, the missing BDOs are declared locally, in the same BDO file. The other entities exist for these local BDOs declarations.

4. Then, by observing the `Entity` definition (lines 27-29), is is possible to see that it supports:
   - A `name` attribute, which corresponds to the name of the BDO. This name may be preceded by an article, just for the sake of a correctly constructed statement;
   - A `superEntity` attribute, which corresponds to the BDO’s parent, the BDO which is extended;
   - A group of `fields`, which corresponds to the BDO attributes (name, type and multiplicity for each), which are the `element` tags in the BDO XML language.

Bearing this in mind, the text in Listing 4.12 would be a valid BDO view.

```
Model: com.i2s.policy

Read:
    + com.i2s.products.CarProduct
    + com.i2s.policy.Policy

A CarPolicy is a com.i2s.policy.Policy and has:
    - One brand, here referring to CharSequence;
    - Many seat(s), here referring to Integer;
    - One class, here referring to CharSequence;
```

Listing 4.12: Example of a valid view of a BDO

4.2.2 *Java Transformation*

This section aims to explain one of the implementations defined to specify the transformation to solve the Suitability problem. In this case, the implementation defined follows the *lenses* approach defined before in section 2.3.2, which means that two unidirectional transformations had to be specified: the *get* transformation; and the *put* transformation. Bearing this in mind, this section is divided in two parts, corresponding to those two unidirectional transformations. These two transformations were specified using Java.

In order for this transformation to be a *well behaved* one (term defined in section 2.3.2), conditions GetPut and PutGet had to be taken into account. To guarantee that the implementations had the expected behaviour and, essentially, that those conditions were not being violated, some *unit tests* were specified, and none of them failed. Since the specification of the transformation was done following this *ad-hoc* approach of implementing everything in Java, the unit tests were defined in order to have some kind of higher assurance of the correctness of the implementation. The procedure carried out for testing is explained later in this section.

Get

The *Get* transformation receives a BDO in the input and outputs the corresponding view ($BDO \longrightarrow VBDO$). This transformation was specified as a common Java class. By observing the classes's attributes, one can identify both the input and output of the transformation (Listing 4.13). The `bdo` attribute corresponds to the input and the `m` attribute corresponds to the output. The `vbdo` attribute corresponds to the EMF resource to hold the output model, as the EMF rules dictate. The `bdo`'s Resource is not present once again because we are only interested in reading (not updating) the BDO. The remaining attributes are generally necessary for the transformation to occur.

```java
private Resource vbdo;
private BusinessobjectComplexType bdo;
private Model m;
private Set<String> imports;
private Set<Entity> entities;
```

Listing 4.13: Java *Get* transformation attributes

Next, the actual transformation is explained in detail. In order to do so easily, one BDO and the corresponding VBDO obtained by applying the transformation are given as example in Listings 4.14 and 4.15, respectively.

```
 1 <I2sbusinessobject:BusinessobjectComplexType package="com.i2s.gis.peritagem" name="
       HistoricoPeritagem">
 2     <import>com.i2s.utilities.Process</import>
 3     <interfaceDef extends="GISDataObj">
 4         <import>com.i2s.gis.GISDataObj</import>
 5     </interfaceDef>
 6     <classDef extends="GISDataObjImplSer">
 7         <import>com.i2s.gis.GISHome</import>
 8     </classDef>
 9     <element name="peritagem" read="true" type="Peritagem" write="false"/>
10     <element name="sequencia" read="true" type="int" write="true"/>
11     <element name="situacao" read="true" type="com.i2s.Situacao" write="true"/>
12     <element name="estado" read="true" type="com.states.Estado" write="true"/>
13     <element name="dataCriacao" read="true" type="Data" write="true"/>
14     <element name="horaCriacao" read="true" type="int" write="true"/>
15     <element name="utilizador" read="true" type="String" write="true"/>
16 </I2sbusinessobject:BusinessobjectComplexType>
```

Listing 4.14: Input BDO

```
 1 Model: com.i2s.gis.peritagem
 2
 3 Read:
 4     + com.i2s.Situacao
 5     + com.states.Estado
 6
 7 A HistoricoPeritagem is a GISDataObjImplSer and has:
 8     - Possibly one peritagem, here referring to Peritagem
 9     - One sequencia, here referring to Integer
10     - Possibly one situacao, here referring to com.i2s.Situacao
11     - Possibly one estado, here referring to com.states.Estado
12     - One dataCriacao, here referring to Date
13     - One horaCriacao, here referring to Integer
14     - One utilizador, here referring to CharacterSequence
15
16 Peritagem
```

Listing 4.15: Output VBDO

The principal method of this transformation is illustrated in Listing 4.16 and its name is `bdo2vbdo`. First of all, the root of the BDO's view is created by the `createModel` method (line 2). This method instantiates a `Model` object with the BDO´s package value in the `Model`'s corresponding `name` attribute. By looking at the example given for the transformation, it is possible to see this step of the transformation by looking at the first line of the output VBDO (Listing 4.15) where the `Model` is named the value of the `package` attribute in line 1 of the input BDO (Listing 4.14).

Then, back on the `bdo2vbdo` method in Listing 4.16, the actual `Entity` instance to hold the BDO is instantiated, with the name of the BDO (line 3), and added right after into the just created model instance's `entities` (line 4). The following `if` structure is there to guarantee that, in case the BDO extends some other BDO, that information is also mapped into the BDO's entity in the output model. In that case, that reference is saved in the `superEntity` or `superDefault` attributes of the `Entity` class, in case the BDO's extension is another BDO or a default BDO, respectively. In the example given, we can see that `HistoricoPeritagem is a GISDataObjImplSer` in line 7 of Listing 4.15. By observing the VBDO language grammar in Listing 4.11 it is possible to note that `GISDataObjImplSer` is a `DefaultType`, thus, in this case, the BDO extends a default BDO.

The next and penultimate `if` structure in Listing 4.16 maps each attribute of the BDO into the view. This is done by iterating over the BDO's `elements` (lines 19-22) and then converting them into `Fields`, which are added to the BDO's entity `fields` attribute (line 21). For instance, the `peritagem element` in line 9 of Listing 4.14 is mapped to the `Field` in line 8 of Listing 4.15. The multiplicity, in this case, is `Possibly one` because the type of the attribute is a BDO (`Peritagem`). The next field is called `sequencia` and, now, the multiplicity is `one` because the type of the attribute is a primitive one.

Finally, the last `if` structures (lines 25-28) are present just to make sure that the *syntatic sugar* of the VBDO language is used: the articles ("A", "An", "a" and "an") and conjunction ("and") are set, in case that makes sense.

```
 1 private void bdo2vbdo() {
 2         m = createModel();
 3         Entity e = createEntity(bdo.getName().trim());
 4         m.getEntities().add(e);
 5         e.setAnd(null);
 6 
 7         boolean hasSuper = false;
 8         ClassDefComplexType cd = bdo.getClassDef();
 9         if (cd != null) {
10             if (cd.getExtends() != null && !cd.getExtends().trim().isEmpty()) {
11                 hasSuper = true;
12                 treatSuper(e,cd.getExtends().trim());
13             }
14         }
15 
16         boolean hasFields = false;
17         if (bdo.getElement() != null && !bdo.getElement().isEmpty()) {
18             hasFields = true;
19             for (PropertyType pt : bdo.getElement()) {
20                 Field f = treatField(pt);
21                 e.getFields().add(f);
22             }
23         }
24 
25         if (hasFields && hasSuper) e.setAnd("and");
26         if (hasFields || hasSuper)
27             if (isVowel(e.getName().charAt(0))) e.setArticle("An");
28             else e.setArticle("A");
29 
30         treatImports();
31     }
```

Listing 4.16: Method `bdo2vbdo` of the *Get* Java transformation

Two final considerations about the *Get* implementation remain to be given. The first point to mention, regarding the example, is the absence in the VBDO output of the only import stated in the BDO file. This happened because the *Get* transformation filters unnecessary imports. In this case, the import in question is `com.i2s.utilities.Process` (Listing 4.14 line 2), which, by looking at the VBDO instance in Listing 4.15 is not used anywhere. Bearing this in mind, including this import would be meaningless. The fact that the *Get* implementation filters unimportant imports has implications in the *Put* implementation, because, the information of the BDOs that is disregarded by the *Get* must be recovered afterwards by the *Put*, for the conditions GetPut and PutGet to be respected. This will be noted in the next section.

The final point to mention is associated with the `Peritagem` Entity in the VBDO output (Listing 4.15 Line 16). This is an example of an entity created locally to solve cross-references inconsistencies in the BDO. By observing the BDO example in Listing 4.14, we can see that the first `element` is of type `Peritagem`. `Peritagem` is not a primitive type nor a default type, so it must be a reference to another BDO with that name. By looking at the imports stated in the BDO, none of them ends with `Peritagem`. Bearing this in mind, `Peritagem` is assumed to be a local BDO and, as consequence, it was declared the way it was in the VBDO output. The only imports that are declared in the VBDO (lines 4-5) are the ones used throughout the model. In this case, the first one is used in line 10 and the second in line 11.

Put

The *Put* transformation is the other part that remains to be explained for the *lenses* approach to be complete. This transformation was also specified as a Java class. The signature of the transformation is $BDO \times VBDO \longrightarrow BDO$: instead of receiving just a VBDO to return afterwards the updated BDO, the *Put* also receives a BDO, which corresponds to the BDO which originally generated the VBDO (by applying the *Get*), which may have been updated ever since. The BDO in the input is basically the *pre-state* that guarantees that the information lost after applying the *Get* is possible to recover when applying the *Put*.

Listing 4.17 shows some of the transformation's class attributes. The input pair is represented by the `bdo1` and `vbdo2` attributes. The `bdo2` attribute corresponds to the output BDO of the transformation and the `bdo2Rec` is its EMF Resource. The `theBdoInvbdo2` corresponds to the `Entity` instance that holds the BDO in the VBDO input. The `bdo1` and `bdo2` attributes, in the beginning of the transformation, are equal. What the transformation does is refactor the `bdo2` attribute according to the updates made in the `vbdo2`. Bearing this in mind, this is an *out-place* implementation.

```
private Resource bdo2Rec;
private BusinessobjectComplexType bdo1, bdo2;
private Model vbdo2;
private Entity theBdoInvbdo2;
```

Listing 4.17: Attributes of the *Put* Java transformation

The method that triggers the *Put* transformation is illustrated in Listing 4.18 and is called `bdoXvbdo2bdo` and its code represents very well the way the transformation was specified. Bearing this in mind, the explanation of the implementation will be done by detailing each line of code in the method. In the end, an example of the execution of the transformation is given to illustrate the explanation presented, as done previously with the *Get*.

```
1  private void bdoXvbdo2bdo() {
2      //Set the package of bdo2 to the new one possibly defined in vbdo2
3      bdo2.setPackage(vbdo2.getName());
4
5      //Set the name of bdo2 to the new one possibly defined in vbdo2
6      bdo2.setName(theBdoInvbdo2.getName());
7
8      //Set bdo2's superclass to the possible new one defined in vbdo2
9      setBdoSuperclassInBdo2();
10
11     //Iterate over the properties of the bdo1 to see if anyone maintains
12     checkForChangedPropertiesInBdo1();
13
14     //Iterate over the remaining properties of the new vbdo
15     addUnprocessedFieldsInVbdo2ToBdo2();
16
17     //Include necessary imports in bdo2
18     treatImports();
19 }
```

Listing 4.18: Method `bdoXvbdo2bdo` of the *Put* Java transformation

The transformation starts by updating the BDO's `package` with the corresponding value in the VBDO model (line 3). Then, the actual content of the BDO is updated. The first attribute to be updated is the `name` of the BDO, which may have changed (line 6). Then the BDO extension is dealt with. The method `setBdoSuperclassInBdo2` (line 9) does this process, by updating, deleting or even creating the `classDef` element in the `bdo2` instance.

Then the attributes/properties (`element` tags) of the BDO are updated. The `element` tags correspond to the `Fields` in the VBDO. As one can see by looking at Listing 4.18, the process of updating the BDO's `elements` starts by dealing with the changed ones (line 12) and after that with the new ones, by adding them to `bdo2` (line 15). Changed properties are `elements` that already existed before in the BDO *pre-state* (`bdo1` instance), and that were maintained or updated in the BDO's view (`vbdo2` instance) as `Fields`. To check if some `element` is a changed one, the transformation tries to locate a `Field` with the same name in the `vbdo2` object: if it does find, that `element` has been changed or was simply maintained the same; if it does not, the `element` is eliminated from the `bdo2` instance. *New* `elements` are added to the BDO's `Entity` in the VBDO model as `Fields`, which did not exist previously in the corresponding BDO. Dealing with the BDO's `element` tags this way, instead of simply converting the VBDO's fields into elements in the `bdo2` instance, is completely different, as the example of the *Put* execution to detail later will demonstrate.

The last method invoked is the `treatImports`. This method adds the imports defined in the `vbdo2` instance into the `bdo2` instance and guarantees that the imports that used to exist in the `bdo1` instance are also maintained if necessary. This is done, to guarantee the *well behaviour* rules are not violated and is necessary because the *get* implementation filters unnecessary imports, as mentioned before.

Listings 4.19, 4.20 and 4.21 illustrate an example of the application of the *Put* transformation. Listings 4.19 and 4.20 are the BDO and the VBDO given as input, respectively, and listing 4.21 is the BDO generated in the output.

```
<I2sbusinessobject:BusinessobjectComplexType package="com.i2s.ctl.reclamacao.cfg.dto" name="
    ReclamacaoCfg">
    <interfaceDef extends="com.i2s.serializable.bdo.I2SData"/>
    <classDef extends="I2SDataObjSerializableImpl"></classDef>
    <element name="reclamacaoCfgDTOKey" read="true" type="com.i2s.ctl.reclamacao.cfg.dto.
        ReclamacaoCfgDTOKey" write="true" />
    <element name="reclamacaoInfDTO" read="true" type="com.i2s.ctl.reclamacao.cfg.dto.
        ReclamacaoInfDTO[]" write="true" />
    <element name="oid" read="true" type="int" write="true" />
    <element name="area" read="true" type="String" write="true" />
    <element name="descricao" read="true" type="String" write="true" />
    <element name="accaoCorretiva" read="true" type="String" write="true" />
    <element name="allReclamacaoCfgKeys" read="true" type="ReclamacaoCfgDTOKey[]" write="
        true" />
</I2sbusinessobject:BusinessobjectComplexType>
```

Listing 4.19: BDO given as input to the *Put*

```
Model: com.i2s.ctl.reclamacao.cfg.dto

Read:
    + com.i2s.ctl.reclamacao.cfg.dto.ReclamacaoCfgDTOKey
    + com.i2s.ctl.reclamacao.cfg.dto.ReclamacaoInfDTO
    + com.i2s.seccoes.Seccao

A ReclamacaoCfg has:
    - Possibly one reclamacaoCfgDTOKey, here referring to com.i2s.ctl.reclamacao.cfg.dto.
        ReclamacaoCfgDTOKey
    - Many reclamacaoInfDTO(s), here referring to com.i2s.ctl.reclamacao.cfg.dto.
        ReclamacaoInfDTO
    - One oid, here referring to Integer
    - Possibly one descricao, here referring to Descricao
    - Possibly one seccao, here referring to com.i2s.seccoes.Seccao
    - One accaoCorretiva, here referring to CharacterSequence
    - Many allReclamacaoCfgKeys(s), here referring to com.i2s.ctl.reclamacao.cfg.dto.
        ReclamacaoCfgDTOKey

Descricao
```

Listing 4.20: VBDO given as input to the *Put*

```
<I2sbusinessobject:BusinessobjectComplexType package="com.i2s.ctl.reclamacao.cfg.dto" name="
    ReclamacaoCfg">
    <import>com.i2s.seccoes.Seccao</import>
    <interfaceDef extends="com.i2s.serializable.bdo.I2SData"/>
    <classDef></classDef>
    <element name="reclamacaoCfgDTOKey" read="true" type="com.i2s.ctl.reclamacao.cfg.dto.
        ReclamacaoCfgDTOKey" write="true"/>
    <element name="reclamacaoInfDTO" read="true" type="com.i2s.ctl.reclamacao.cfg.dto.
        ReclamacaoInfDTO[]" write="true"/>
    <element name="oid" read="true" type="int" write="true"/>
    <element name="descricao" read="true" type="Descricao" write="true"/>
    <element name="accaoCorretiva" read="true" type="String" write="true"/>
    <element name="allReclamacaoCfgKeys" read="true" type="com.i2s.ctl.reclamacao.cfg.dto.
        ReclamacaoCfgDTOKey[]" write="true"/>
    <element name="seccao" type="com.i2s.seccoes.Seccao"/>
  </I2sbusinessobject:BusinessobjectComplexType>
```

Listing 4.21: BDO resulted from applying the *Put* with 4.19 and 4.20 as input

Analysing the BDO and the corresponding updated view (in listings 4.19 and 4.20, respectively), both passed as input to the *Put* transformation, the following can be concluded:

- The BDO's `package` has not been changed (`com.i2s.ctl.reclamacao.cfg.dto`);
- The `name` of the BDO has not been changed (`ReclamacaoCfg`);
- Three new imports have been added in the view. Each one of them is used in the view;
- The BDO used to extend `I2SDataObjSerializableImpl`, which it does not anymore;
- `Elements reclamacaoCfgDTOKey`, `reclamacaoInfDTO`, `oid`, `accaoCorretiva` and `allReclamacaoCfgKeys` have been maintained the same in the BDO's view;
- `Element descricao` has been changed, by changing its `type` to a locally defined BDO, named `Descricao`;
- `Element area` has been removed in the BDO's view;
- `Element seccao` has been added in the BDO's view;

By observing the BDO output (Listing 4.21) by the *Put* transformation, one can draw the following conclusions:

- Only one of the imports added in the view was added to the BDO (`com.i2s.seccoes.Seccao`). This happened because the other two imports were being used in the BDO's *pre-state*, in the elements `reclamacaoCfgDTOKey` and `reclamacaoInfDTO` (Listing 4.19 Lines 4 and 5), but were never declared in the imports section. Thus, they remain undeclared. The only import added resulted from a newly created field in the view, thus it had to be added;

- The BDO no longer extends `I2SDataObjSerializableImpl` (in the `classDef` XML element there does not exists the `extends` attribute anymore), like the view states;
- The element with name `descricao` was correctly updated, because it's type is now `Descricao` and not `String` as it used to be. The reason why we may say that this `element` was, in fact, updated is because its `read` and `write` attributes remained with the same values they had in BDO *pre-state* (Listing 4.19 line 8). If the transformation opted for adding a new `element` with the `descricao` name instead, these two attributes would not even be defined (they would be false by default);
- The field `seccao` which was added in the view, has been mapped to the BDO, by a corresponding element with the same name (Listing 4.21 line 11);

Tests

In order to have a higher assurance of the correctness of the implementation of this bidirectional transformation, some unit tests were defined. The fact that this implementation was carried out in an *ad-hoc* fashion, by implementing everything from scratch in Java, was the reason for defining these unit tests. If the implementation had been done using a DSL for bidirectional transformations, the unit tests would not have the same importance they have in this case, because in a bidirectional framework, the *get* and *put* transformations are derived automatically ensuring that they are well-behaved by construction.

The tests aimed to check if the conditions GetPut and PutGet were not being violated and if updates were being propagated correctly. In order to do this, two types of tests were conducted:

- *GetPut* tests

  For each BDO *bdo*:

  1. Apply the *Get* transformation over *bdo* to obtain a VBDO *vbdo*;
  2. Apply the *Put* transformation over the pair *bdo* and *vbdo* to obtain the BDO *bdo*2;
  3. Compare *bdo*1 and *bdo*2: if they are equal, the test has passed.

- *PutGet* tests

  1. Define a series of possible updates one can perform over a VBDO;
  2. For each update *u* defined:
     a) Select one BDO *bdo*;
     b) Apply the *Get* transformation over *bdo* to obtain a VBDO *vbdo*;
     c) Perform update *u* over *vbdo*;
     d) Apply the *Put* transformation over the pair *bdo* and *vbdo* to obtain BDO *bdo*2;

e) Apply the *Get* transformation over *bdo*2 to obtain the VBDO *vbdo*2;

f) Compare *vbdo* and *vbdo*2: if they are equal the test has passed.

The *PutGet* tests focus on making sure that updates made in VBDOs are correctly mapped into BDOs. To be able of performing the update tests, first it was necessary to define some updates as point 1 emphasises. The following list of updates was outlined:

- Change the `Model`'s `name` attribute;

  Ex: `Model: com.total.products => Model: com.singular.recipes`

- Change the BDO's `entity name`;

  Ex: `A Product has: => An Object has:`

- Remove the BDO's `entity` extension BDO;

  Ex: `A Toy is a Product and has: => A Toy has:`

- Set a BDO `entity` extension pointing to a BDO;

  Ex: `A Toy has: => A Toy is a Product and has:`

- Change a BDO's `entity` extension;

  Ex: `A Toy is a Product and has: => A Toy is a Device and has:`

- Change the multiplicity of a BDO's `entity field`;

  Ex: `One oid, here referring to Integer => Many oid(s), here referring to Integer`

- Change the name of a BDO's `entity field`;

  Ex: `One oid, here referring to Integer => One objid, here referring to Integer`

- Change the type of a BDO's `entity field`;

  Ex: `One oid, here referring to Integer => One oid, here referring to CharacterSequence`

- Add a new `field` to a BDO's `entity`;

  Ex:

  ```
  One oid, here referring to Integer
  =>
  One oid, here referring to Integer
  Possibly one reference, here referring to Reference
  ```

- Remove a `field` from a BDO's `entity`;

  Ex:

  ```
  One oid, here referring to Integer
  Possibly one reference, here referring to Reference
  =>
  Possibly one reference, here referring to Reference
  ```

- Add a new `import`;

  Ex:

  ```
  Read:
  + com.i2s.support.Helper;
  =>
  Read:
  + com.i2s.support.Helper;
  + com.i2s.products.RetailProduct;
  ```

In the end, 328 *GetPut* tests were executed, which corresponds to the number of BDOs available and 15 *PutGet* tests were made, which corresponds to a test for each update mentioned above and some more which resulted from combining updates.

One last point remains to be mentioned. If one observes each one of the tests specified, one can realise that, at their last step, all of them end with a comparison between BDOs or VBDOs. In order to make these comparisons possible, it was necessary to implement two custom comparators: one for BDOs; and one for VBDOs. This happened because we were not interested in a "default" file comparison were a file equals other if they are exactly the same, considering the content, formatting, etc. In this case, two BDOs can be regarded equal even if their respective files are not exactly the same. For instance, when the attributes of two BDOs are the same but are displayed in different orders in the BDOs' files, the BDOs are considered equal, nonetheless. Bearing this in mind, we can say that the comparison made is at the abstraction level, not at the syntax level. Consequently, when "equality" is referred one should consider $\sim$ instead of the strictly equality symbol $=$. An approach like this was conducted in Foster et al. (2008).

4.2.3 *QVTr Implementation*

QVTr was the other language chosen to specify the bidirectional transformation. This section aims essentially to explain the implementation achieved in that case. Since QVTr was not introduced before in this thesis, an introduction of the language is presented before delving into de concrete implementation, just like what was done with ATL.

QVTr Introduction

QVTr stands for *QVT relational* and is a DSL implemented for one to be able of specifying bidirectional transformations. The approach followed by QVTr is a *relational* one, in the sense that, in order to specify the transformations, rather than stating the steps on how to transform from Source to Target and vice-versa (like QVTo does), one actually states equivalencies between the Source and the Target. These equivalencies normally state, in words, "if an element X exists in the source then an element Y must exist in the target and vice versa". Such approach has one clear advantage which is the fact that with a single specification, one takes care of both directions of the transformation, because equivalencies can be split into two implications. This is good for maintainability. On the other hand, being a relational approach can make some specifications in QVTr hard to achieve, the more complex they are. Normally, a complex specification defined in an operational format is equally implemented with the relational approach by a much larger specification. By complex specification we mean a specification filled with "decision branches" or *if* conditions.

QVTr transformations are implemented by defining *relations*. "Relations in a transformation declare constraints that must be satisfied by the elements of the candidate models." Object Management Group (2011). A relation is composed by two or more *domains*, variables and, possibly, a pair of *when* and *where* conditions. To explain easily the concept of a relation, Listing 4.22 displays an example of a QVTr relation.

```
 1 top relation C2T {
 2   cn: String;
 3   domain uml c:Class {
 4       persistent=true,
 5       opposite(Package::classes)=p:Package{},
 6       name=cn };
 7   domain rdbms t:Table {
 8       scheme=s:Scheme{},
 9       name=cn };
10   when { P2S(p,s); }
11   where { A2C(c,t); }
12 }
```

Listing 4.22: Example of a QVTr relation

There are two types of relations: *top-level* relations; and non *top-level* relations. The presented relation is a *top-level* one, because of the `top` keyword in line 1, which means that it must hold upon the execution of the transformation. Non *top-level* relations only have to hold when they are invoked directly or transitively from the where or when clauses of another relation.

In this case, a variable named `cn` is declared (line 2), two domains are declared (lines 3-9) and one *where* and *when* clause are also declared (lines 10 and 11). The *where* clause states that relation `C2T` is supposed to hold only when relation `P2S` holds for `Package p` and `Scheme s`. The *where* clause indicates that whenever relation `C2T` holds, relation `A2C` must also hold for `Class c` and `Table t`.

The domains of a relation are basically the elements that are being synchronised between each other. In this case, this relation states that whenever a `persistent Class c` exists (in the `uml` model), a `Table t` with the same `name` as the class must also exist (in the `rdbms` model) (lines 6 and 9). This relation is only supposed to hold for pairs of `Class`es and `Table`s whose respective `Package p` and `Scheme s` are related by relation `P2S` (line 10). The reason why lines 5 and 8 are included is simply for the specifier to be able to reference the `Package p` and `Scheme s` in the *when* and *where* clauses. Variable `cn` is a *free* variable because it is what unites the `Class` and `Table` elements - they must share the same `name` which is represented by the `cn` variable. During execution, that variable is unified with the several cases that are available.

The statements inside domains can be regarded as conditions. For example, in lines 3 to 6 we are referring to `Class`es which are `persistent`, and belong to a certain `Package p` and which `name` equals the `cn` variable. Besides these conditions, it is also possible to define more complex ones, recurring to OCL expressions. These ones go into a second pair of curly brackets, defined in front of the previous one.

To conclude, two special keywords can be attached to domains: `checkonly`; or `enforce`. By default, when nothing is declared, the transformation executer acts as if the `checkonly` keyword was attached to each domain. Supposing line 3 started with the `enforce` keyword and line 7 with `checkonly`. If the transformation was executed in the direction of the `uml` metamodel, if in the `rdbms` side there was a `Table` element and on the `uml` side there did not exist the corresponding `Class` element, that element would be automatically created. On the contrary, if the `Class` element existed and the `Table` element did not, the `Class` would be deleted. This is what the `enforce` keyword does - it makes the transformation proactive. `checkonly` only returns warnings whenever the `enforce` would act.

QVTr Transformation

This section explains the specified QVTr transformation. Nevertheless, before delving into the actual specification, it is important to clarify that, in this case, the specification which is to be presented should be regarded as a *pseudo* specification rather than the final artefact. The Eclipse framework currently does no support execution capabilities for QVTr transformations. In order to execute those, external tools are required: ModelMorf[3]; Medini QVT[4]; and Echo[5] are three possibilities available nowadays. Nevertheless, these tools are not stable yet and there's little support available on the specification of transformations using QVTr. Consequently, it was not possible to test the specified transformation, which is the main reason why it should be regarded as a *pseudo* transformation.

The transformation's domains are BDOs and VBDOs. The first relation of the transformation maps `BusinessobjectComplexTypes`, from the BDO metamodel, and `Models`, from the VBDO metamodel. Listing 4.23 displays the relation. Since the element root of every VBDO model is a

3 `http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`
4 `http://projects.ikv.de/qvt`
5 `https://github.com/haslab/echo`

Model element, this rule is necessary to guarantee that element does exist. This rule states that whenever a BusinessobjectComplexType b exists, a Model m must also exist, and its name attribute must correspond to b's package value, and vice versa (lines 3 and 4).

```
1 top relation BusinessObjectModel {
2   p: String;
3   enforce domain bdo b:BusinessobjectComplexType {package = p};
4   enforce domain vbdo m:Model {name = p};
5 }
```

Listing 4.23: BusinessobjetcComplexType relation

The next relation maps BDOs and their respective Entitys. As mentioned when explaining the Java implementation in section 4.2.2, in the VBDO metamodel, each BDO is corresponded by an Entity. Relation BusinessObjectEntity provides this mapping and is presented in Listing 4.24.

```
top relation BusinessObjectEntity {
  n: String;
  enforce domain bdo b:BusinessobjectComplexType {
    name = n
  } {
    (n.toLowerCase().at(0) == 'a' or n.toLowerCase().at(0) == 'e' or
    n.toLowerCase().at(0) == 'i' or n.toLowerCase().at(0) == 'o' or
    n.toLowerCase().at(0) == 'u') and b->element.isEmpty and
    !b->classDef.extends.oclIsUndefined
  };
  enforce domain vbdo e:Entity {
    article = 'An'
    name = n,
    opposite(Model::entities)=m:Model{}
  };
  when {BusinessObjectModel(b,m)}
  where {
    ExtendsI2SHome2DefaultType(b,e) and
    ...
    ExtendsBDO2EntityType(b,e);
  }
}
```

Listing 4.24: BusinessobjetcComplexType relation

The VBDO language provides some "syntatic sugar" to make it more natural for insurance experts. Nevertheless, such option generates a lot of different ways one can specify a BDO. If the BDO's name starts with a vowel and extends another BDO, one is supposed to write, for example, An InsurancePolicy is a Policy. This is precisely the case mapped by the relation in Listing 4.24. However, if the BDO's name does not start with a vowel, the same relation would not fit. For that case, a similar relation would have to be specified. This is precisely one of the problems mentioned when introducing QVTr. The more complex the specification is, the more extensive it tends to get, in comparison with *operational* approaches, because more relations have to be specified.

The relation `BusinessObjectEntity` in Listing 4.24 states that a BDO (line 3) which `name` starts with a vowel (lines 6-8) is corresponded by an `Entity` (line 11), with the same `name` (lines 4 and 13) and with the `article` attribute equal to `An` (line 12), only when the entity's `Model m` and BDO `b` are related by relation `BusinessObjectModel` (line 16). The `where` clause is there to make sure that the extension BDO is set. The first relation included in the `where` clause guarantees the case when the BDO extends the `I2SHome` default BDO (line 18). The `...` is a *pseudo* way of representing the other relations that need to be declared for the other default BDOs. Once again this "non-ending" enumeration is necessary because of QVTr's *mechanics*. The last line of the `where` clause is for the case when the BDO extends other (normal) BDO (line 20). The `ExtendsI2SHome2DefaultType` relation, referenced in line 18 of Listing 4.24, is presented in Listing 4.25.

```
 1 relation ExtendsI2SHome2DefaultType {
 2   domain bdo b:BusinessobjectComplexType {
 3     classDef = cd:ClassDefComplexType {
 4       extends = 'I2SHome'
 5     }
 6   };
 7   domain vbdo e:Entity {
 8     superDefault = sd:DefaultWrapper {
 9       default = dt:DefaultType.I2SHome
10     }
11   };
12 }
```

Listing 4.25: `ExtendsI2SHome2DefaultType` relation

When a BDO extends a default one, the `superDefault` attribute of the `Entity` element is the one to use to state that extension (Listing 4.25 line 8). In order to guarantee the case when a BDO extends another default BDO, besides the `I2SHome` one, like the `GISDataObjImplSer` BDO, another relation similar to the one in Listing 4.25 would have to be specified and then included in the `where` clause of rule `BusinessObjectEntity`. This relation would be exactly the same as the one in Listing 4.25 - the only difference would be on the lines 4 and 9, by replacing `I2SHome` for `GISDataObjImplSer`. If a concept similar to ATL helper functions was available in QVTr this specification would be easier. Helper functions are basically imperative functions that would allow to merge all this relations necessary for each default BDO into a single relation. That single relation would be something like Listing 4.26 presents.

```
 1 relation ExtendsDefaultType {
 2   type : String;
 3   domain bdo b:BusinessobjectComplexType {
 4     classDef = cd:ClassDefComplexType {
 5       extends = type
 6     }
 7   };
 8   domain vbdo e:Entity {
 9     superDefault = sd:DefaultWrapper {
10       default = converType2DefaultBdo(type)
11     }
12   };
13   when { type = 'I2SHome' or ... or type = 'GISDataObjImplSer'}
14 }
```

Listing 4.26: Unified relation reccuring to helpers

In this case, the helper being used would be called `converType2DefaultBdo` (line 10), and it would basically map the strings representing the default BDOs in the `bdo.ecore` into the default BDOs declared in the `owl.ecore`. As a matter of fact, the QVTr standard declares *helpers* are indeed supported, nevertheless, it is not clear weather the QVTr execution tools do also support helpers, which is the reason why they were not used. Besides this, most QVTr transformations do not make use of helpers.

Another important feature to map are the BDO's attributes. BDO's attributes exist as `PropertyTypes` in the BDO metamodel and are mapped by `Fields` in the VBDO metamodel. In order to deal with attributes, firstly, another relation like the one in Listing 4.24 has to be specified so that BDOs carrying attributes are also identified. By copying the relation in Listing 4.24 and then changing line 8, by preceding with the negation sign the expression `b.element.isEmpty`, the relation then deals with BDOs, which `name` starts with a vowel, which extend other BDO and which have some attributes. In that case, the `where` clause also needs to be updated by adding the relations to deal with the mapping of the attributes. Since an attribute type can be a primitive type, a default BDO or a normal BDO, once again, several relations are necessary to be included. The `where` clause is something like Listing 4.27 displays.

```
where {
  ExtendsI2SHome2DefaultType(b,e) and
  ...
  ExtendsBDO2EntityType(b,e) and
  AttributeString2PrimitiveType(b,e) and
  ...
  AttributeI2SHome2DefaultType(b,e) and
  ...
  AttributeBDO2EntityType(b,e) and
}
```

Listing 4.27: `where` clause

Lines 1 to 4 are maintained to take care of the extension BDO. Line 5 is for the attributes which type is `String` and the following `...` is for the other primitive types. Lines 7 and 8 are included for the cases where the attributes' type is a default BDO. Line 9 is for the attributes which type points to normal BDOs. Relations `AttributeString2PrimitiveType` and `AttributeBDO2EntityType` are presented in Listings 4.28 and 4.29, respectively.

```
relation AttributeString2PrimitiveType {
  attrName : String;
  enforce domain bdo b:BusinessobjectComplexType {
    element = attr: BDO::PropertyType {
      name = attrName,
      type = 'String'
    }
  };
  enforce domain vbdo e:Entity {
    fields = f:Field {
      inlineAlias = ia:InlineAlias {
        name = attrName,
        type = ptw:PrimitiveTypeWrapper {
          primitive = PrimitiveType.CharacterSequence
        }
      }
    }
  };
}
```

Listing 4.28: `AttributeString2PrimitiveType` relation

```
relation AttributeBDO2EntityType {
  attrName : String; attrType : String;
  enforce domain bdo b:BusinessobjectComplexType {
    element = attr: BDO::PropertyType {
      name = attrName,
      type = attrType
    }
  };
  enforce domain vbdo e:Entity {
    fields = f:Field {
      inlineAlias = ia:InlineAlias {
        name = attrName,
        entity = et:Entity {
          name = attrType
        }
      }
    }
  };
  when {attrType != 'string' and ... and attrType != 'I2SHome'}
}
```

Listing 4.29: `AttributeBDO2EntityType` relation

In both relations, a `Field` and an `InlineAlias` inside it are instantiated, on the side of the VBDO metamodel. However, in Listing 4.28, the `type` attribute is the one used to hold the primitive type (line 12), whereas in Listing 4.29, the `entity` attribute is the one used to hold the BDO reference (an `Entity`) (line 11). This difference is what makes the need for having two relations and not one. It is important to notice line 17 in Listing 4.29 - the `when` condition guarantees that the primitive types and default BDO are not considered by this rule, because those are supposed to be handled by the corresponding relations, like the one in Listing 4.28. There is a slight difference in this implementation in comparison to the Java one because, in Java, the *Get* implementation deals with all the BDOs references by using their *fully qualified name* if possible, which is found by looking up the imports of the BDO. In this case, the simple name of the BDOs is what is being used to reference them (Listing 4.29 lines 5 and 12). To be more precise, a more complex OCL expression would have to be used.

4.2.4 *Comparisson*

The comparison between the Java and the QVTr implementations was intended to be done in the same basis of the comparison done for the unidirectional transformation. Bearing this in mind, the comparison terms considered were, once again, performance and maintainability.

Taking the performance into account, since the QVTr implementation could not be executed, in this case, it was impossible to establish a comparison entirely based on numbers. Nevertheless, for the Java implementation, some performance tests were planned to have at least a slight notion of how the implementation would act. Since the Java implementation follows the lenses approach, it was important to test both the *Get* and *Put* transformations. To do so, the following tests were executed:

For each BDO b available:

1. Apply the *Get* transformation over b to obtain the VBDO v, and register the time taken;
2. Apply the *Put* transformation over b and v to obtain the BDO $b2$, and register the time taken;
3. Generate an *empty* BDO b';
4. Apply the *Put* transformation over b' and v to obtain the BDO $b3$, and register the time taken;

Point 1 intended to test the performance of the *Get* implementation. Point 2 intended to test the performance of the *Put* implementation. Finally, points 3 and 4, were also aimed at the *Put* implementation, however, they targeted a specific case of execution, where the pre-state BDO given as input is “empty” and has to be completely “constructed” from scratch having the VBDO in the input as reference. These last tests enclosed by points 3 and 4 are related to a specific term known as “batch-mode”.

The results obtained for these tests were not very elucidative. For most of the BDOs, the times registered for every test were always 0 milliseconds, because the Java implementation proved to be very fast for the BDOs used. The only BDO for which the the *Get* and the *Put* tests were a little bit longer was the `Produto.bdo`, which returned 1 millisecond for both tests. This BDO is the biggest one in size.

Shifting focus into the maintainability issue, as mentioned before, the main advantage of the QVTr implementation in comparison to the Java one is the fact that for QVTr only one specification artefact is necessary whereas with Java an implementation for the *Get* and one for the *Put* must be specified. Furthermore, QVTr makes the specification of the transformation completely focused in the relations to be specified while Java, for itself, forces the developer to specify not only the transformation code but also some necessary functionality that QVTr offers "for free", such as the detection of metamodels' specific elements. Nevertheless, in spite of a QVTr transformation being completely specified in a single file, in this case, the file ended up being rather extensive.

The declarational approach followed by QVTr makes complex transformations harder to specify. What sometimes is expressed in the operational approach with a simple `if` block that one can fit wherever in the code, can be much harder to accomplish in QVTr. In this case, the necessity that existed to specify a `relation` for each primitive type or default BDO could be easily avoided if QVTr helpers were used. Helpers were not used because, in spite of QVTr documentation referencing them, it is not clear how they should be used. At the same time, using helpers makes bidirectionalisation unclear to accomplish for the execution tools supporting QVTr, because basically using helpers implies introducing imperative code to what used to be declarative transformations.

Finally, in terms of support, which should be another point to consider in analysing maintainability, QVTr has not reached a satisfiable level yet. In spite of being regarded as a standard for bidirectional transformations it is not yet a de facto standard, because there does not exist tool support for the language. Nevertheless, it is important to mention that, at the same time, none of the other languages with the same purpose of QVTr are de facto standards. This fact resulted in a internal threat to the validity of the QVTr implementation because very few examples of QVTr implementations could be found to guide on the actual implementation of the transformation that had to be specified.

Bearing this in mind, I believe that, for now, QVTr is only suitable for the implementation of "small" transformations, suitable for academic means. When considering "industrial" cases like the transformation which had to be implemented, the best implementation to opt for would be Java. In terms of workload, Java can still be the implementation which is harder to achieve in terms of time dispensed, nevertheless, considering the fact that QVTr lacks the necessary support in terms of learnability and executability and that, in the end, the actual transformation ends up being extremely extensive, Java seems to be best choice in terms of maintainability, even so. In terms of executability, Java proved also proved to be an excellent choice considering the BDOs experimental base provided by the EISC.

5 CONCLUSIONS AND FUTURE WORK

This chapter finalises the dissertation with the conclusions and prospects of future work.

5.1 CONCLUSIONS

This dissertation presented two different implementations for the specification of two model transformations: a unidirectional transformation; and a bidirectional transformation. For each transformation, one of the implementations was achieved by using a specific DSL whereas the other was specified using a GPL.

In the case of the unidirectional transformation, the main conclusion was that one should opt for the DSL implementation rather than the GPL one, in the cases when performance is not so much a critical parameter to take into consideration. DSLs such as ATL, offer a lot of advantages in terms of maintainability, in comparison to GPLs, which eventually makes them a better choice. By using languages such as ATL, the developer only has to focus on the actual transformation rules to specify, instead of also having to deal with other secondary but equally necessary concepts for the transformation to occur. ATL comes attached with a fully implemented structure that guarantees that the developer does not have to worry about aspects such as *rule scheduling*, which determines which rules to be applied first, or *location determination*, which fetches the metamodels' elements over which rules should be applied, etc. In my opinion, the time necessary to implement those auxiliary but also necessary features from scratch in a GPL such as Java should be greater than the time needed for getting acquainted with the way those same features are treated in a specific DSL plataform. This is where DSLs outweigh GPLs when it comes to unidirectional transformations. At the same time, ATL is also very positive when it comes to support, not only because of the great variety of sample code one can find online, but mainly because it is a stable language recognised in the MDE as the de facto standard for unidirectional transformations. Nevertheless, when performance is a critical factor, Java normally out performs ATL, so in that cases one must sacrifice maintainability for execution speed.

When it comes to bidirectional transformations, the conclusions drawn are slightly different. Similarly to what happens with DSLs for unidirectional transformations, in terms of maintainability, theoretically, the DSLs for bidirectional transformations should outweigh GPLs. However, in practise, this is not necessarily true. The first reason for this is the fact that, in spite of the reasonable number of different technologies specifically for implementing bidirectional transformations, such as Boomerang or QVTr, all of them have still not matured to the point of mainstream adoption. This undermines maintainability because little support can be found on those technologies. This was one of the problems encountered when specifying the QVTr implementation.

At the same time, in the particular case of QVTr, one of the reasons for choosing it as the DSL for the specification of the bidirectional transformation was the great advantage that comes with it of specifying in a single document what would typically be done in two documents (forward and backward transformation). This is, in fact, a great advantage for maintainability, because instead of having to specify to separate files to hold the two unidirectional transformations that compose the bidirectional one, everything can be done in a single document in an unified way. Much as that proved to be true during the actual specification of the transformation, I found the specification considerably easier in Java than in QVTr. What would simply be implemented with a single `if` structure, most times would be implemented in a harder and more extensive way with QVTr. Thus, the actual QVTr specification could be done in a single document as one was expecting, but, simultaneously, that file should also be very long, the more complex the transformation should be. Bearing this in mind, I think it is reasonable to say that, for now, the DSL approach towards bidirectional transformations is still not the best choice to opt for even considering maintainability - at least for complex transformations. Bearing this in mind, in this case, the GPL implementation was considered the best option. Furthermore, that same implementation displayed positive results when subjected to the performance tests over the sample BDOs, which accounted for its nomination.

5.2 PROSPECT FOR FUTURE WORK

The work developed in this dissertation suggests several open directions that can be explored in future work. We enunciate some of them in this section.

SYNTETIC TESTS In order to test the implementations for both transformations, the BDOs provided by the insurance company were used as our experimental basis. This is positive because, in most academic scenarios, normally it is difficult to have representative samples to use in order to test the produced work. On the other hand, being given such experimental sample can also have its disadvantages. The main one is the "lack of control" in the sample. If one wanted to test, for instance, the performance of the implementations for a BDO with 1, 2, 4, 8... attributes, probably it would be possible to find examples of BDOs with those characteristics in the given sample. However, if more precise conditions were fixed, probably the sample would not be representative enough to cover those cases. In those situations, the only solution would be to generate *syntetic* BDOs for testing. This could be interesting to test the implementations under more specific scenarios as the ones tested, such as, over BDOs with an increasing number of cross-references between each other.

5.2 PROSPECT FOR FUTURE WORK

MAINTAINABILITY COMPARISON BASED ON CONCRETE METRICS The implementations achieved for each transformations were compared based on their performance and maintainability. The performance was tested by measuring execution times whenever possible. However, for the maintainability guideline, the comparison established was only based on the volume metric. It would be interesting to trying to establish a comparison focused on more concrete metrics, such as the number of lines of code, the number of functions invoked, etc. Such approach should be split into two parts, where in the first one, concrete metrics would be eleced and, in the last one, those metrics would be measured to proceed with the comparison. This would be a more plausible comparison.

TRYING OTHER TECHNOLOGIES The DSLs chosen to implement each one of the transformations were ATL and QVTr. Those were the choices made, essentially, because these languages are (or are on the verge of being) Eclipse recognised standards. Nonetheless, it could also be interesting to implement the same transformations using different technologies, and then trying to compare the obtained results. In particular, it could be interesting to use Boomerang as the language to specify the bidirectional transformation because, contrarily to QVTr, Boomerang is a relatively stable language. At the same time, the implementation paradigm would be different because Boomerang is oriented to the concrete syntax (text transformations) while QVTr allows the specification of model transformations.

BIBLIOGRAPHY

Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. *SIGPLAN Not.*, 43(1):407–419, January 2008. ISSN 0362-1340. doi: 10.1145/1328897.1328487. URL http://doi.acm.org/10.1145/1328897.1328487.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for xml languages. *Inf. Syst.*, 33(4-5):385–406, June 2008. ISSN 0306-4379. doi: 10.1016/j.is.2008.01.006. URL http://dx.doi.org/10.1016/j.is.2008.01.006.

M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. G - Reference,Information and Interdisciplinary Subjects Series. Morgan & Claypool, 2012. ISBN 9781608458820. URL http://books.google.pt/books?id=2tVu-wC4XkAC.

Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Jtl: A bidirectional and change propagating transformation language. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 183–202, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19439-9. URL http://dl.acm.org/citation.cfm?id=1964571.1964586.

K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006. ISSN 0018-8670. doi: 10.1147/sj.453.0621. URL http://dx.doi.org/10.1147/sj.453.0621.

Eclipse. Xtend User Guide. Technical report, Eclipse, September 2013a.

Eclipse. Xtext 2.4.3 Documentation. Technical report, Eclipse, September 2013b.

J. Nathan Foster and Benjamin C. Pierce. Boomerang Programmer's Manual. Technical report, SEAS, September 2009.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007. ISSN 0164-0925. doi: 10.1145/1232420.1232424. URL http://doi.acm.org/10.1145/1232420.1232424.

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. *SIGPLAN Not.*, 43(9):383–396, September 2008. ISSN 0362-1340. doi: 10.1145/1411203.1411257. URL http://doi.acm.org/10.1145/1411203.1411257.

D.S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. OMG. Wiley, 2003. ISBN 9780471462279. URL http://books.google.pt/books?id=EUsyPEsszrYC.

Aniruddha Gokhale, Douglas C. Schmidt, Balachandran Natarajan, and Nanbor Wang. Applying model-integrated computing to component middleware and enterprise applications. *Commun. ACM*, 45(10):65–70, October 2002. ISSN 0001-0782. doi: 10.1145/570907.570933. URL http://doi.acm.org/10.1145/570907.570933.

J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Timely, practical, reliable. Wiley, 2004. ISBN 9780471202844. URL http://books.google.pt/books?id=06dQAAAAMAAJ.

Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12-14, 2007, Proceedings*, pages 30–39, 2007. doi: 10.1109/QUATIC.2007.8. URL http://dx.doi.org/10.1109/QUATIC.2007.8.

D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012. ISBN 9780262017152. URL http://books.google.pt/books?id=DDv8Ie_jBUQC.

Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, pages 128–138, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-31780-5, 978-3-540-31780-7. doi: 10.1007/11663430_14. URL http://dx.doi.org/10.1007/11663430_14.

Shinya Kawanaka and Haruo Hosoya. bixid: A bidirectional transformation language for xml. *SIGPLAN Not.*, 41(9):201–214, September 2006. ISSN 0362-1340. doi: 10.1145/1160074.1159830. URL http://doi.acm.org/10.1145/1160074.1159830.

Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.10.021. URL http://dx.doi.org/10.1016/j.entcs.2005.10.021.

Object Management Group. Object Constraint Language (Version 2.4). Technical report, OMG, a.

Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure. Technical report, OMG, b.

Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Technical report, OMG, January 2011.

Hugo Pacheco. *Bidirectional Data Transformation by Calculation*. PhD thesis, Universidade do Minho - Escola de Engenharia, 2012.

I. Sommerville. *Software Engineering*. International computer science series. Addison-Wesley, 2007. ISBN 9780321313799. URL http://books.google.pt/books?id=B7idKfL0H64C.

J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, Apr 1997. ISSN 0018-9162. doi: 10.1109/2.585163.

Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*, pages 446–453, 2003. doi: 10.1007/978-3-540-25959-6_35. URL http://dx.doi.org/10.1007/978-3-540-25959-6_35.

M. Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013. ISBN 9781481218580. URL http://books.google.pt/books?id=J2i0lwEACAAJ.

World Wide Web Consortium. OWL Web Ontology Language Reference. Technical report, W3C, February 2004.

Mahdi Yusuf. Programming languages popularity. http://www.mahdiyusuf.com/post/3279728032/solved-what-programming-languages-should-i-learn.

Part III

APPENDICES

.1 BUSINESSOBJECT.XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="unqualified">
  <xsd:element name="businessobject"
    type="businessobjectComplexType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        businessobject e a estrutura de dados XML que server
        como source para os geradores de BDOs
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>

  <xsd:complexType name="IgnorableType">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element name="ignoreElement" minOccurs="0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">

            <xsd:whiteSpace value="collapse"></xsd:whiteSpace>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="ignore" minOccurs="0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:whiteSpace value="collapse"></xsd:whiteSpace>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>

    </xsd:choice>
  </xsd:complexType>

  <xsd:group name="ignorablesGroup">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="inconsistenteignorables"
        type="IgnorableType">
      </xsd:element>
      <xsd:element minOccurs="0" name="alteracaoignorables">
        <xsd:complexType>
          <xsd:complexContent>
            <xsd:extension base="IgnorableType">
```

```
              <xsd:attribute use="optional" name="equalsInconsistenteIgnorables"
                type="xsd:boolean" default="false">
              </xsd:attribute>
            </xsd:extension>
          </xsd:complexContent>
        </xsd:complexType>
      </xsd:element>
      <xsd:element minOccurs="0" name="loadallignorables" type="IgnorableType">
      </xsd:element>
    </xsd:sequence>
  </xsd:group>


  <xsd:complexType name="propertyType">
    <xsd:sequence>
      <xsd:element name="calculationdependency" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
            ****************************** NAO IMPLEMENTADO
                ******************************
]]>
          </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="depends" type="dependType"
              maxOccurs="unbounded" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="validationdependency" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
            ****************************** NAO IMPLEMENTADO
                ******************************
]]>
          </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="depends" type="dependType"
              maxOccurs="unbounded" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

```
<xsd:element name="internalSet1" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>
      <![CDATA[ Caso exista este elemento, sera
              gerado um abstract extra, que sera evocado no
              inicio do metodo Set do elemento, antes de ter
              sido feita qualquer outra operacao. ]]>
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="throw" type="xsd:string"
        minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="internalSet2" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>
      <![CDATA[ Caso exista este elemento, sera
              gerado um abstract extra, que sera evocado no
              metodo Set do elemento, depois de ter sido
              evocado o fireVetoableChange e ter sido feito o
              SetAtribute do elemento, mas antes do
              firePropertyChange ]]>
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="throw" type="xsd:string"
        minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="internalSet3" minOccurs="0">
  <xsd:annotation>
    <xsd:documentation>
      <![CDATA[ Caso exista este elemento, sera
              gerado um abstract extra, que sera evocado no
              fim metodo Set do elemento. ]]>
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="throw" type="xsd:string"
        minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
```

```
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="throw" type="xsd:string" minOccurs="0"
        maxOccurs="unbounded" />
      <xsd:element name="setcomment" type="xsd:string"
        default="@TODO DOCUMENT ME" minOccurs="0" />
      <xsd:element name="getcomment" type="xsd:string"
        default="@TODO DOCUMENT ME" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required">
      <xsd:annotation>
        <xsd:documentation>
          Nome que sera atribuido ao elemento da classe.
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="type" type="xsd:string" use="required">
      <xsd:annotation>
        <xsd:documentation>
          Tipo de dados do elemento (Ex. int, String, char,
          ...)
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="read" type="propertyAccessType"
      use="optional" default="true">
      <xsd:annotation>
        <xsd:documentation>
          Tipo de acesso de leitura que tera o elemento.
          Opcoes:

          - true: Serao gerados metodos get publicos na
          interface de negocio. E implementados na classe
          BaseImpl

          - false: Nao existirao metodos publicos de acesso ao
          elemento.

          - internal: Serao gerados metodos get publicos na
          interface interna.E implementados na classe BaseImpl

          - abstract: Serao declarados na interface de negocio
          metodos de acesso abstractos que nao serao
          implementados na classe de BaseImpl
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
```

```
  <xsd:attribute name="write" type="propertyAccessType"
    use="optional" default="true">
    <xsd:annotation>
      <xsd:documentation>
        Tipo de acesso de escrita que tera o elemento.
        Opcoes:

        - true: Serao gerados metodos set publicos na
        interface de negocio. E implementados na classe
        BaseImpl

        - false: Nao existirao metodos publicos de acesso ao
        elemento.

        - internal: Serao gerados metodos set publicos na
        interface interna.E implementados na classe BaseImpl

        - abstract: Serao declarados na interface de negocio
        metodos de acesso abstractos que nao serao
        implementados na classe de BaseImpl
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="attributeName" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        ****************************** NAO IMPLEMENTADO
        ******************************]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="propertyname" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        ****************************** NAO IMPLEMENTADO
        ******************************]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="defaultValue" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
```

```
          <![CDATA[
          ****************************** NAO IMPLEMENTADO
          ******************************]]>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="abstract" type="xsd:boolean"
      use="optional">
      <xsd:annotation>
        <xsd:documentation>
          Flag que define se os metodos de acesso serao ou nao
          declarados como abstractos. NOTA: Esta flag quando
          usada, faz override aos atributos read e
          write.Opcoes: true/false
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="final" type="xsd:boolean" use="optional"
      default="true">
      <xsd:annotation>
        <xsd:documentation>
          Flag que define se o metodo sera ou nao declarado
          como final. Opcoes: true/false
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="extramethodType">
    <xsd:sequence>
      <xsd:element name="comment" type="xsd:string"
        default="@TODO DOCUMENT ME" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
(Opcional) Comentario auxiliar, para acompanhar o codigo extra.
]]>
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="extra">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
Codigo extra, ter em atencao que devera ser escrito como codigo Java.
]]>
          </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="dependType">
  <xsd:attribute name="propname" type="xsd:string" use="required" />
  <xsd:attribute name="proppath" type="xsd:string" use="optional" />
</xsd:complexType>
<xsd:attributeGroup name="classAttributeType">
  <xsd:attribute name="extends" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        (Opcional) Interface que a Interface gerada extende.
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="implements" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        (Opcional) Interfaces que a Interface gerada
        implementa. Nota: Se for mais do que uma, devem ser
        separadas por virgulas.
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
</xsd:attributeGroup>
<xsd:attributeGroup name="constructorParmsType">
  <xsd:attribute name="tipo" type="xsd:string" use="required">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        Tipo de dados do paramentro de entrada.
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="name" type="xsd:string" use="required">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        Nome do parametro de entrada
      ]]>
```

```
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="element" type="xsd:string"
      use="optional">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
          Define a variavel da classe a qual este constructor
          deve attribuir para segurar o resulado.
        ]]>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="cfgrequired" type="xsd:boolean"
      use="optional" default="true">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
          Define se e ou nao necessario o carregamento de
          configuracoes extra, desencadeia tambem a geracao de
          estrategias sobre BDO, nomeadamente estrategias de
          calculo, estrategias de objecto alterado e
          estrategias de validacao.
        ]]>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
  </xsd:attributeGroup>
  <xsd:simpleType name="propertyAccessType">

    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="true" />
      <xsd:enumeration value="false" />
      <xsd:enumeration value="abstract" />
      <xsd:enumeration value="internal" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="classDefComplexType">
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded"
        name="import">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
            Imports especificos para a Classe
          ]]>
          </xsd:documentation>
        </xsd:annotation>
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:whiteSpace value="collapse"></xsd:whiteSpace>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element minOccurs="0" name="comment"
        type="xsd:string">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
            Comentario que aparecera antes da declaracao da
            Classe
          ]]>
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element minOccurs="0" maxOccurs="unbounded"
        name="constructor">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
            Definicao de um construtor para a classe
          ]]>
          </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element minOccurs="0" name="comment"
              type="xsd:string">
              <xsd:annotation>
                <xsd:documentation>
                  <![CDATA[
                  Comentario que aparecera antes da
                  declaracao do construtor da Classe
                ]]>
                </xsd:documentation>
              </xsd:annotation>
            </xsd:element>
            <xsd:element minOccurs="0" maxOccurs="unbounded"
              name="parm">
              <xsd:annotation>
                <xsd:documentation>
                  <![CDATA[
                  Parametro de entrada para o metodo
                construtor, definido atravez de
                atributos.
              ]]>
              </xsd:documentation>
            </xsd:annotation>
            <xsd:complexType>
              <xsd:attributeGroup
                ref="constructorParmsType">
              </xsd:attributeGroup>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:group ref="ignorablesGroup"></xsd:group>
    <xsd:element minOccurs="0" maxOccurs="1"
      name="beaneventsupport">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
                Elemento para definir se deve ou nao ser gerado
                codigo extra para suporte a BeansEvent, para
                isso deve ser alterado o atributo type, para
                true.
              ]]>
        </xsd:documentation>
      </xsd:annotation>
      <xsd:complexType>
        <xsd:attribute use="optional" default="provided"
          name="type">
          <xsd:annotation>
            <xsd:documentation>
              <![CDATA[
                        Quando true faz com que seja
                        implementado codigo extra para suporte a
                        BeanEvent.
                      ]]>
            </xsd:documentation>
          </xsd:annotation>
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:whiteSpace value="collapse"></xsd:whiteSpace>
              <xsd:enumeration value="provided"></xsd:enumeration>
              <xsd:enumeration value="true"></xsd:enumeration>
              <xsd:enumeration value="false"></xsd:enumeration>
            </xsd:restriction>
          </xsd:simpleType>
          </xsd:attribute>
        </xsd:complexType>
      </xsd:element>
      <xsd:element minOccurs="0" maxOccurs="1"
        name="logsupport">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
                    Elemento para definir se deve ou nao ser gerado
                    codigo para logs org.apache.commons.logging.Log,
                    para isso deve ser alterado o atributo type para
                    true.
                  ]]>
          </xsd:documentation>
        </xsd:annotation>
        <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="xsd:string">
              <xsd:attribute use="optional"
                default="provided" name="type">
                <xsd:annotation>
                  <xsd:documentation>
                    <![CDATA[
                                    Define se deve ou nao ser gerado
                                    codigo para logs
                                    org.apache.commons.logging.Log,
                                    para isso deve ser alterado o
                                    atributo type para true.
                                  ]]>
                  </xsd:documentation>
                </xsd:annotation>
                <xsd:simpleType>
                  <xsd:restriction
                    base="xsd:string">
                    <xsd:enumeration value="true"></xsd:enumeration>
                    <xsd:enumeration
                      value="false">
                    </xsd:enumeration>
                    <xsd:enumeration
                      value="provided">
                    </xsd:enumeration>
                    <xsd:whiteSpace
                      value="collapse">
                    </xsd:whiteSpace>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:attribute>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attributeGroup ref="classAttributeType"></xsd:attributeGroup>
  <xsd:attribute name="userdefinedclassname" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        Nome da class de implementacao do BDO que sera
        implementado pelo utilizador. Se nao for definido
        ser calculado como {nome do BDO}Impl.
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="interfacename" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        Nome do interface do BDO. Se nao for definido ser
        calculado como {nome do BDO}.
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="internalinterfacename" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        Nome da interface Interno do BDO. Se nao for
        definido ser calculado como {nome do BDO}Interno.
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="package" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        Package para a Interface, caso seja diferente do
        default declarado no RootElement.
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="serialversionUID" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        Serial Version UID to be included if existes else
        classe is generated without serialVersionUID.
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
</xsd:complexType>
<xsd:complexType name="interfaceInternoDefComplexType">
  <xsd:sequence>
    <xsd:element minOccurs="0" maxOccurs="unbounded"
      name="import">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
          Imports especificos para Interface
        ]]>
        </xsd:documentation>
      </xsd:annotation>
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:whiteSpace value="collapse"></xsd:whiteSpace>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element minOccurs="0" name="comment"
      type="xsd:string">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
          Comentario que aparecera antes da declaracao da
          Interface Interna
        ]]>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
  <xsd:attributeGroup ref="classAttributeType"></xsd:attributeGroup>
  <xsd:attribute name="package" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        (Opcional) Package for internal interface in case it
        is diferente from the default.
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
</xsd:complexType>
<xsd:complexType name="interfaceDefComplexType">
  <xsd:sequence>
    <xsd:element minOccurs="0" maxOccurs="unbounded"
      name="import">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
          Imports especificos para Interface
        ]]>
        </xsd:documentation>
      </xsd:annotation>
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:whiteSpace value="collapse"></xsd:whiteSpace>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element minOccurs="0" name="comment"
      type="xsd:string">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
          Comentario que aparecera antes da declaracao da
          Interface de negocio
        ]]>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
  </xsd:sequence>
  <xsd:attributeGroup ref="classAttributeType"></xsd:attributeGroup>
  <xsd:attribute name="package" type="xsd:string"
    use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[
        Package para a Interface, caso seja diferente do
        default declarado no RootElement.
      ]]>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
</xsd:complexType>
<xsd:complexType name="businessobjectComplexType">
  <xsd:sequence>
    <xsd:element name="package" type="xsd:string">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
          Default package para as classes que serao
          geradas.
        ]]>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
    <xsd:element minOccurs="0" maxOccurs="unbounded"
      name="import">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
          Imports extra que serao adicionados a todas as
          classes geradas. Nota: Os imports colocados
          aqui, serao adicionados a Interface de Negocio,
          Interface Interna e a classe da Implementacao
          Base
        ]]>
        </xsd:documentation>
      </xsd:annotation>
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:whiteSpace value="collapse"></xsd:whiteSpace>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="interfaceDef"
      type="interfaceDefComplexType">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
          Definicao da Interface de negocio
        ]]>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:element>
```

```
      <xsd:element name="interfaceInternoDef"
        type="interfaceInternoDefComplexType">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
            Definicao da Interface Interna
          ]]>
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="classDef" type="classDefComplexType">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
            Definicao da Classe da implementacao Base.
          ]]>
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element minOccurs="0" maxOccurs="1" name="keys" type="keysComplexType"
          >
      </xsd:element>
      <xsd:element minOccurs="0" maxOccurs="unbounded"
        name="element" type="propertyType">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
                    Definicao de um elemento do BDO.
                  ]]>
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:element minOccurs="0" maxOccurs="unbounded"
        name="extramethod" type="extramethodType">
        <xsd:annotation>
          <xsd:documentation>
            <![CDATA[
Elemento para definir codigo extra que sera gerado na Interface de Negocio.
]]>
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute use="required" name="name" type="xsd:string">
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[
          Nome do BDO, sera usado tambem como nome das classes
          geradas
        ]]>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="keysComplexType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="keyfieldelement">
        <xsd:complexType>
          <xsd:attribute use="required" name="name" type="xsd:string">
            <xsd:annotation>
              <xsd:documentation>
                Nome do elemento que sera a chave do
                BDO, na colecao.
                          </xsd:documentation>
            </xsd:annotation>
          </xsd:attribute>
          <xsd:attribute use="required" name="type" type="xsd:string">
            <xsd:annotation>
              <xsd:documentation>
                Tipo de dados do elemento.
                          </xsd:documentation>
            </xsd:annotation>
          </xsd:attribute>
          <xsd:attribute use="optional" default=".toString()"
            name="convertToString" type="xsd:string">
          </xsd:attribute>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="NewType">
    <xsd:restriction base="xsd:string"></xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

.2 OWL.XSD

.2 OWL.XSD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:owl="http://www.w3
    .org/2002/07/owl#"
  targetNamespace="http://www.w3.org/2002/07/owl#" elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xsd:import namespace="http://www.w3.org/XML/1998/namespace"
    schemaLocation="http://www.w3.org/2001/xml.xsd"/>

  <!-- The ontology -->

  <xsd:complexType name="Prefix">
    <xsd:attribute name="name" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="&PN_PREFIX;|"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="IRI" type="xsd:anyURI" use="required"/>
  </xsd:complexType>
  <xsd:element name="Prefix" type="owl:Prefix"/>

  <xsd:complexType name="Import">
    <xsd:simpleContent>
      <xsd:extension base="xsd:anyURI">
        <xsd:attributeGroup ref="xml:specialAttrs"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:element name="Import" type="owl:Import"/>

  <xsd:complexType name="Ontology">
    <xsd:sequence>
      <xsd:element ref="owl:Prefix" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="owl:Import" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:group ref="owl:ontologyAnnotations"/>
      <xsd:group ref="owl:Axiom" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="ontologyIRI" type="xsd:anyURI" use="optional"/>
    <xsd:attribute name="versionIRI" type="xsd:anyURI" use="optional"/>
    <xsd:attributeGroup ref="xml:specialAttrs"/>
  </xsd:complexType>
  <xsd:element name="Ontology" type="owl:Ontology">
    <xsd:unique name="prefix">
      <xsd:selector xpath="owl:Prefix"/>
      <xsd:field xpath="@name"/>
    </xsd:unique>
```

```
</xsd:element>

<!-- Entities, anonymous individuals, and literals -->

<!-- Note that the "Entity" group does not have a corresponding abstract type.
     This is due to the fact that XML Schema does not support multiple
         inheritence.
     "owl:Class" is both an entity and a class expression. The authors of this
         schema
     determined it was more useful to be able to retrieve "owl:Class" in such
         queries
     as schema(*, owl:ClassExpression).
      -->
<xsd:group name="Entity">
  <xsd:choice>
    <xsd:element ref="owl:Class"/>
    <xsd:element ref="owl:Datatype"/>
    <xsd:element ref="owl:ObjectProperty"/>
    <xsd:element ref="owl:DataProperty"/>
    <xsd:element ref="owl:AnnotationProperty"/>
    <xsd:element ref="owl:NamedIndividual"/>
  </xsd:choice>
</xsd:group>

<!-- This is the type for the attribute. The complex type for the element is
    capitalized. -->
<xsd:simpleType name="abbreviatedIRI">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="&PrefixedName;"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="Class">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:attribute name="IRI" type="xsd:anyURI" use="optional"/>
      <xsd:attribute name="abbreviatedIRI" type="owl:abbreviatedIRI" use="
          optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="Class" type="owl:Class"/>

<xsd:complexType name="Datatype">
  <xsd:complexContent>
    <xsd:extension base="owl:DataRange">
      <xsd:attribute name="IRI" type="xsd:anyURI" use="optional"/>
      <xsd:attribute name="abbreviatedIRI" type="owl:abbreviatedIRI" use="
         optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="Datatype" type="owl:Datatype"/>

<xsd:complexType name="ObjectProperty">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyExpression">
      <xsd:attribute name="IRI" type="xsd:anyURI" use="optional"/>
      <xsd:attribute name="abbreviatedIRI" type="owl:abbreviatedIRI" use="
         optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectProperty" type="owl:ObjectProperty"/>

<xsd:complexType name="DataProperty">
  <xsd:complexContent>
    <xsd:extension base="owl:DataPropertyExpression">
      <xsd:attribute name="IRI" type="xsd:anyURI" use="optional"/>
      <xsd:attribute name="abbreviatedIRI" type="owl:abbreviatedIRI" use="
         optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DataProperty" type="owl:DataProperty"/>

<xsd:complexType name="AnnotationProperty">
  <xsd:attribute name="IRI" type="xsd:anyURI" use="optional"/>
  <xsd:attribute name="abbreviatedIRI" type="owl:abbreviatedIRI" use="optional"
     />
  <xsd:attributeGroup ref="xml:specialAttrs"/>
</xsd:complexType>
<xsd:element name="AnnotationProperty" type="owl:AnnotationProperty"/>

<xsd:complexType name="Individual" abstract="true">
  <xsd:attributeGroup ref="xml:specialAttrs"/>
</xsd:complexType>
<xsd:group name="Individual">
  <xsd:choice>
    <xsd:element ref="owl:NamedIndividual"/>
    <xsd:element ref="owl:AnonymousIndividual"/>
  </xsd:choice>
</xsd:group>
```

```
<xsd:complexType name="NamedIndividual">
  <xsd:complexContent>
    <xsd:extension base="owl:Individual">
      <xsd:attribute name="IRI" type="xsd:anyURI" use="optional"/>
      <xsd:attribute name="abbreviatedIRI" type="owl:abbreviatedIRI" use="
          optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="NamedIndividual" type="owl:NamedIndividual"/>

<xsd:complexType name="AnonymousIndividual">
  <xsd:complexContent>
    <xsd:extension base="owl:Individual">
      <xsd:attribute name="nodeID" type="xsd:NCName" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="AnonymousIndividual" type="owl:AnonymousIndividual"/>

<xsd:complexType name="Literal">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="datatypeIRI" type="xsd:anyURI"/>
      <xsd:attributeGroup ref="xml:specialAttrs"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<xsd:element name="Literal" type="owl:Literal"/>

<!-- Declarations -->

<xsd:complexType name="Declaration">
  <xsd:complexContent>
    <xsd:extension base="owl:Axiom">
      <xsd:sequence>
        <xsd:group ref="owl:Entity"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="Declaration" type="owl:Declaration"/>

<!-- Object property expressions -->

<xsd:complexType name="ObjectPropertyExpression" abstract="true">
  <xsd:attributeGroup ref="xml:specialAttrs"/>
</xsd:complexType>
<xsd:group name="ObjectPropertyExpression">
  <xsd:choice>
    <xsd:element ref="owl:ObjectProperty"/>
    <xsd:element ref="owl:ObjectInverseOf"/>
  </xsd:choice>
</xsd:group>

<xsd:complexType name="ObjectInverseOf">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyExpression">
      <xsd:sequence>
        <xsd:element ref="owl:ObjectProperty"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectInverseOf" type="owl:ObjectInverseOf"/>

<!-- Data property expressions -->

<xsd:complexType name="DataPropertyExpression" abstract="true">
  <xsd:attributeGroup ref="xml:specialAttrs"/>
</xsd:complexType>
<xsd:group name="DataPropertyExpression">
  <xsd:sequence>
    <xsd:element ref="owl:DataProperty"/>
  </xsd:sequence>
</xsd:group>

<!-- Data ranges -->

<xsd:complexType name="DataRange" abstract="true">
  <xsd:attributeGroup ref="xml:specialAttrs"/>
</xsd:complexType>
<xsd:group name="DataRange">
  <xsd:choice>
    <xsd:element ref="owl:Datatype"/>
    <xsd:element ref="owl:DataIntersectionOf"/>
    <xsd:element ref="owl:DataUnionOf"/>
    <xsd:element ref="owl:DataComplementOf"/>
    <xsd:element ref="owl:DataOneOf"/>
    <xsd:element ref="owl:DatatypeRestriction"/>
  </xsd:choice>
</xsd:group>

<xsd:complexType name="DataIntersectionOf">
```

```
    <xsd:complexContent>
      <xsd:extension base="owl:DataRange">
        <xsd:sequence>
          <xsd:group ref="owl:DataRange" minOccurs="2" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="DataIntersectionOf" type="owl:DataIntersectionOf"/>

  <xsd:complexType name="DataUnionOf">
    <xsd:complexContent>
      <xsd:extension base="owl:DataRange">
        <xsd:sequence>
          <xsd:group ref="owl:DataRange" minOccurs="2" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="DataUnionOf" type="owl:DataUnionOf"/>

  <xsd:complexType name="DataComplementOf">
    <xsd:complexContent>
      <xsd:extension base="owl:DataRange">
        <xsd:sequence>
          <xsd:group ref="owl:DataRange"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="DataComplementOf" type="owl:DataComplementOf"/>

  <xsd:complexType name="DataOneOf">
    <xsd:complexContent>
      <xsd:extension base="owl:DataRange">
        <xsd:sequence>
          <xsd:element ref="owl:Literal" minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="DataOneOf" type="owl:DataOneOf"/>

  <xsd:complexType name="DatatypeRestriction">
    <xsd:complexContent>
      <xsd:extension base="owl:DataRange">
        <xsd:sequence>
          <xsd:element ref="owl:Datatype"/>
          <xsd:element name="FacetRestriction" type="owl:FacetRestriction"
             minOccurs="1"
            maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="DatatypeRestriction" type="owl:DatatypeRestriction"/>

  <xsd:complexType name="FacetRestriction">
    <xsd:sequence>
      <xsd:element ref="owl:Literal"/>
    </xsd:sequence>
    <xsd:attribute name="facet" type="xsd:anyURI" use="required"/>
    <xsd:attributeGroup ref="xml:specialAttrs"/>
  </xsd:complexType>

  <!-- Class expressions -->

  <xsd:complexType name="ClassExpression" abstract="true">
    <xsd:attributeGroup ref="xml:specialAttrs"/>
  </xsd:complexType>
  <xsd:group name="ClassExpression">
    <xsd:choice>
      <xsd:element ref="owl:Class"/>
      <xsd:element ref="owl:ObjectIntersectionOf"/>
      <xsd:element ref="owl:ObjectUnionOf"/>
      <xsd:element ref="owl:ObjectComplementOf"/>
      <xsd:element ref="owl:ObjectOneOf"/>
      <xsd:element ref="owl:ObjectSomeValuesFrom"/>
      <xsd:element ref="owl:ObjectAllValuesFrom"/>
      <xsd:element ref="owl:ObjectHasValue"/>
      <xsd:element ref="owl:ObjectHasSelf"/>
      <xsd:element ref="owl:ObjectMinCardinality"/>
      <xsd:element ref="owl:ObjectMaxCardinality"/>
      <xsd:element ref="owl:ObjectExactCardinality"/>
      <xsd:element ref="owl:DataSomeValuesFrom"/>
      <xsd:element ref="owl:DataAllValuesFrom"/>
      <xsd:element ref="owl:DataHasValue"/>
      <xsd:element ref="owl:DataMinCardinality"/>
      <xsd:element ref="owl:DataMaxCardinality"/>
      <xsd:element ref="owl:DataExactCardinality"/>
    </xsd:choice>
  </xsd:group>

  <xsd:complexType name="ObjectIntersectionOf">
```

```
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:ClassExpression" minOccurs="2" maxOccurs="unbounded
            "/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectIntersectionOf" type="owl:ObjectIntersectionOf"/>

<xsd:complexType name="ObjectUnionOf">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:ClassExpression" minOccurs="2" maxOccurs="unbounded
            "/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectUnionOf" type="owl:ObjectUnionOf"/>

<xsd:complexType name="ObjectComplementOf">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:ClassExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectComplementOf" type="owl:ObjectComplementOf"/>

<xsd:complexType name="ObjectOneOf">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:Individual" minOccurs="1" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectOneOf" type="owl:ObjectOneOf"/>

<xsd:complexType name="ObjectSomeValuesFrom">
  <xsd:complexContent>
```

```
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <xsd:group ref="owl:ClassExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectSomeValuesFrom" type="owl:ObjectSomeValuesFrom"/>

<xsd:complexType name="ObjectAllValuesFrom">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <xsd:group ref="owl:ClassExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectAllValuesFrom" type="owl:ObjectAllValuesFrom"/>

<xsd:complexType name="ObjectHasValue">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <xsd:group ref="owl:Individual"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectHasValue" type="owl:ObjectHasValue"/>

<xsd:complexType name="ObjectHasSelf">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectHasSelf" type="owl:ObjectHasSelf"/>

<xsd:complexType name="ObjectMinCardinality">
  <xsd:complexContent>
```

```
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <xsd:group ref="owl:ClassExpression" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="cardinality" type="xsd:nonNegativeInteger" use="
          required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectMinCardinality" type="owl:ObjectMinCardinality"/>

<xsd:complexType name="ObjectMaxCardinality">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <xsd:group ref="owl:ClassExpression" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="cardinality" type="xsd:nonNegativeInteger" use="
          required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectMaxCardinality" type="owl:ObjectMaxCardinality"/>

<xsd:complexType name="ObjectExactCardinality">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <xsd:group ref="owl:ClassExpression" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="cardinality" type="xsd:nonNegativeInteger" use="
          required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectExactCardinality" type="owl:ObjectExactCardinality"/>

<xsd:complexType name="DataSomeValuesFrom">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression" minOccurs="1" maxOccurs="
            unbounded"/>
        <xsd:group ref="owl:DataRange"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DataSomeValuesFrom" type="owl:DataSomeValuesFrom"/>

<xsd:complexType name="DataAllValuesFrom">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression" minOccurs="1" maxOccurs="
            unbounded"/>
        <xsd:group ref="owl:DataRange"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DataAllValuesFrom" type="owl:DataAllValuesFrom"/>

<xsd:complexType name="DataHasValue">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression"/>
        <xsd:element ref="owl:Literal"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DataHasValue" type="owl:DataHasValue"/>

<xsd:complexType name="DataMinCardinality">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression"/>
        <xsd:group ref="owl:DataRange" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="cardinality" type="xsd:nonNegativeInteger" use="
          required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DataMinCardinality" type="owl:DataMinCardinality"/>

<xsd:complexType name="DataMaxCardinality">
  <xsd:complexContent>
```

```
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression"/>
        <xsd:group ref="owl:DataRange" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="cardinality" type="xsd:nonNegativeInteger" use="
          required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DataMaxCardinality" type="owl:DataMaxCardinality"/>

<xsd:complexType name="DataExactCardinality">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassExpression">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression"/>
        <xsd:group ref="owl:DataRange" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="cardinality" type="xsd:nonNegativeInteger" use="
          required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DataExactCardinality" type="owl:DataExactCardinality"/>

<!-- Axioms -->

<xsd:complexType name="Axiom" abstract="true">
  <xsd:sequence>
    <xsd:group ref="owl:axiomAnnotations"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="xml:specialAttrs"/>
</xsd:complexType>
<xsd:group name="Axiom">
  <xsd:choice>
    <xsd:element ref="owl:Declaration"/>
    <xsd:group ref="owl:ClassAxiom"/>
    <xsd:group ref="owl:ObjectPropertyAxiom"/>
    <xsd:group ref="owl:DataPropertyAxiom"/>
    <xsd:element ref="owl:DatatypeDefinition"/>
    <xsd:element ref="owl:HasKey"/>
    <xsd:group ref="owl:Assertion"/>
    <xsd:group ref="owl:AnnotationAxiom"/>
  </xsd:choice>
</xsd:group>
```

```
<!-- Class expression axioms -->

<xsd:complexType name="ClassAxiom" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="owl:Axiom"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:group name="ClassAxiom">
  <xsd:choice>
    <xsd:element ref="owl:SubClassOf"/>
    <xsd:element ref="owl:EquivalentClasses"/>
    <xsd:element ref="owl:DisjointClasses"/>
    <xsd:element ref="owl:DisjointUnion"/>
  </xsd:choice>
</xsd:group>

<xsd:complexType name="SubClassOf">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ClassExpression"/>
        <!-- This is the subexpression -->
        <xsd:group ref="owl:ClassExpression"/>
        <!-- This is the superexpression -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SubClassOf" type="owl:SubClassOf"/>

<xsd:complexType name="EquivalentClasses">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ClassExpression" minOccurs="2" maxOccurs="unbounded
            "/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="EquivalentClasses" type="owl:EquivalentClasses"/>

<xsd:complexType name="DisjointClasses">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassAxiom">
      <xsd:sequence>
```

```
        <xsd:group ref="owl:ClassExpression" minOccurs="2" maxOccurs="unbounded
            "/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DisjointClasses" type="owl:DisjointClasses"/>

<xsd:complexType name="DisjointUnion">
  <xsd:complexContent>
    <xsd:extension base="owl:ClassAxiom">
      <xsd:sequence>
        <xsd:element ref="owl:Class"/>
        <xsd:group ref="owl:ClassExpression" minOccurs="2" maxOccurs="unbounded
            "/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DisjointUnion" type="owl:DisjointUnion"/>

<!-- Object property axioms -->

<xsd:complexType name="ObjectPropertyAxiom" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="owl:Axiom"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:group name="ObjectPropertyAxiom">
  <xsd:choice>
    <xsd:element ref="owl:SubObjectPropertyOf"/>
    <xsd:element ref="owl:EquivalentObjectProperties"/>
    <xsd:element ref="owl:DisjointObjectProperties"/>
    <xsd:element ref="owl:InverseObjectProperties"/>
    <xsd:element ref="owl:ObjectPropertyDomain"/>
    <xsd:element ref="owl:ObjectPropertyRange"/>
    <xsd:element ref="owl:FunctionalObjectProperty"/>
    <xsd:element ref="owl:InverseFunctionalObjectProperty"/>
    <xsd:element ref="owl:ReflexiveObjectProperty"/>
    <xsd:element ref="owl:IrreflexiveObjectProperty"/>
    <xsd:element ref="owl:SymmetricObjectProperty"/>
    <xsd:element ref="owl:AsymmetricObjectProperty"/>
    <xsd:element ref="owl:TransitiveObjectProperty"/>
  </xsd:choice>
</xsd:group>

<xsd:complexType name="SubObjectPropertyOf">
```

```
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:choice>
          <!-- This is the subproperty expression or the property chain -->
          <xsd:group ref="owl:ObjectPropertyExpression"/>
          <xsd:element name="ObjectPropertyChain" type="owl:ObjectPropertyChain
              "/>
        </xsd:choice>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <!-- This is the superproperty expression -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SubObjectPropertyOf" type="owl:SubObjectPropertyOf"/>

<xsd:complexType name="ObjectPropertyChain">
  <xsd:sequence>
    <xsd:group ref="owl:ObjectPropertyExpression" minOccurs="2" maxOccurs="
        unbounded"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="xml:specialAttrs"/>
</xsd:complexType>

<xsd:complexType name="EquivalentObjectProperties">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression" minOccurs="2" maxOccurs="
            unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="EquivalentObjectProperties" type="owl:
    EquivalentObjectProperties"/>

<xsd:complexType name="DisjointObjectProperties">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression" minOccurs="2" maxOccurs="
            unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
```

```
</xsd:complexType>
<xsd:element name="DisjointObjectProperties" type="owl:DisjointObjectProperties
    "/>

<xsd:complexType name="ObjectPropertyDomain">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <xsd:group ref="owl:ClassExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectPropertyDomain" type="owl:ObjectPropertyDomain"/>

<xsd:complexType name="ObjectPropertyRange">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <xsd:group ref="owl:ClassExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectPropertyRange" type="owl:ObjectPropertyRange"/>

<xsd:complexType name="InverseObjectProperties">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression" minOccurs="2" maxOccurs="
            2"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="InverseObjectProperties" type="owl:InverseObjectProperties"
    />

<xsd:complexType name="FunctionalObjectProperty">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="FunctionalObjectProperty" type="owl:FunctionalObjectProperty
    "/>

<xsd:complexType name="InverseFunctionalObjectProperty">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="InverseFunctionalObjectProperty" type="owl:
    InverseFunctionalObjectProperty"/>

<xsd:complexType name="ReflexiveObjectProperty">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ReflexiveObjectProperty" type="owl:ReflexiveObjectProperty"
    />

<xsd:complexType name="IrreflexiveObjectProperty">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="IrreflexiveObjectProperty" type="owl:
    IrreflexiveObjectProperty"/>

<xsd:complexType name="SymmetricObjectProperty">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SymmetricObjectProperty" type="owl:SymmetricObjectProperty"
    />

<xsd:complexType name="AsymmetricObjectProperty">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="AsymmetricObjectProperty" type="owl:AsymmetricObjectProperty
    "/>

<xsd:complexType name="TransitiveObjectProperty">
  <xsd:complexContent>
    <xsd:extension base="owl:ObjectPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="TransitiveObjectProperty" type="owl:TransitiveObjectProperty
    "/>

<!-- Data property axioms -->

<xsd:complexType name="DataPropertyAxiom" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="owl:Axiom"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:group name="DataPropertyAxiom">
  <xsd:choice>
    <xsd:element ref="owl:SubDataPropertyOf"/>
    <xsd:element ref="owl:EquivalentDataProperties"/>
    <xsd:element ref="owl:DisjointDataProperties"/>
    <xsd:element ref="owl:DataPropertyDomain"/>
    <xsd:element ref="owl:DataPropertyRange"/>
    <xsd:element ref="owl:FunctionalDataProperty"/>
  </xsd:choice>
```

```
</xsd:group>

<xsd:complexType name="SubDataPropertyOf">
  <xsd:complexContent>
    <xsd:extension base="owl:DataPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression"/>
        <!-- This is the subproperty expression -->
        <xsd:group ref="owl:DataPropertyExpression"/>
        <!-- This is the superproperty expression -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SubDataPropertyOf" type="owl:SubDataPropertyOf"/>

<xsd:complexType name="EquivalentDataProperties">
  <xsd:complexContent>
    <xsd:extension base="owl:DataPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression" minOccurs="2" maxOccurs="
            unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="EquivalentDataProperties" type="owl:EquivalentDataProperties
    "/>

<xsd:complexType name="DisjointDataProperties">
  <xsd:complexContent>
    <xsd:extension base="owl:DataPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression" minOccurs="2" maxOccurs="
            unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DisjointDataProperties" type="owl:DisjointDataProperties"/>

<xsd:complexType name="DataPropertyDomain">
  <xsd:complexContent>
    <xsd:extension base="owl:DataPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression"/>
        <xsd:group ref="owl:ClassExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DataPropertyDomain" type="owl:DataPropertyDomain"/>

<xsd:complexType name="DataPropertyRange">
  <xsd:complexContent>
    <xsd:extension base="owl:DataPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression"/>
        <xsd:group ref="owl:DataRange"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DataPropertyRange" type="owl:DataPropertyRange"/>

<xsd:complexType name="FunctionalDataProperty">
  <xsd:complexContent>
    <xsd:extension base="owl:DataPropertyAxiom">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="FunctionalDataProperty" type="owl:FunctionalDataProperty"/>

<!-- Datatype definitions -->

<xsd:complexType name="DatatypeDefinition">
  <xsd:complexContent>
    <xsd:extension base="owl:Axiom">
      <xsd:sequence>
        <xsd:element ref="owl:Datatype"/>
        <xsd:group ref="owl:DataRange"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DatatypeDefinition" type="owl:DatatypeDefinition"/>

<!-- Key axioms -->

<xsd:complexType name="HasKey">
  <xsd:complexContent>
```

```
    <xsd:extension base="owl:Axiom">
      <xsd:sequence>
        <xsd:group ref="owl:ClassExpression"/>
        <xsd:group ref="owl:ObjectPropertyExpression" minOccurs="0" maxOccurs="
            unbounded"/>
        <xsd:group ref="owl:DataPropertyExpression" minOccurs="0" maxOccurs="
            unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="HasKey" type="owl:HasKey"/>

<!-- Assertions -->

<xsd:complexType name="Assertion" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="owl:Axiom"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:group name="Assertion">
  <xsd:choice>
    <xsd:element ref="owl:SameIndividual"/>
    <xsd:element ref="owl:DifferentIndividuals"/>
    <xsd:element ref="owl:ClassAssertion"/>
    <xsd:element ref="owl:ObjectPropertyAssertion"/>
    <xsd:element ref="owl:NegativeObjectPropertyAssertion"/>
    <xsd:element ref="owl:DataPropertyAssertion"/>
    <xsd:element ref="owl:NegativeDataPropertyAssertion"/>
  </xsd:choice>
</xsd:group>

<xsd:complexType name="SameIndividual">
  <xsd:complexContent>
    <xsd:extension base="owl:Assertion">
      <xsd:sequence>
        <xsd:group ref="owl:Individual" minOccurs="2" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SameIndividual" type="owl:SameIndividual"/>

<xsd:complexType name="DifferentIndividuals">
  <xsd:complexContent>
    <xsd:extension base="owl:Assertion">
      <xsd:sequence>
```

```
        <xsd:group ref="owl:Individual" minOccurs="2" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DifferentIndividuals" type="owl:DifferentIndividuals"/>

<xsd:complexType name="ClassAssertion">
  <xsd:complexContent>
    <xsd:extension base="owl:Assertion">
      <xsd:sequence>
        <xsd:group ref="owl:ClassExpression"/>
        <xsd:group ref="owl:Individual"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ClassAssertion" type="owl:ClassAssertion"/>

<xsd:complexType name="ObjectPropertyAssertion">
  <xsd:complexContent>
    <xsd:extension base="owl:Assertion">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <xsd:group ref="owl:Individual"/>
        <!-- This is the source invididual  -->
        <xsd:group ref="owl:Individual"/>
        <!-- This is the target individual -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="ObjectPropertyAssertion" type="owl:ObjectPropertyAssertion"
    />

<xsd:complexType name="NegativeObjectPropertyAssertion">
  <xsd:complexContent>
    <xsd:extension base="owl:Assertion">
      <xsd:sequence>
        <xsd:group ref="owl:ObjectPropertyExpression"/>
        <xsd:group ref="owl:Individual"/>
        <!-- This is the source invididual  -->
        <xsd:group ref="owl:Individual"/>
        <!-- This is the target individual -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
```

```
</xsd:complexType>
<xsd:element name="NegativeObjectPropertyAssertion" type="owl:
    NegativeObjectPropertyAssertion"/>

<xsd:complexType name="DataPropertyAssertion">
  <xsd:complexContent>
    <xsd:extension base="owl:Assertion">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression"/>
        <xsd:group ref="owl:Individual"/>
        <!-- This is the source invididual  -->
        <xsd:element ref="owl:Literal"/>
        <!-- This is the target individual -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="DataPropertyAssertion" type="owl:DataPropertyAssertion"/>

<xsd:complexType name="NegativeDataPropertyAssertion">
  <xsd:complexContent>
    <xsd:extension base="owl:Assertion">
      <xsd:sequence>
        <xsd:group ref="owl:DataPropertyExpression"/>
        <xsd:group ref="owl:Individual"/>
        <!-- This is the source invididual  -->
        <xsd:element ref="owl:Literal"/>
        <!-- This is the target individual -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="NegativeDataPropertyAssertion" type="owl:
    NegativeDataPropertyAssertion"/>

<!-- Annotations  -->

<xsd:complexType name="IRI">
  <xsd:simpleContent>
    <xsd:extension base="xsd:anyURI">
      <xsd:attributeGroup ref="xml:specialAttrs"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<xsd:element name="IRI" type="owl:IRI"/>

<xsd:complexType name="AbbreviatedIRI">
```

```
  <xsd:simpleContent>
    <xsd:extension base="owl:abbreviatedIRI">
      <xsd:attributeGroup ref="xml:specialAttrs"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<xsd:element name="AbbreviatedIRI" type="owl:AbbreviatedIRI"/>

<xsd:group name="AnnotationSubject">
  <xsd:choice>
    <xsd:element ref="owl:IRI"/>
    <xsd:element ref="owl:AbbreviatedIRI"/>
    <xsd:element ref="owl:AnonymousIndividual"/>
  </xsd:choice>
</xsd:group>

<xsd:group name="AnnotationValue">
  <xsd:choice>
    <xsd:element ref="owl:IRI"/>
    <xsd:element ref="owl:AbbreviatedIRI"/>
    <xsd:element ref="owl:AnonymousIndividual"/>
    <xsd:element ref="owl:Literal"/>
  </xsd:choice>
</xsd:group>

<xsd:complexType name="Annotation">
  <xsd:sequence>
    <xsd:group ref="owl:annotationAnnotations"/>
    <xsd:element ref="owl:AnnotationProperty"/>
    <xsd:group ref="owl:AnnotationValue"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="xml:specialAttrs"/>
</xsd:complexType>
<xsd:element name="Annotation" type="owl:Annotation"/>

<xsd:group name="axiomAnnotations">
  <xsd:sequence>
    <xsd:element ref="owl:Annotation" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:group>

<xsd:group name="ontologyAnnotations">
  <xsd:sequence>
    <xsd:element ref="owl:Annotation" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:group>
```

```
<xsd:group name="annotationAnnotations">
  <xsd:sequence>
    <xsd:element ref="owl:Annotation" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:group>

<!-- Annotation axioms -->

<xsd:complexType name="AnnotationAxiom" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="owl:Axiom"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:group name="AnnotationAxiom">
  <xsd:choice>
    <xsd:element ref="owl:AnnotationAssertion"/>
    <xsd:element ref="owl:SubAnnotationPropertyOf"/>
    <xsd:element ref="owl:AnnotationPropertyDomain"/>
    <xsd:element ref="owl:AnnotationPropertyRange"/>
  </xsd:choice>
</xsd:group>

<xsd:complexType name="AnnotationAssertion">
  <xsd:complexContent>
    <xsd:extension base="owl:AnnotationAxiom">
      <xsd:sequence>
        <xsd:element ref="owl:AnnotationProperty"/>
        <xsd:group ref="owl:AnnotationSubject"/>
        <xsd:group ref="owl:AnnotationValue"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="AnnotationAssertion" type="owl:AnnotationAssertion"/>

<xsd:complexType name="SubAnnotationPropertyOf">
  <xsd:complexContent>
    <xsd:extension base="owl:AnnotationAxiom">
      <xsd:sequence>
        <xsd:element ref="owl:AnnotationProperty"/>
        <!-- This is the subproperty -->
        <xsd:element ref="owl:AnnotationProperty"/>
        <!-- This is the superproperty -->
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

```
  <xsd:element name="SubAnnotationPropertyOf" type="owl:SubAnnotationPropertyOf"
      />

  <xsd:complexType name="AnnotationPropertyDomain">
    <xsd:complexContent>
      <xsd:extension base="owl:AnnotationAxiom">
        <xsd:sequence>
          <xsd:element ref="owl:AnnotationProperty"/>
          <xsd:choice>
            <xsd:element ref="owl:IRI"/>
            <xsd:element ref="owl:AbbreviatedIRI"/>
          </xsd:choice>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="AnnotationPropertyDomain" type="owl:AnnotationPropertyDomain
      "/>

  <xsd:complexType name="AnnotationPropertyRange">
    <xsd:complexContent>
      <xsd:extension base="owl:AnnotationAxiom">
        <xsd:sequence>
          <xsd:element ref="owl:AnnotationProperty"/>
          <xsd:choice>
            <xsd:element ref="owl:IRI"/>
            <xsd:element ref="owl:AbbreviatedIRI"/>
          </xsd:choice>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="AnnotationPropertyRange" type="owl:AnnotationPropertyRange"
      />

</xsd:schema>
```

.3 VBDO GRAMMAR

```
grammar org.xtext.domainmodel.DomainModel with org.eclipse.xtext.common.Terminals

generate domainModel "http://www.xtext.org/domainmodel/DomainModel"
```

```
Model:
  ('Model:' | 'model:') name = QualifiedName
  (imports = Imports)?
  entities += Entity+
;

Imports:
  ('Read:' | 'read:') imports += Import+
;

Import:
  '+' importedNamespace = QualifiedNameWithWildcard
;

enum PrimitiveType:
  CharacterSequence | Integer | Real | Date | Boolean | Character | Bytes
;

enum DefaultType:
  CTLObjBaseImpl2 |
  GISCfgDataObjImplSer |
  GISDataCltImpl |
  GISDataCltObjImplSer |
  GISDataObjImplSer |
  GIVCltImpl |
  GIVObjImpl |
  I2SCltImpl |
  I2SCurrencyAmount |
  I2SDataObjImpl |
  I2SDataObjSerializableImpl |
  I2SHome
;

DefaultWrapper:
  default = DefaultType
;

PrimitiveTypeWrapper:
  primitive = PrimitiveType
;

Entity:
  article = ('A' | 'An' | 'a' | 'an')? name = ID ('is a' (superEntity = [Entity |
       QualifiedName] | superDefault = DefaultWrapper) and = ('and')?)? ('has:'
      fields += Field+)?
;
```

```
Field:
  '-' multiplicity = Multiplicity inlineAlias = InlineAlias
;

InlineAlias:
  name = ID (plural='(s)')? ', here referring to' (type = PrimitiveTypeWrapper |
      default = DefaultWrapper | entity = [Entity | QualifiedName])
;

Multiplicity:
  {SpecificMultiplicity} number = INT |
  {ManyMultiplicity} ('Many' | 'many') |
  {PossiblyOneMultiplicity} ('Possibly one' | 'possibly one') |
  {OneMultiplicity} ('One' | 'one')
;

QualifiedName:
  ID ('.' ID)*
;

QualifiedNameWithWildcard:
  QualifiedName '.*'?
;
```

A

DETAILS OF RESULTS

Details of results whose length would compromise readability of main text.

B

LISTINGS

Should this be the case

C

TOOLING

(Should this be the case)