UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Tecnura

http://revistas.udistrital.edu.co/ojs/index.php/Tecnura/issue/view/650

DOI: http://dx.doi.org/10.14483/udistrital.jour.tecnura.2015.2.a09

Metamodels composition strategy for the model driven engineering context

Estrategia de composicion de metamodelos para el contexto de ingeniería basada en modelos

Héctor Arturo Flórez Fernández*

Fecha de recepción: 23 de julio de 2013 **Fecha de aceptación:** 5 de diciembre de 2014

Citation / Para citar este artículo: Flórez Fernández, H. A. (2015). Metamodels composition strategy for the model driven engineering context. *Revista Tecnura, 19*(44), 121-132. doi:http://dx.doi.org/10.14483/udistrital.jour.tecnura.2015.2.a09

Abstract

In Model Driven Engineering (MDE) approaches, metamodelers usually need to create a metamodel based on existing metamodels, where each one abstracts a specific domain, in order to abstract a new domain, which includes elements that could be taken from the other already created metamodels. This kind of constructions allows getting advantage of the knowledge obtained in the construction of the previous built metamodels. This paper presents a proposal to solve metamodel composition through a Domain Specific Language (DSL). This DSL is used by metamodelers, who are the people that know the domains abstracted by the different metamodels and know how to combine those metamodels in order to generate the new one. Moreover, a simple case study is presented so as to demonstrate the low level of complexity of the DSL.

Keywords: computational modeling, metamodeling, software prototyping.

Resumen

En enfoques de ingeniería basada en modelos (MDE), los metamodeladores usualmente tienen que crear un metamodelo basado en metamodelos existentes, en donde cada uno abstrae un dominio específico con el fin de abstraer un nuevo dominio, que incluye los elementos que se podrían tomar de los otros metamodelos ya creados. Este tipo de construcciones permite obtener ventaja de los conocimientos obtenidos en la construcción de los metamodelos previamente construidos. En este trabajo se presenta una propuesta para resolver la composición de metamodelos a través de un lenguaje de dominio específico (DSL). Este DSL es utilizado por metamodeladores, que son las personas que conocen los dominios abstraídos por los diferentes metamodelos y saben cómo combinar los metamodelos para generar uno nuevo. Además, se presenta un caso de estudio simple con el fin de demostrar el bajo nivel de complejidad del DSL.

Palabras clave: metamodelamiento, modelamiento computacional, prototipo de software.

* Electronic Engineer, Computer Science Engineer, Specialist in Management, Magister in Information and Communication Sciences, Magister in Management, Doctor candidate in Engineering. Assistant professor in the Universidad Distrital Francisco José de Caldas, Bogotá, Colombia. Contacto: *haflorezf@udistrital.edu.co*

INTRODUCTION

A metamodel is a component to abstract the concepts of a specific domain of information, and it is constructed by one metamodeler, who is the person that knows the domain and can solve a problem using a MDE approach. Also, one model is a simplification of a system with an intended goal (J. Bézivin, 2005), (P.A. Muller, F.; Fondement, B. Baudry, and B. Combemale, 2009), or an artifact to represent a specific case of a domain that is constructed by modelers. In MDE approaches the models must conform to the metamodel that abstract the domain. In addition, modeling has an important role in developing software systems because it provides means to concepts (J. Henriksson, F. Heidenreich, J. Johannes, S. Zschaler, and U. Assmann, 2008), (H. Florez, 2012) abstracted in a specific domain.

Furthermore, in several cases metamodelers need to represent a new domain; however, the new domain can have concepts already abstracted in existing metamodels. Consequently, metamodelers can reuse several concepts from several metamodels in order to create the new metamodel (M. Emerson and J. Sztipanovits, 2006) (G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits, 2004).

This proposal presents a metamodel composition solution strategy, where one metamodeler, who knows several domains abstracted in correspondent metamodels, constructs a new metamodel, which is intended to abstract the new domain, based on the existing metamodels. The new metamodel is generated by creating and executing a script that has instructions defined in one DSL. Instructions of the DSL allow metamodelers to include several existing metamodels and make several operations regarding the elements, attributes, and relations included in the selected metamodels. In addition, the DSL also allows creating new elements, attributes, and relations in the composed metamodel.

The rest of the paper is structured as follows. Section 2 presents different techniques to solve the metamodel composition problem. Section 3 presents the proposed methodology and strategy for solving the metamodels composition. Section 4 presents the language developed to solve metamodels composition; also, an example about composition using the proposed language. Section 5 presents the composition engine in which the process for composition is explained. Section 6 presents a summarized case of study. In section 7, the related work is presented. Finally, section 8 presents the conclusions.

METAMODELS COMPOSITION

In MDE, metamodels composition is necessary for several reasons (A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti, 2001), (J. Oldevik, L. Kutvonen, and N. Alonistioti, 2005). One metamodel is the set of abstractions and techniques that govern how systems related with the domain are going to be modeled; as a result, metamodels represent the way in which a particular engineering domain is abstracted. When a new domain is needed to be abstracted, several previously constructed domains could represent some elements that the new domain needs to include in their abstraction. For instance, in the case that a language designer requires to create a new language, it is possible to get the knowledge included in existing languages with the purpose to reuse the common existing elements between the existing languages and the new language. Consequently, the effort in process of the construction of the new language can be reduced as much as possible getting advantage of the efforts invested in the domains taken through the correspondent metamodels. Then, metamodel composition offers benefits to Domain Specific Modeling Language (DSML) analogous to software reuse offers benefits to software engineering (M. Emerson and J. Sztipanovits, 2006). For instance, it is possible to achieve: the avoidance of duplication effort, emergence of high quality reusable metamodel fragments, recognition of metamodeling patterns, and reduction of time in the creation of new DSMLs.

Metamodel composition strategies aim to support the construction of complex metamodels using atomic transformations (J. Oldevik, L.

Kutvonen, and N. Alonistioti, 2005). In the context of MDE, there are some processes for metamodel composition: 1) matching elements, 2) elements merge, and 3) class refinement (M. Emerson and J. Sztipanovits, 2006). Matching models is a process used to identify different views of the same concept (R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh, 2007), in order to unify those several equivalent concepts in one composed concepts. This strategy can create new concepts that are enriched by different descriptions previously made in existing metamodels. Metamodel merge combines several concepts creating a new one in order to avoid collisions between the elements described in two different metamodels (M. Emerson and J. Sztipanovits, 2006) used for the metamodel composition process. Merging concepts not only allows joining concepts defined in existing metamodels, but also allows customizing those concepts by adding or removing attributes or relations. Class refinement is used to add details in one single element that has not been composed with other elements. The refined class can be taking from existing metamodels; as a result, the refinement does not imply the creation of all features of the class (e.g., attributes and relations); thus, the refinement becomes an useful mechanism for polishing the composed metamodel in order to obtain the desired abstractions.

METHODOLOGY AND SOLUTION STRATEGY

This proposal consists of a strategy where the domain experts, who are metamodelers, modify the metamodels explicitly specifying the composition process. Metamodelers know the reasons why a metamodel needs to be composed, define the set of input metamodels required to get the composed metamodel, and create a script with the operations needed to built the new metamodel. Figure 1 illustrates composition process and the metamodeler responsibilities. The metamodeler, who is intended to create a composed metamodel and understands the domains abstracted by other metamodels, selects at least two

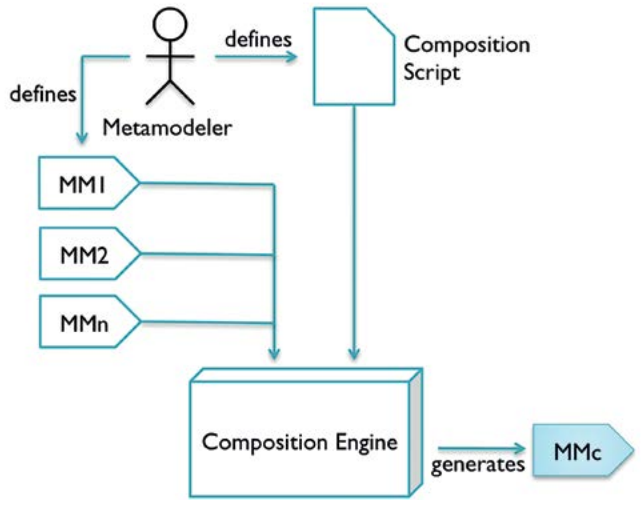

Figure 1. Composition process.

Source: own work.

existing metamodels as inputs of the process. Later on, based on the selected metamodels, he/she creates the composition script that is one source code written using the DSL presented in section 4. Finally, the metamodeler executes the composition script in the "Composition Engine" presented in section 5 providing the required metamodels and obtaining as output one composed metamodel.

The composition engine receives the script developed by metamodelers and imports the support metamodels defined in the script. These scripts are loaded in dynamic memory and manipulated with Eclipse Modeling Framework (EMF) making the modifications determined in the script in order to generate only one composed metamodel as output.

The composition engine process (see figure 2) is based on the set of instructions Δ that corresponds to several instructions ($\Delta = \{\delta_1, \delta_2, \dots, \delta_n\}$). Each instruction δ_i changes the composed metamodel MM_0 based on the support metamodels $\{MM_{sup-1}, MM_{sup-2}, \dots, MM_{sup-m}\}$. In addition, the instruction δ_i changes the affected support metamodels, so after the execution of the instruction δ_i the engine will contain a new version of the composed metamodel (MM_i) and the support metamodels {$MM_{sup-1.i}$, $MM_{sup-2.i}$, ... , $MM_{sup-n.i}$}. Thus, after the execution of the instruction δ_i, the composition engine creates the composed metamodel (MM_i) that is ready to use by the metamodeler. The composition engine also is capable to identify exceptions in the process. Then, due to several instructions can depend on the results of the execution of previous instructions; one exception finalizes the composition engine process and the composed metamodel is not created.

The work of creating metamodels can be considered demanding because it should include the analysis of the context that usually contains a big amount of elements and relations between elements. Due to this strategy is based on existing metamodels that already abstract properly domains, metamodelers do not need to make a big effort for abstracting fragments of the domain, but they can dedicate this effort for understanding the way in which the existing metamodels can support the composition process. In addition, effort for the creation of the new metamodel decreases because the composition engine is able to provide one validated composed metamodel; thus, the metamodeler just need to write one basic script in order to generate the desired result.

COMPOSITION LANGUAGE

The proposal resolves the metamodels composition by defining one Domain Specific Language (DSL). This DSL includes an instructions catalog of the possible operations that can be applied over several input metamodels in order to generate a unique output composed metamodel. The instructions presented in the catalog are created following the technique "class refinement"; however, one instruction, which is *joinClasses* is based in the technique "element merge" in order to take advantage of the specific features of this technique.

The structure of the DSL consists in the next three operations:

- Operation "import". This operation allows specifying several input metamodels.
- Operation "export". This operation allows specifying the output composed metamodel.
- Instructions. Each instruction specifies a change in the composed metamodel.

The DSL has a set of operations that allow metamodelers to define possible changes over the input metamodels in order to construct the composed metamodel, which are defined in the instruction catalog. This proposal is completeness from the principle that each instruction has high granularity, which implies that the operation cannot be decomposed into smaller operations (M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth, 2010), to ensure unitary changes on the metamodel in the composition process. As a result, the DSL has a catalog made up of 16 instructions. With these instructions metamodelers can make the necessary changes on the classes, attributes and references from the input metamodels. Also, metamodelers can include new classes, attributes and references that are not defined in any input metamodel.

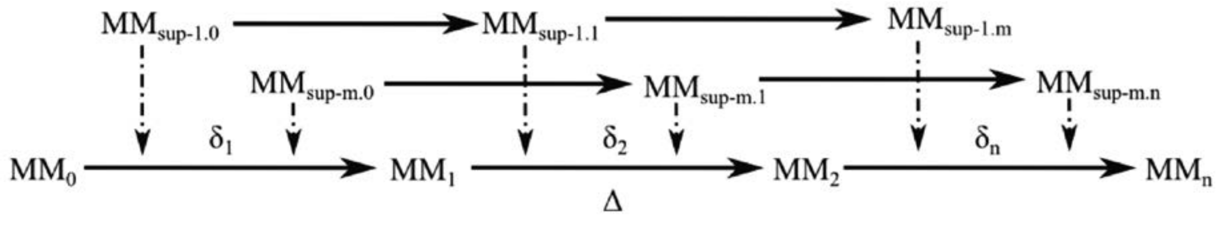

Figure 2. Composition strategy.

Source: own work.

Possible changes related with classes include create, delete, rename, set abstract, unset abstract, divide a class in several classes, and join several classes. Possible changes related with attributes include create, delete, rename, and update. Possible changes related with references include create, delete, rename, create inheritance reference, and delete inheritance reference. Table 1 presents the instruction catalog created for the composition language.

When any instruction make reference to a class, it is necessary to indicate the name of the input metamodel in which the class is placed. In the case that the instruction does not have the name of the input metamodel, the engine will search the class

Table 1. Instruction Catalog.

	Instruction	Parameters
Class	newClass	Class Name
	deleteClass	Class Name
	renameClass	Class Name New Class Name
	setAbstractClass	Class Name
	setNonAbstractClass	Class Name
	joinClasses	New Class Name Class Name 1 Class Name 2
	divideClasses	Class Name Divided Classes -Divided class name -Divided class attributes -Divided class references
Attribute	newAttribute	Class Name Attribute Name Type
	deleteAttribute	Class Name Attribute Name
	renameAttribute	Class Name Attribute Name New Attribute Name
	updateAttribute	Class Name Attribute Name Type
Reference	newReference	Reference Name Source Class Name Target Class Name Containment Min Cardinality Max Cardinality
	deleteReference	Class Name Reference Name
	updateReference	Class Name Containment Min Cardinality Max Cardinality
	newInheritanceReference	Sub Class Name Super Class Name
	deleteInheritanceReference	Sub Class Name

Source: own work.

between the classes created before in the composition process.

With this instructions catalog, the composition language offers a language that supports a great variety of metamodel composition cases.

In order to explain how the operations can be used, the next two metamodels presented in figure 3 will be used.

The goal of the example will include the next operations.

- Create a new class named "N"
- Create a new attribute named "attN1" in the class N with type EInt
- Create a new class named "M"
- Create a new attribute named "attM1" in the class M with type EInt
- Create a new attribute named "attB3" in the class B with type EInt
- Set abstract the class V
- Join the classes E and N with the name EN
- Create a new reference named M_Z with source class M, target class Z, containment false, min cardinality 1 and max cardinality *.
- Create a new reference named X_B with source class X, target class B, containment false, min cardinality 0 and max cardinality *.

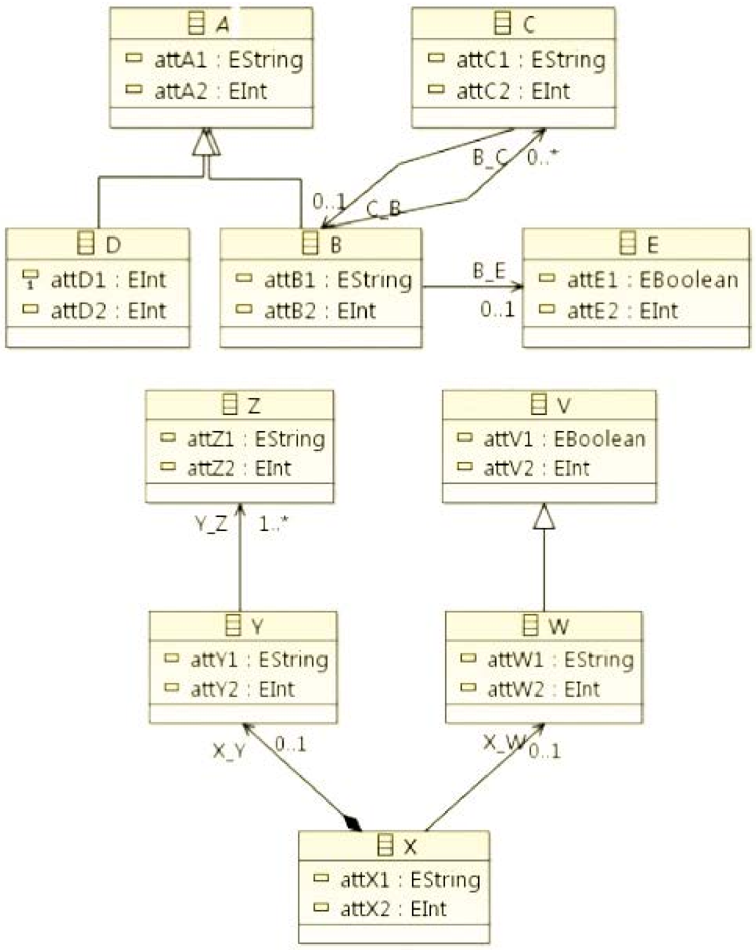

Figure 3. Imported metamodels

Source: own work.

- Divide the class X creating the class X1 with the attribute attX1 and the references X_Y and X_B; and the class X2 with the attribute attX1 and the references X_Y and X_W.

The listing presents with the source code for generating the composed metamodel following the requirements described.

1. import "inputMM1.ecore"
2. import "inputMM2.ecore"
3. export "outputMM"
4. newClass (N)
5. newAttribute (N, attN1, EInt)
6. newClass (M)
7. newAttribute (M.attM1, EInt)
8. newAttribute (B.attB3, EInt)
9. setAbstractClass (inputMM1.V)
10. joinClasses (EN, inputMM1.E, N)
11. newReference (M_Z, M, inputMM2.Z, false, 1, -1)
12. newReference (X_B, exampleMM2.X, exampleMM1.B, false, 0, -1)
13. divideClass (exampleMM2.X [X1, attX1, X_Y, X_B], [X2, attX1, X_Y, X_W])

As a result of the composition process, the composed metamodel generated is shown in the figure 4.

COMPOSITION ENGINE

The composition engine of this proposal executes the composition script sequentially. Once, the engine executes the import operations, it creates in dynamically memory the objects of each metamodel inside the correspondent package. Using the metamodels shown in the figure 3, and the previous example script, after executing the first, the second, and the third operations (lines 1, 2, and 3), the distribution of the elements in dynamic memory is presented in the figure 5. With these

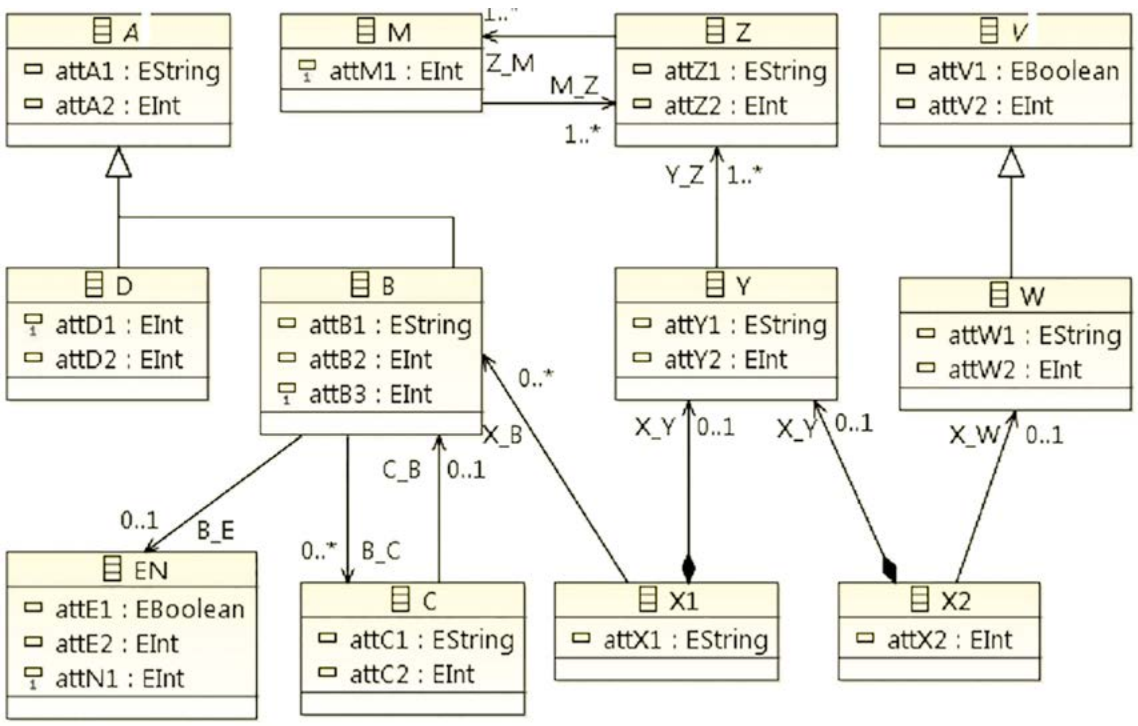

Figure 4. Output metamodel.

Source: own work.

operations the engine imports two support metamodels (MM_{sup-1}, MM_{sup-2}) and creates the composed metamodel (MM_0). The elements that belong to MM_{sup-1} and MM_{sup-2} are included in MM_0.

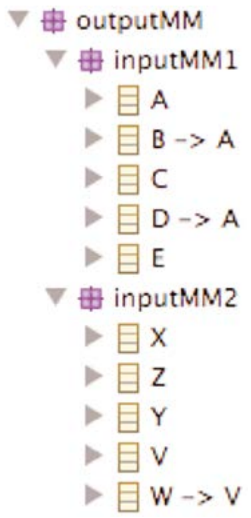

Figure 5. Distribution of the elements in dynamic memory.

Source: own work.

In addition, after the composition engine executes the sixth operation, the classes N and M are created in the generic package "outputMM". The distribution of the elements in dynamic memory after the seventh operation is presented in the figure 6.

Also, after the composition engine executes the tenth operation, the class EN is created in the generic package "outputMM". However, the classes involved in this operation that are E (that belongs to MM_{sup-1}) and N will be deleted from the correspondent packages. The distribution of the elements in dynamic memory after the tenth operation is presented in the figure 7.

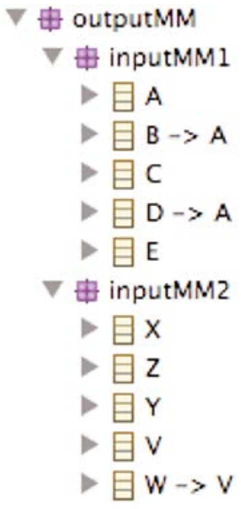

Figure 6. Distribution of the elements in dynamic memory.

Source: own work.

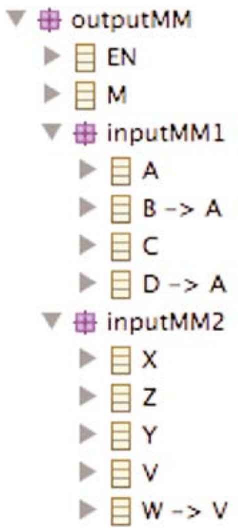

Figure 7. Distribution of the elements in dynamic memory.

Source: own work.

Finally, after the composition engine executes the thirteenth operation, the classes X1 and X2 are created in the generic package "outputMM". However, the class X (that belongs to MM_{sup-2}) will be deleted from the correspondent package. The distribution of the elements in dynamic memory after the thirteenth operation is presented in the figure 8.

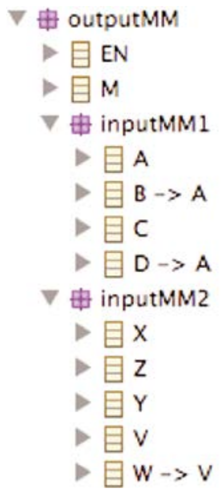

Figure 8. Distribution of the elements in dynamic memory.

Source: own work.

Once the composition engine executes the script, the classes from the import metamodels that have not been affected will be translated to the generic package "outputMM". Also the packages of the imported metamodels will be deleted. As a result, all elements in the composed metamodel will belong to the generic package.

In the case that the engine finds that one operation cannot be executed, the engine will report the mistake and the process will not continue. The reasons in which the process can fail are the follows:

- The import metamodel does not exist.
- The class, attribute, or reference required does not exist.
- In the case of creation of new elements; the class, attribute, or reference related already exist.
- After executing the script, there are duplicated classes.

CASE STUDY

In order to demonstrate the functionality of the language, a simple case is taken. In this case, it is taken a summarized metamodel of a bike and a summarized metamodel of a car.

Bike Metamodel

The figure 9 presents a summarized metamodel of the bike with the elements abstracted for this domain. This metamodel consist in the main parts of a

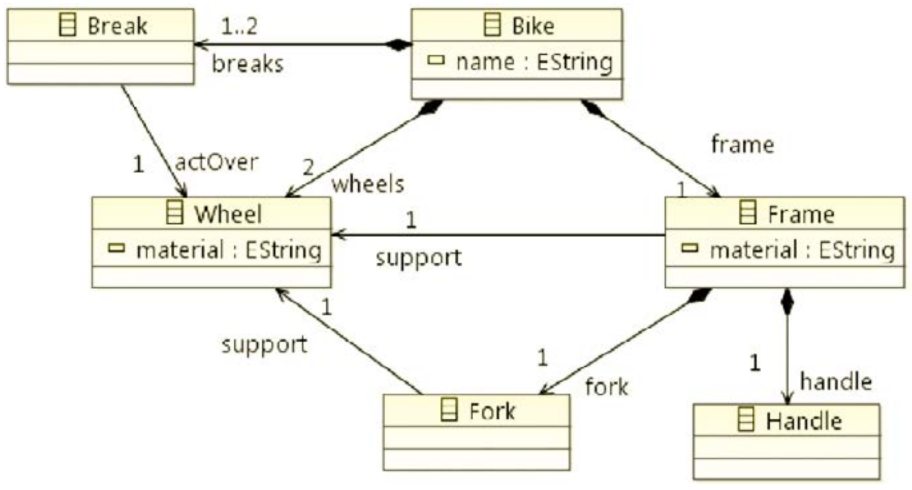

Figure 9. Bike metamodel.

Source: own work.

bike. This metamodel indicates that one bike has one frame, two wheels, and one or two breaks. Moreover, the frame has one handle and one fork, and supports just one wheel. Furthermore, the fork supports the other wheel. Finally, each break acts over one wheel.

Car Metamodel

The figure 10 presents a summarized metamodel of the car with the elements abstracted for this domain. This metamodel consist in the main parts of a car. This metamodel indicates that one car has one chassis, four wheels, four hydraulic breaks, and one engine. Moreover, the chassis has one body that has up to 5 doors, and supports four wheels. Finally, each break acts over one wheel.

Composed Metamodel

Based on the previous metamodels, the next script has been created in order to generate a composed metamodel related with a summarized motorcycle domain. The motorcycle has the majority of the components included in the bike metamodel; however, it requires more elements that can be provided by the car metamodel. The listing presents the composition script used for composing the metamodel.

1. import "bike.ecore"
2. import "car.ecore"
3. export "motorcycle"
4. renameClass (bike.Bike, "Motorcycle")
5. deleteClass (bike.Break)
6. deleteClass (car.Car)
7. deleteClass (car.Chassis)
8. deleteClass (car.Body)
9. deleteClass (car.Door)
10. deleteClass (car.HydraulicBreak.actOver)
11. newReference (hydraulicBreaks, bike.Motorcycle, car.HydraulicBreak, trae, 2, 2)
12. joinClasses (NewWheel, bike.Wheel, car.Wheel)
13. newReference (actOver, car.HydraulicBreak, bike.Wheel, trae, 1, 1)
14. newReference (engine, bike.Motorcycle, car.Engine, trae, 1, 1)
15. divideClass (bike.Frame, [FrameChassis, material, handle, fork], [Seat, material]
16. newReference (seat, bike.Motorcycle, Seat, true, 1, 1)
17. newReference (frame, bike.Motorcycle, FrameChassis, true, 1, 1)

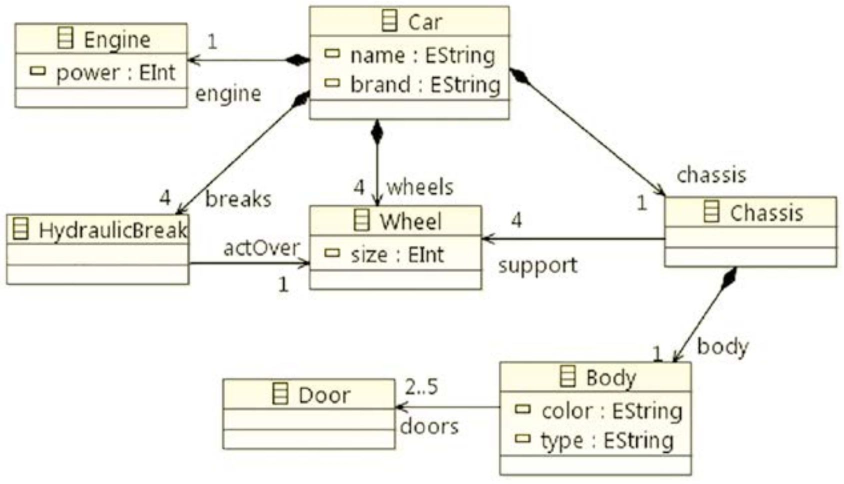

Figure 10. Car metamodel.

Source: own work.

The figure 11 presents a metamodel of the motorcycle generated by the composition engine after applying the previous script based on the composition DSL.

RELATED WORK

There are some approaches that already have treated the problem of metamodel composition. For instance, the work of Emerson *et al.*[5] presents a complete description of metamodels composition. Also, it presents a detailed characterization of different techniques (e.g., merge, refinement) providing a wide understanding regarding advantages and disadvantages for the composition in the MDE context. However, this work does not present a specific proposal for solving metamodels composition.

In addition, Karsai *et al.*6 presents a proposal focused in the reusability of metamodels in one specific domain of information. This proposal also includes a language that allows the manipulation of metamodels that abstracts subdomains of the desired domain of information. Thus, the composition process is not possible with metamodels of different domains. Finally, this proposal acts over diagrams based on UML, which can be taken as disadvantage due to the MDE context has became more relevant for academic and industrial communities.

Another related work is the proposal presented by (J. Oldevik, L. Kutvonen, and N. Alonistioti, 2005). In this work, the composition is achieved by the execution of several transformations using the transformation language Query View Transformation (QVT). This proposal consists in the creation of one framework that supports the execution of several transformations, where metamodels, which conform ECORE metamodel, are treated as models obtaining the desired behavior and results. This proposal has a disadvantage regarding with the level of knowledge of QVT because in this project, this language is focused in the transformation of one metamodel instead of the creation of on new metamodel as a result of the composition of several input metamodels.

Finally (R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh, 2007) present a similar proposal for composition because they have created a DSL for defining the way in which the composition would be achieved. This proposal also takes into account the strategies presented in this paper (i.e. matching, merging, and refining); however, they focus their results not in metamodels, but in

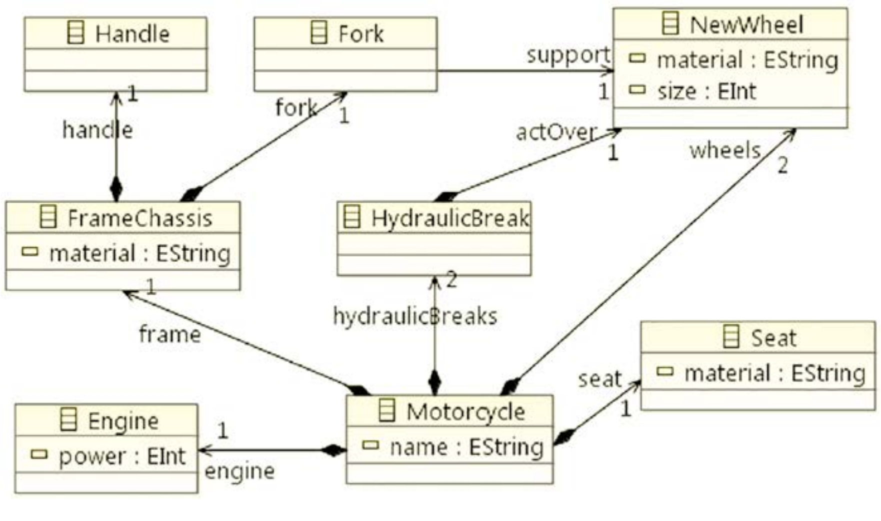

Figure 11. Motorcycle metamodell.

Source: own work.

UML models restricting the solution in one specifc domain. Also, the language created is based on constraints, which makes harder the declaration of simple composition tasks such us the refinement of existing elements (e.g., updating of attributes or relations).

CONCLUSIONS

A metamodel composition process, where metamodelers can adapt concepts abstracted in several existing metamodels, is possible. In this approach one DSL allows metamodelers define the creation of new elements for the composed metamodel; the way to adapt the elements existing in those input metamodels, and the elements created in the composition process; and the generation of the new composed metamodel.

An advantage of this approach is that metamodelers cannot perform illogical composition operations. Another advantage of this approach is based on the execution of the composition as a set of atomic operations over the input metamodels; each transition can use the modifications done in the previous operations. One more advantage is the creation of metamodels reducing the effort for metamodelers by getting the elements abstracted in existing metamodels.

The presented approach is simple, completeness and has high granularity, for each composition operation can be done independently and all of them cannot be decompose in smaller operations; as a result, the proposal is adequate to be used by metamodelers in order to create new abstractions through a metamodel based on existing metamodels.

ACKNOWLEDGEMENT

The author thanks the Universidad Distrital Francisco José de Caldas.

REFERENCES

A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti (2001). On metamodel composition. *Control Applications,. (CCA '01). Proceedings of the 2001 IEEE International Conference on*, pp. 756-760.

G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits (2004). Composition and cloning in modeling and meta-modeling. *Control Systems Technology, IEEE Transactions on,* vol. 12, pp. 263-278.

H. Florez (2012). Model Transformation Chains as Strategy for Software Development Projects. *The 3rd International Multi-Conference on Complexity, Informatics and Cybernetics: IMCIC 2012.*

J. Bézivin (2005). On the unification power of models. *Software and Systems Modeling,* vol. 4, pp. 171-188.

J. Henriksson, F. Heidenreich, J. Johannes, S. Zschaler, and U. Assmann (2008). Extending grammars and metamodels for reuse: the Reuseware approach. *Software, IET,* vol. 2, pp. 165-184.

J. Oldevik, L. Kutvonen, and N. Alonistioti (2005). Transformation Composition Modelling Framework. Distributed Applications and Interoperable Systems. vol. 3543, ed: Springer Berlin / Heidelberg, pp. 1135-1136.

M. Emerson and J. Sztipanovits (2006). Techniques for Metamodel Composition. *Computer Science and Information Systems Reports,* pp. 123-139.

M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth (2010). Language Evolution in Practice: The History of GMF. *Software Language Engineering*. vol. 5969, ed: Springer Berlin / Heidelberg, pp. 3-22.

P.A. Muller, F.; Fondement, B. Baudry, and B. Combemale (2009). Modeling modeling modeling. *Software and Systems Modeling,* pp. 1-13.

R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh (2007). Providing Support for Model Composition in Metamodels. *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pp. 253-253.