Using EMF and ATL to improve primitive types management in MDE proposals

Juan M. Vara, Verónica Bollati, Emanuel A. Irrazábal, Esperanza Marcos
Grupo de Investigación Kybele
Universidad Rey Juan Carlos
Madrid (España)
{juanmanuel.vara, veronica.bollati, emanuel.irrazabal, esperanza.marcos}@urjc.es

Abstract. An additional complexity inherent to any model-driven engineering proposal is the task of modeling the primitive types supported by each technological platform. If we want the models to be used as the input to generate directly the code that implements the system, the model has to be both complete and detailed. Otherwise, the code generation process results in mere skeletons of the working-code, requiring from a lot of hand-programming to have a fully operable system. This work sketches a simple, yet efficient technique for modeling of primitive types in platform specific models and shows its application for the modeling of Object-Relational databases. As well, we explore the implications of modeling primitive types in the development of model transformations. To that end, we specify a generic way of addressing this issue and show its application using again the Object-Relational databases modeling scenario.

Key words: Primitive Types Modeling, Model Transformations, ATL

1 Introduction

In recent years, Model Driven Engineering (MDE) [2, 18] and more specifically, Model-Driven Software Development (MDSD) have begun to achieve certain levels of maturity. MDSD proposes the use of models to represent the Information System (IS) at different abstraction levels. Such models are subsequently bridged by means of model (to model) transformations until their level of abstraction is that of the underlying platform. Then, a last step is encoded in another model (to text) transformation. All this given, the only way to get a full return of MDSD is by providing with the technical support for each task related with a MDSD proposal [1].

A huge amount of the proposals that have emerged during these years have followed the distinction of abstraction levels proposed by the OMG's Model-Driven Architecture (MDA) [14]. This way, following the MOF specification [13], MDA defines three levels of abstraction for modeling purposes: system requirements are modeled by Computer Independent Model (CIM), whereas Platform Independent Models (PIM) are intended to represent the functionality and structure of the system without considering technical details of the targeted platform; finally, Platform Specific Models (PSM) are used to combine the specifications contained in the PIMs with the details of the selected platform. From the different PSMs one can automatically generate different implementations for the same system.

For the last step being performed, i.e. to carry out the last model to text transformation, we need from very accurate and detailed models that provide just with a thin abstraction layer over the working-code. In this context, one of the main challenges that MDE has to face in order to be

adopted by the industry is the definition of more and more accurate DSLs [6] that allows modeling the system in a very detailed way [4, 19].

In this regard, a recurring problem is the management of primitive types. If your low-level models are meant to be used as mere plans of the underlying, the modeling of primitive types requires from special attention and treatment. Otherwise, the result of the code generation will be just a simple skeleton that needs to a lot of programming to become useful code. The latter was the way forward of the first CASE tools of the late 80s, such as Rational Rose and one of the main reasons for failure [8].

Using Database Management Systems (DBMS) as an example of technological platform, this work proposes a number of techniques to improve the management of primitive types in MDE proposals. Such techniques are structured into two main lines: support the modeling of primitive types in PSM models in an efficient and systematic manner and handling primitive types when developing model transformations, with special emphasis on horizontal transformations between PSM models. For each problem, we propose a generic technique enough to provide with a usable protocol for any given scenario. As well, we provide with a proof of concept. To that end, we present a DSL for modeling standard Object-Relational Database (ORDB) schemes and the transformation to move from such DSL to the corresponding one for Oracle. These DSLs as well as the transformation are bundled in M2DAT-DB (MIDAS MDA Tool for DataBases), a framework for model-driven development of modern DB schemas that support the whole development cycle, from PIM to working code [22].

This paper is organized as follows: Section 2 presents the main issues related with modeling of primitive data types in SQL:2003 and how EMF-tree like editors are adapted accordingly. Section 3 presents some techniques for primitive data type management in model transformation and illustrate them by means of two different scenarios. Finally, in section 4 presents the main conclusion and future works.

2 Modeling of primitive types in platform specific models

This section presents the technique proposed to model primitive types in platform specific models and its application to the definition of a DSL for modeling ORDB schemas conforming to the SQL:2003 standard [9].

2.1 Modeling of primitive types

Typically, primitive type systems follow a hierarchical structure. One can infer such conclusion from reviewing and studying how some of the most common and adopted technological platforms, such as Java [5] or the different DBs vendors implementing the SQL: 2003 standard [9], handle primitive types. The different groups and families of primitive types are subsequently specialized until reaching the leaves of the tree, where the concrete and instantiable types are found. For example, Fig. 1 shows the primitive types system of the DB2 DBMS [7].

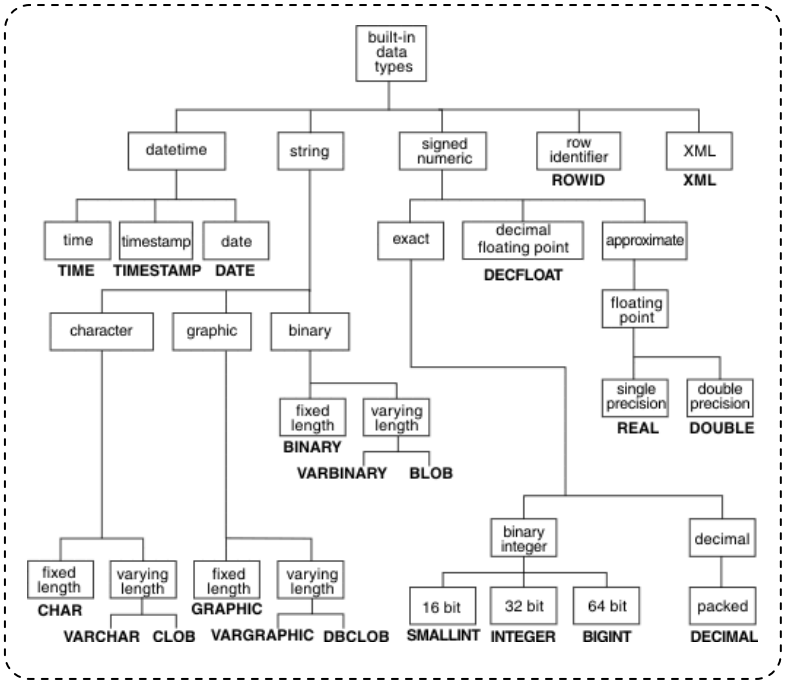

Fig. 1. Primitive types system of DB2

When defining a DSL to model DB schemas, this fact hampers the definition of the underlying metamodel that captures the abstract syntax of the DSL. Although hierarchical structures are simple to model, the large number of final elements or leaves that have to be included adds complexity to the metamodel, and consequently to any terminal model conforming to such metamodel. An alternative is grouping together into families of data types the leaves of the tree and using *enum* data types to identify which specific type is.

Fig. 2 shows a partial view of the SQL:2003 metamodel defined this way.

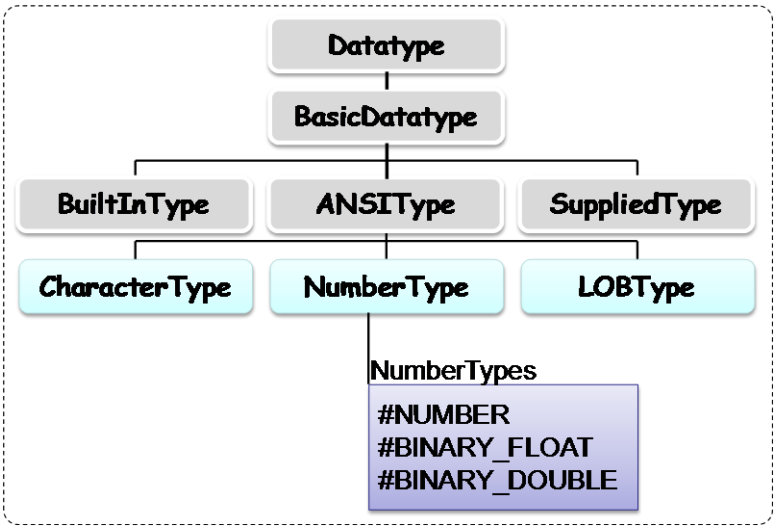

Fig. 2. Partial view of the primitive types system of the SQL:2003 standard

Numeric types are grouped into the *NumberType* class. It owns an attribute whose possible values may be: *NUMBER*, *BINARY_FLOAT* and *BINARY_DOUBLE*. Working this way, we do not need to include a different class for each leave in the primitive types system.

However, for each data type we have to consider a set of inherent features. When a given primitive type is used to define the type of an element in the model, we have to assign a value for each feature (even if we decide for using default values, we need a way to do so). That assignment is valid just for that specific use of the data type. For example, Fig. 3 shows a *Customer* table owning two columns: *Name* and *Address*, both of *Character* type. We can set different values for the *size* feature, which restricts the maximum size that can reach each column.

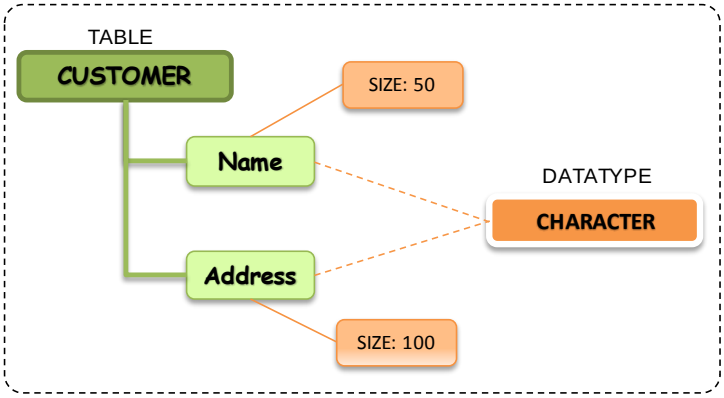

Fig. 3. Using features to restrict the size of character type elements

To address this issue we advocate in favor of modeling such features while keeping a reasonable level of complexity in the DSL metamodel. The proposal can be decomposed into two steps:

- On the one hand, adding a new metaclass (*Feature*) to represent the features inherent to any given primitive type. This metaclass is later specialized to define a valid set of features metaclasses for each family of types. Each specialization consists of a key-value pair. The key is defined over an *enum* data type that collects the valid features for each final type. The value represents the value assigned to that feature for that specific use of the primitive data type.
- On the other hand, each model element which may be typed by means of a primitive type might contain several feature objects. Those objects model the features of the primitive type when it is used to type the given object.

Fig. 4 shows the result of applying the technique sketched to the previous example. Again, the columns *Name* and *Address* use the same data type, but each one customizes the data type according to their needs. To that end, both *Address* and *Name* include a feature object whose possible values for the key are taken from an enumerated data type that specifies which the valid features characteristics of character data types are.

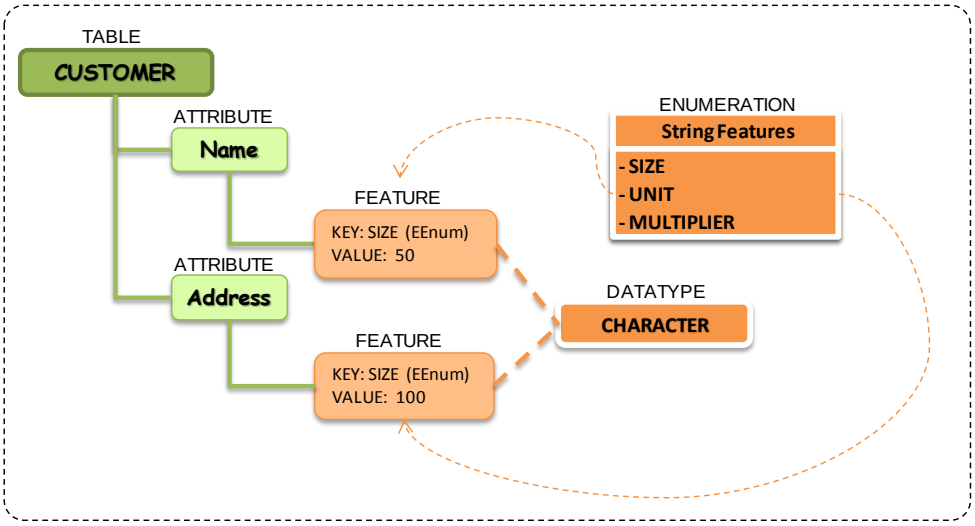

Fig. 4. Using *feature* objects to model the restriction of size for character data types

Nevertheless, by extending the DSL metamodel to support the modeling of primitive types, we are also widening the range of terminal models that are valid (syntactically speaking). However, the specification of the abstract syntax of a DSL is not limited to the definition of the metamodel. To complete the specification we might add constraints at metamodel level to gather those domain rules that cannot be directly reflected in the metamodel. Later on, we will show how we have addressed this task during the implementation of the proposed technique.

2.2 Modeling of primitive types in SQL:2003

In the following, we show the result of applying the technique described above to the specification of a DSL for modeling ORDB schemas conforming to the SQL:2003 standard.

Fig. 5 shows a partial view of the metamodel which defines the abstract syntax of the DSL focused in the metaclasses related with representing primitive types.

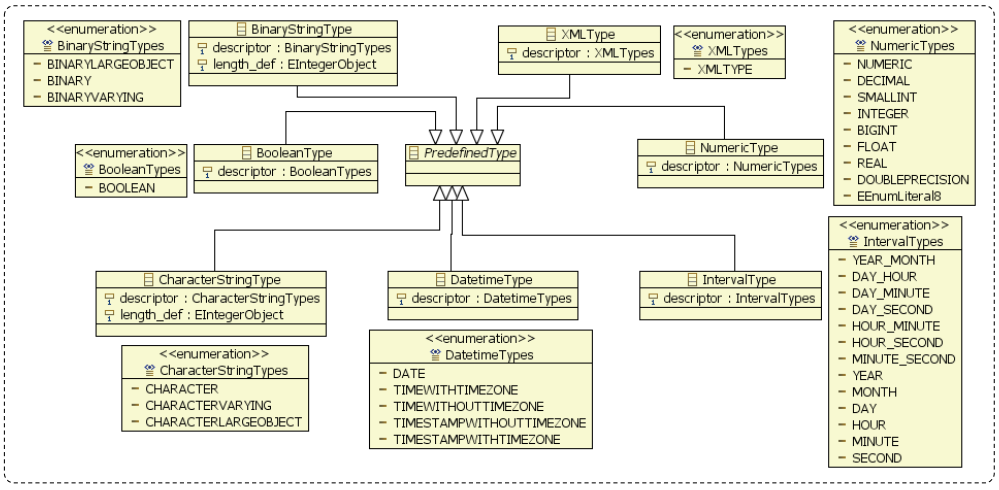

Fig. 5. Partial view of the SQL:2003 metamodel: primitive types

Based on the proposal sketched in the previous section, Fig. 6 shows how the abstractions needed to model the features of each primitive type are included in the metamodel.

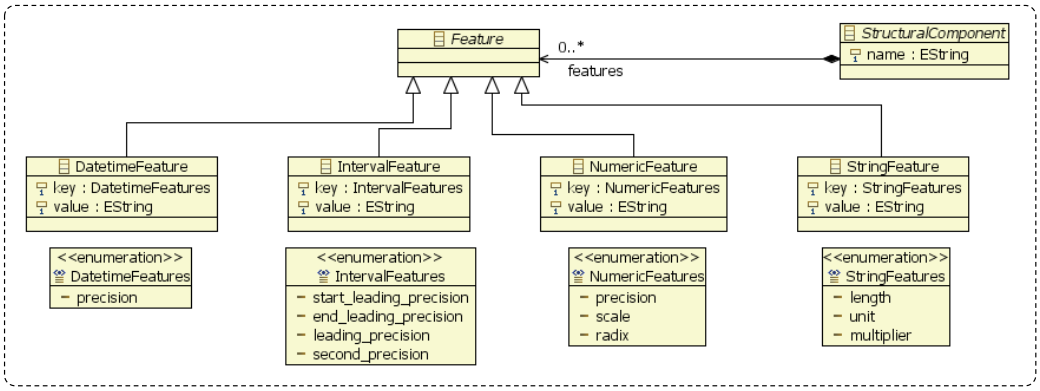

Fig. 6. Partial view of the SQL:2003 metamodel: features.

Note that it also follows a hierarchical pattern: the features are grouped together into families that match the existing families of primitive data types. Primitive data types are used to type columns or attributes, both factorized in the *Structural Component* metaclass. Therefore, each *Structural Component* can contain a set of *Feature* objects. They serve to specify the value for the features of the primitive type over which the given column or attribute is defined. Finally, the values that the key property can take vary from one family of primitive data types to the other. Thus, an enumerated data type collects the valid values for each case. For instance, we can fix the *precision* of a *Datetime* column/attribute or we can set the *scale* of a *Numeric* column/ attribute.

At the end of previous section we put forward the need to attach a number of restrictions to the metamodel to complete the specification of the DSL abstract syntax. In particular, we must control which are the features that can be attached to any model element: just those inherent to the primitive data type used to type the object.

Although there are several ways to define metamodel level constraints that are later checked at model level, the most accepted in the context of MDE is the use of OCL (or an OCL-based language). In this work, we use the Epsilon Validation Language [11] (EVL) to complete the DSL specification. EVL extends the OCL capabilities in terms of model validation: it improves OCL expressiveness in order to ease the coding of constraints and model navigation, it bundles an API to provide with user feedback on constraints violation in a friendly manner, it support the definition of *fixes*, that are actions to repair any detected inconsistency , etc.

To show a simple example of the use of EVL in this work, Fig. 7 shows one of the EVL restrictions coded. In particular, an invariant that ensures that only character features can be attached to model objects whose type is one of those from the family of character data types.

```
-- String features can be binded just to CharacterString or BinaryString types
constraint OnlyStringFeature {
        guard : self.satisfies('notEmptyDistinctTypeSourceType') and
                (self.source_type.isTypeOf(CharacterStringType) or self.source_type.isTypeOf(BinaryStringType))
                and self.features->notEmpty()
        check : self.features->forAll(f | f.isTypeOf(StringFeature))
        message : getMessageOnlyFeature('source type', self.source_type.descriptor.asString(), 'Distinct Type', self.name, 'String')
}
```

Fig. 7. EVL invariant to ensure correct use of character features

To conclude this section, Fig. 8 shows an example of use of the DSL specified following the proposed technique. We focus on how to add a new feature to a particular attribute.

Fig. 8. Modeling of characteristic in elements defined on pritimitve types

First, we add a *Feature* object, in the *country* attribute. Since the type of the attribute is character (CHARACTERVARYING), the *Feature* object is an instance of the *StringFeature* class. Indeed, if we try to use any other *Feature* subclass, model validation would raise an error. Next, we use the key-value properties of the feature object to set the size (*length*) of the attribute. Finally, we show how the result is displayed in the (improved) EMF-tree editor: the *country* attribute is of CHARACTERVARYING and its length is limited to 25 chars. Note that, proceeding this way, the model contains all the information needed to generate the SQL script that implements the modeled DB scheme.

2.3 Customizing EMF-tree like editors to improve primitive types management

The specification of a DSL supporting the entire system of primitive data types of a specific platform requires conducting certain considerations in terms of usability of the associated DSL toolkit. Indeed, usability has been widely acknowledged as one of the main issues of existing tooling for MDE nowadays [17].

As we will show in this section, deploying the presented proposal for modeling primitive types has a direct impact over the usability of the EMF-tree like editors bundled in the DSL toolkit. Therefore in the following we show how we mitigate this impact by customizing such editors to improve their level of usability.

Automatic inclusion of primitive types in PSM models. When working with PSMs models we have to model explicitly each primitive data type to be used. For example, back to the DSL we are using as proof of concept for this work, if we want a model element to be of CHARACTER VARYING type, we must instantiate the *Character String* metaclass in the model and assign the value CHARACTER VARYING to the descriptor property *descriptor* of such object.

To address this issue we propose to follow the practice adopted by commercial modeling tools. They use to include by default all the primitive data types supported by the targeted platform in any new model. Thus, the user has only to choose the most appropriate for each model element. Following this practice we propose to modify the code generated by EMF so that every time you create a new PSM model, metaclasses abstracting primitive data types are automatically instantiated.

Fig. 9 shows the result of using the generic EMF editor (a) versus using the improved one (b).

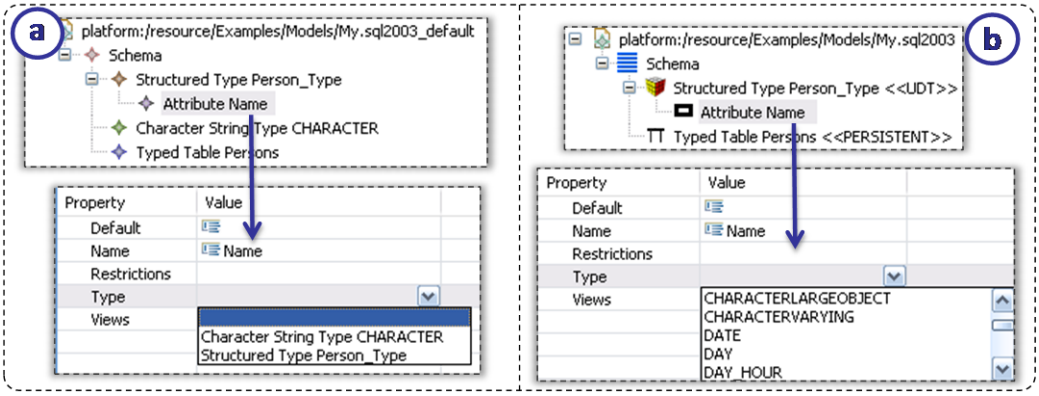

Fig. 9. Assigning primitive types in EMF editors: (a) Generic editor (b) Improved editor

In both cases the purpose is the same: using the CHARACTERVARYING type to type the *Name* attribute of the structured type *Person_Type*. However, when we use de editor generated by EMF (a), we need first to create a *Character String Type* object. Furthermore, we can only use this type and the *Person_Type* type to define the type of the attribute (i.e. only those types that have been previously created on the model can be used in new objects).

By contrast, when using the custom editor (b), we can use any of the primitive types supported by the SQL: 2003 standard, since they have been all instantiated at the time of creating the model. Of course, this instantiation process remains hidden for the final user.

Finally, it should be mentioned that by applying this technique on the tree-like editor of EMF, the functionality is automatically reflected in the diagrammer built with the Generic Modeling Framework [20] (GMF), since the code that implements the latter is partially based the one implementing the former.

Filtering of primitive data types. If the improvement of the EMF editor is limited to including by default the set of primitive data types supported by the platform, the result may still be unwieldy since the model will include a large number of objects - those that represent each one of the primitive data types, which are not really relevant for the construction of the model. In fact, these objects will not be modified during model edition: they are created at the time the model is created and play a secondary role for the rest of the modeling process. They are simply used to define the type of other model elements. Despite that, they only serve to add noise to any view of the model.

Therefore, we have decided to customize EMF editors in order to filter the set of displayed objects, so that primitive data types remain hidden and they are only shown when they are really needed, i.e. when the user has to set the type for a given attribute or column. To that end, the set of available primitive data types is displayed in the combo box shown in the properties view that allow defining the type of any given object of the model (see Fig. 9-b).

Fig. 10 shows the difference between using a non-improved editor, where primitive data types have not been filtered (a) versus one in which the filtering is applied (b). Note that, for the sake of space, just a very few number from the set of the primitive data types supported is shown; otherwise, the figure had been too large. Although the functionality provided by both editors is exactly the same - both allow you to use any of the primitive data types supported by SQL: 2003 to define the type of a model element, their levels of usability differ widely.

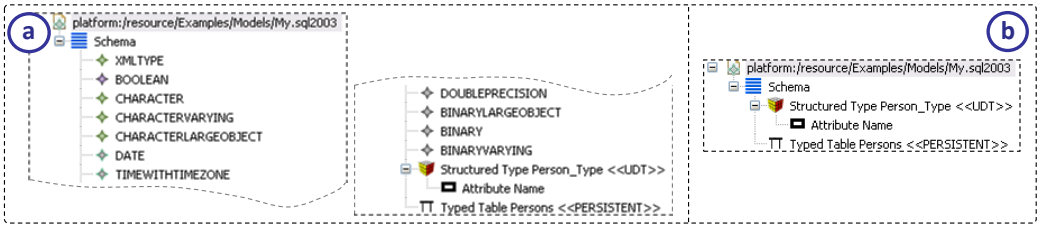

Fig. 10. Filtering of primitive types in EMF editors

3 Primitive types management in model transformations

So far, we have shown how to address the specification of a DSL that supports modeling of primitive data types in a complete, correct and detailed enough manner to use any model as input for a code generator. From now on, we focus on the implications of modeling primitive data types with regard to the development of model transformations in which models. Moreover, we show how we address these concerns using the ATL transformation language [10].

3.1 Vertical transformations (PIM2PSM)

In general, PIM to PSM mappings does not need from additional considerations, since the primitive data types considered at PIM level, which use to be reduced to an integer type, a character type, and optionally a boolean type, are mapped to one of the primitive data types supported by the targeted platform. To do so, we just need to add a set of matched rules to map each primitive data type included in the PIM.

Besides, according to the editor's improvements presented in section 2.3, the transformation has to take care of creating the objects needed to model whole set of primitive data types in the target model. This way, we are ensuring that the designer can use them in order to refine the model obtained. To do so, we just need to include an imperative rule that create such objects. In particular, it is an *end point* rule, an ATL rule that is automatically executed just before the transformation execution is finished.

Continuing with the DSL for ORDB modeling, Fig. 11 shows the imperative rule included in the UML2SQL2003 ATL transformation developed to move from pure conceptual data models to ORDB schemas for the SQL:2003 standard.

```
endpoint rule generateTypes(){
        to
                datetime_timewithtimezone : SQL2003!DatetimeType (
                        descriptor <-#TIMEWITHTIMEZONE,
                        schema <- thisModule.PACKAGE()
                ),
                datetime_timewithouttimezone : SQL2003!DatetimeType (
                        descriptor <-#TIMEWITHOUTTIMEZONE,
                        schema <- thisModule.PACKAGE()
                ),
```

Fig. 11. ATL rule to create primitive types (UML2 to SQL2003)

Note that it only contains a target pattern, i.e. it just instantiate new objects in the target model, without the need for a previous matching with some source elements.

3.2 Horizontal transformations (PSM2PSM)

The task of mapping primitive types in PSM2PSM transformations is more challenging. Apart from mapping the primitive types, we need to map the features that each element uses to customize the primitive data type used.

We propose two different techniques to tackle these issues: one for mapping the primitive data type objects, and, another one for the *Feature* objects. Next, we introduce them using the SQL20032ORDB4ORA transformation bundled in M2DAT-DB [22]. It maps ORDB schemas conforming to the SQL:2003 standard to ORDB schemas for Oracle.

Mapping primitive data type objects. Table 1 identifies the most common scenarios that can occur in terms of correspondence between primitive data type systems of different platforms

Table 1. Scenarios in primitive data types management for PSM2PSM transformations

SOURCE MODEL	TARGET MODEL
One source type	One target type
One source type	Any target type
One source type	Multiple target type
Any source type	One target type
Multiple source type	One target type

In general, all scenarios but the last one can be addressed by means of fairly intuitive rules, or at least, they can be decomposed into simpler scenarios. Therefore we will focus on the last one: when multiple source types have to be mapped to the same type in the target model. This scenario is more complex by nature.

Indeed, a great part of the logic of a (declarative) model transformation resides largely on the automatic management of the traces between elements from source and target models. Any reference to a source element in a target pattern is automatically replaced to a reference to the corresponding target element. However, in a N:1 scenario like the one we tackle here, we cannot use automatic traces management, since the target type corresponds to several source types: which one should be use to refer to the desired target type?. For example, both the CHAR and NCHAR data types from the SQL:2003 standard correspond to a single data type in Oracle: CHARACTER.

To address this issue we propose a two steps solution:

- First, we choose one of the source types (so-called *hidden* types) and map it to the desired target type. We refer to this source type as mirror type
- Then, each reference in the source model to one of the hidden types has to be replaced by a reference to the mirror type. This way, when the transformation is run, the engine resolves every reference to the mirror type by a reference to the appropriate target type.

The encoding of the first step is immediate. For example, Fig. 12(a) shows the rule that allows mapping the NUMERIC data type from Oracle to SQl:2003. On the other hand, Fig. 12(b) shows how the second step is articulated when mapping PARAMETER objects. When there is a need of referencing a primitive data type from source model, the *mirrorType ()* helper is invoked. Such helper returns the corresponding mirror data type. That is, the source data type of the matched rule coded in the previous step.

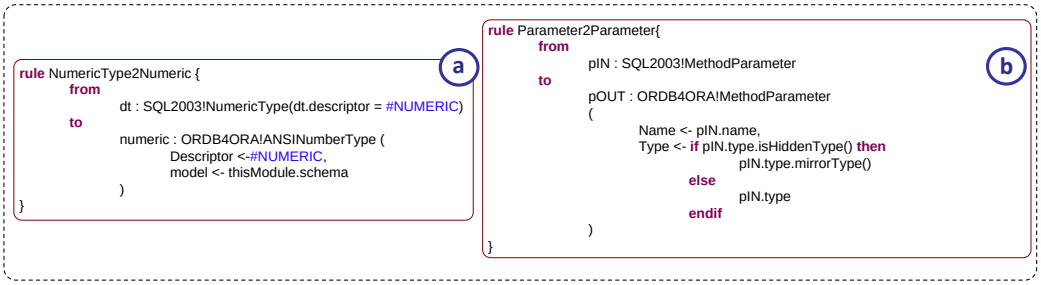

Fig. 12. ATL transformation rule of *Parameter* objects (SQL:2003 to Oracle)

3.3 Mapping *Features*

Finally, once primitive data types are correctly mapped, we need to address the mapping of the feature objects that allow customizing or restricting the use of such data types. Notice that only those source features with a corresponding feature in the target model could be mapped. The rest of information on features is lost. However, if such information is considered relevant, a valid and affordable alternative would be to add annotations to the target model to collect such data.

To illustrate the technique proposed, Fig. 13 shows how the mapping of features is solved when transforming attributes from a SQL:2003 model to the one for Oracle.

```
rule Attribute2Attribute {
        from
                attIN : SQL2003!Attribute
        to
                attOUT : ORDB4ORA!Attribute (
                        Name <- attIN.name,
                        Type <- if attIN.type.isHiddenType() then
                                        attIN.type.mirrorType()
                                else
                                        attIN.type
                                endif,
                        structured <- attIN.structured,
                        features <- attIN.features->select(f|f.haveLegalTarget())->collect(f|thisModule.Feature2Feature(f))
                        )
}
```

Fig. 13. Mapping rule for Attribute objects: SQL:2003 to Oracle

Whenever an *Attribute* is mapped, its features have to be mapped as well. To that end, we first select just those features that have a correspondent feature on the target metamodel. To filter them we use the *haveLegalTarget()* helper. Then, we invoke the rule that creates the target feature (*Feature2Feature()*) (see Fig. 14(a)).

Indeed, the *Feature2Feature* rule is an abstract rule. It maps the source key-value pair to the target key-value pair. To that end, two different helpers return the target key and the target value for each source key and source value. Taking advantage from ATL rule inheritance, the rule is later specialized for each family of primitive types. For instance, Fig. 14(b) shows how the rule is specialized for the families of String (1) and Numeric (2) primitive data types.

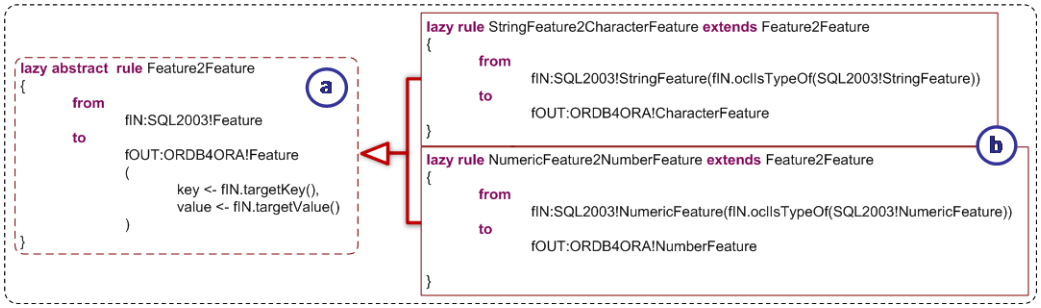

Fig. 14. Mapping rule for Features objects: SQL:2003 to Oracle

4 Conclusions

This work has introduced a number of techniques to improve the management of primitive data types in platform specific models. To that end, we have defined a series of systematic steps that allow, on the one hand, to model completely and correctly the system of primitive data types supported by each platform and, on the other hand, to keep consistent such information when the models are the source and/or the target of a model transformation.

The proposed techniques are original, simple and intuitive and we have provided with a proof of concept by showing their application in the development of a toolkit for model-driven development of ORDB schemas. The results, in terms of usability and completeness of the models developed, are quite successful.

With regard to the development of model transformations, the steps are intuitive and the task of programming the proposed techniques is not very complex. Though we have not included a complete case study for the sake of space, we have shown that the proposed solution works correctly in all scenarios identified.

It is worth mentioning that even though the implementation presented has been created using EMF as (meta)modeling framework, the underlying ideas can be easily deployed in other frameworks, such as GME [3]. Similarly, the implementation with ATL could be extrapolated to other languages adopting a hybrid approach, but essentially declarative, such as RubyTL [16], Tefkat [12] or any implementer of QVT-Relations [15].

As future work, we plan to automate the proposal using MDE techniques. Therefore, we are working to modify EMF generation templates in order to integrate the facilities for primitive data types management in any EMF-editor. As well, we work to be able to generate automatically the ATL code that implements the mapping rules for primitive data types from weaving models representing the relationships between them. In the future, such relationships should be discovered automatically by means of model matching techniques.

Acknowledgment.
This research has been carried out in the framework of the projects: MODEL-CAOS (TIN2008-03582/TIN), AGREEMENT-TECHNOLOGY (CSD2007-0022) both project financed by the Spanish Ministry of Education and Science and the IDONEO project (PAC08-0160-6141) financed by "Consejería de Ciencia y Tecnología de la Junta de Comunidades de Castilla-La Mancha".

References

1. Atkinson, C., & Kuhne, T. (2003). Model-driven development: a metamodelling foundation. *IEEE Software, 20*(5).
2. Bézivin, J. (2004). In search of a Basic Principle for Model Driven Engineering. *Novatica/Upgrade, V*(2), 21-24.
3. Davis, J. (2003). *GME: the generic modeling environment.* 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. OOPSLA '03, Anaheim, USA.
4. Fowler, M. (2009). Code Generation for Dummies. *Methods and Tools*, Spring 2009, 65-82.
5. Gosling, B. & Steele, G. (1996). *The Java Language Specification*. Addison-Wesley, 1996.
6. Gronback, R. C. (2009). *Eclipse Modelling Project: A Domain-Specific Language (DSL) Toolkit*: Addison-Wesley Professional.
7. IBM DB2 Universal Database. http://www-306.ibm.com/software/data/db2/.
8. Iivari, J. 1996. Why are CASE tools not used?. *Communications of the ACM, 39*(10), pp. 94-103.
9. ISO/IEC 9075: 2003 Information technology – Database languages – SQL:2003.
10. Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). ATL: a model transformation tool. *Science of Computer Programming, 72*(1-2), 31-39.
11. Kolovos, D., Paige, R., & Polack, F. (2006). Eclipse Development Tools for Epsilon, *Eclipse Summit Europe, Eclipse Modeling Symposium*. Esslingen, Germany.
12. Lawley, M., & Raymon, K. (2007). *Implementing a practical declarative logic-based model transformation engine.* ACM symposium on Applied computing 2007 (SAC 2007), Seoul, Korea.
13. OMG. The Meta Object Facility (MOF), Version 2.0. OMG Document - formal/06-01-01.
14. OMG. *MDA Guide Version 1.0.1* Document number omg/2003-06-01. Ed.: Miller, J. y Mukerji, J.
15. OMG. MOF 2.0 Query/View/Transformation (QVT), V1.0. OMG Document - formal/08-04-03.
16. Sánchez Cuadrado, J., García Molina, J., & Menarguez Tortosa, M. (2006). *RubyTL: A Practical, Extensible Transformation Language.* ECMDA-FA 2006, Bilbao, Spain
17. Selic, B. (2008). MDA Manifestations. *UPGRADE, IX*(2), 12-16.
18. Selic, B., *The pragmatics of Model-Driven development*, IEEE Software, Volume 20, Issue 5 (2003) pp. 19 - 25.
19. Straeten, R., Mens, T., and Baelen, S. 2009. Challenges in Model-Driven Software Engineering. In *Models in Software Engineering: Workshops and Symposia At MODELS 2008, Toulouse, France*.
20. Tikhomirov, A., & Shatali, A. (2008). *Introduction to the Graphical Modeling Framework*. Tutorial at the EclipseCON 2008. Santa Clara, California.
21. Vara, J. M., Vela, B., Bollati, V., & Marcos, E. (2009). *Supporting Model-Driven Development of Object-Relational Database Schemas: a Case Study* ICMT2009, Zurich, Switzerland.
22. Vara, J.M. (2009). M2DAT: a Technical Solution for Model-Driven Development of Web Information Systems. Ph.D. Thesis. University Rey Juan Carlos. http://www.kybele.etsii.urjc.es/members/jmvara/Thesis/ (last accessed April 2009).