Electronic Communications of the EASST
Volume 21 (2009)

Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling (MPM 2009)

A practical approach to multi-modeling views composition

Andres Yie, Rubby Casallas, Dirk Deridder and Dennis Wagelaar

10 pages

Guest Editors: T. Levendovszky, L. Lengyel, G. Karsai, C. Hardebolle
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A practical approach to multi-modeling views composition

Andres Yie[13*], Rubby Casallas[2], Dirk Deridder[4†] and Dennis Wagelaar[5‡]

[1] a-yie@uniandes.edu.co
[2] rcasallas@uniandes.edu.co
Grupo de Construcción de Software
Universidad de los Andes, Bogota, Colombia

[3] ayiegarz@vub.ac.be
[4] dirk.deridder@vub.ac.be
[5] dennis.wagelaar@vub.ac.be
System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussel, Brussels, Belgium

Abstract: The use of several view models to specify a complex system is a common practice to provide the most appropriate abstractions to model its diverse concerns. When several view models are used to specify a system, it is necessary to compose them to generate the application. When the view models are expressed in different Domain Specific Modeling Languages a problem arises because a *heterogeneous composition* is required. A possible approach to avoid a heterogeneous composition is to transform the diverse models into low-level models using a common low-level modeling language as target. Therefore, when all the view models are transformed in low-level models specified with a common language, it is possible to apply a *homogeneous composition* to obtain the final application. However, it is necessary to identify the elements to compose in the low-level. In this paper, we present an automatic mechanism to identify which elements will be composed. This mechanism is based on defining correspondence relationships between the high-level view models and automatically deriving new correspondence between the generated low-level models.

Keywords: Model Driven Engineering, Model transformation, Model composition, Multi-paradigm modeling

1 Introduction

The use of several models to specify a complex system is a common practice in software engineering. The main objective is to provide the most appropriate abstractions to specify the diverse concerns involved (e.g., UML Class diagrams to specify structure, UML State charts to specify behavior); each of these models is called a view model.

* This author is sponsored by the Caramelos project (VLIR)
† This author is sponsored by the MoVES project (IAP, Belgian Science Policy)
‡ This author is sponsored by the VariBru project (ISRIB)

Some approaches use appropriate Domain Specific Modeling Languages (*DSMLs*) trying to reduce the gap between the problem and the solution domains. The use of several DSMLs helps experts to express the system specifications. Having the correct DSML for each concern reduces the accidental complexities introduced while specifying it using a unsuitable language. For instance, a web application can be specified by means of diverse view models such as data, navigation and presentation models [CD08]. Each view model is expressed with an appropriate DSML, for example a Presentation DSML to express the presentation view model using concepts of the presentation domain, such as *dialogs*, *fields*, *buttons*, etc.

One of the main goals of Model Driven Engineering (*MDE*) is to automatically generate applications from models. When several view models are used to specify a system, it is necessary to compose them to generate the application. When the models are expressed in different DSMLs a problem arises because a *heterogeneous composition* is required (e.g., composition of a business entity from a business DSML and a secured resource from the security DSML). This means that it is necessary to have different composition mechanism for every pair of DSMLs. A possible approach to avoid a heterogeneous composition is to transform the diverse models into low-level models using a common low-level modeling language. Moreover, this common low-level modeling language can be a General-Purpose Language (GPL) to implement the application. Therefore, when all the view models are transformed in low-level models specified with a common language, it is possible to apply a *homogeneous composition* (e.g., composition of two Classes) for the complete set of models. A similar strategy is presented in [Van00], where a common low-level formalism is used to bring together several specification defined in different formalisms.

Additionally, in order to compose two models it is necessary to identify *what* elements will be composed. Sometimes, it is enough to use the signature of the corresponding elements, but when it is required more flexibility, it is necessary to establish correspondence relationships between the elements to compose [BBD+06]. These relationships can be defined in a *Correspondence Model (CM)* that contains the links between elements to compose.

However, when the different high-level models are transformed into a common low-level language it is necessary to derive the correspondence relationships between the generated low-level models. These correspondence relationships identify the elements that actually will be composed in the low-level models.

In this paper we present a practical approach to compose multi-modeling views. This approach uses transformation chains (TC) [1] to bring two high-level models specified in different DSML into a low-level common language. Additionally, the correspondence links defined between the high-level models are used to derive a low-level correspondence links which identify which are the actual low-level elements to compose. In order to obtain the low-level CM our approach generates traces to identify the target sets of elements in the generated low-level models. Using these traces it is possible to identify if a target set is generated from an element that belongs to a correspondence relationship. Additionally, we use a *Derivation Model (DM)* which contains constraints that allows us to find the match between the generated elements, discarding *incompatible* ones. Finally, a common composition mechanism is used to integrate the elements of

[1] A transformation chain (TC) is a sequence of transformation steps that converts the high-level model, which is rooted in the problem domain, into a low-level model, which is rooted in the solution domain.

each low-level model.

The structure of the paper is as follows: the next section presents an example that is used to illustrate the proposed solution. Section 3 describes the problem that we tackle. In Section 4 an overview of our solution is presented. Section 5 details how we derive a low-level CM. This low-level CM is used in Section 6 to execute the actual composition. Finally, the related works are presented in Section 7, and Section 8 presents the conclusions of our work.

2 Illustrative example

We use as case study a simplification of an MDE implementation where web applications based on components and their security are modeled using two independent view models. These two high-level models are specified using two different DSML. The main objective of the first DSML is to specify the core functionality model using business concepts. The objective of the second DSML is to specify the authorization policies of the application. In this section both DSMLs are presented.

2.1 Business modeling language

The three main elements in a DSML are: 1) The *abstract syntax* where the different terms of the language are defined. Usually the abstract syntax is defined in a metamodel. 2) the *concrete syntax* contains the textual or visual representation of a language. 3) The *semantics* of the language that is the actual meaning of each concept and possible sentence in the language.

As it was stated before, the first language is used to specify the main functionality of an application using concepts such as *Entity*, *Attributes* and *Services*. Figure 1(a) presents an extract of the metamodel of the Business modeling language. The main concept of this metamodel is *Entity*, which represents business entities of the application, such as *Project* or *Item*. An *Entity* has *Attributes*, *Relationships* and *Services*. Using this language we modeled a web based change management application called *Changeset*. The business view model of *Changeset* represents a software project with items that belong to the project and changes that could be requested for them. *Project, Item* and *ChangeRequest* are concepts conform to the *Entity* concept. In Figure 1(b) an extract of the business model of *Changeset* is presented[2].

2.2 Security modeling language

We define a Security language using the SecureUML metamodel defined by Lodderstedt et al. in [LBD02], which in turn is based on the Role Based Access Control (RBAC) model. Figure 2 shows the metamodel, where the main elements are *Users* and *Groups*. A *Role* can perform a set of *Actions* on a *Resource* and it can be assigned to *Users* or *Groups*. The *Actions* are grouped as a *Permission*. *Resource* and *Action* are abstract concepts and it is necessary to extend them to include the resources and the actions to be protected. In our case study, we extend the concept *Resource* with *EntityResource*, *AttributeResource* and *ServiceResource*. Additionally, the concept *Action* is extended to define every action that can be protected in the extended resources..

[2] We use the UML/Stereotypes concrete syntax as a common representation for the presented models.

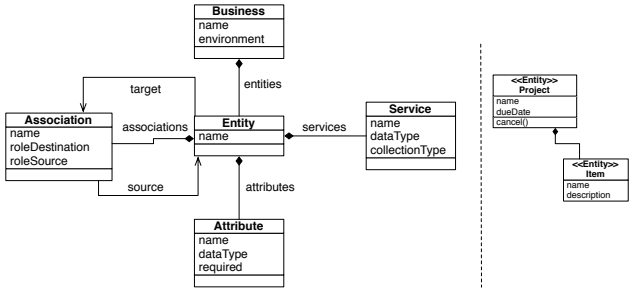

Figure 1: Business Metamodel (MM_{bus}) and Model (M_{bus})

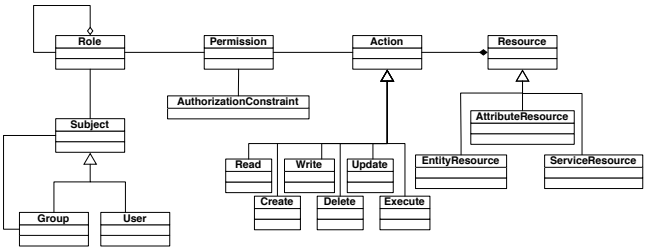

Figure 2: High-level Security Metamodel (MM_{sec})

For instance, the security model *(M_{sec})* for *Changeset* has two roles *User* and *Manager*. The role *User* has a permission over *Read* actions on the EntityResource *Project*. This means that a *User* can read the information about a *Project*, but he cannot change it. The role *Manager* inherits *User* permissions and adds *Create*, *Update*, and *Delete* actions on *Project* and *Execute* action on the *Project.cancel()* ServiceResource. These *Permissions* mean that a *Manager* can read, and change the information of a *Project*.

3 Problem: Composing multi-modeling views

When a complex system is specified using several view models, it is necessary to compose them to obtain the required application. Nevertheless, before performing the composition it is necessary to identify the elements that will be composed. In order to identify those elements we use a CM. As defined in [BBD+06] a CM is a model that explicitly describes the relationships between elements of different models. Furthermore, the CM is the definition of *what* to compose, which elements, described in the models, will be composed. At the high-level of abstraction, the CM model is constructed manually by the modeler.

3.1 High-level Correspondence Model ($CM_{high-level}$)

We align the two high-level models using a CM which relates the elements to be composed. For example, in figure 3 the application model (M_{bus}) contains the entity *Project* and the security

model (M_{sec}) contains the resource *Project* that needs to be protected. The CM between both models $CM_{high-level}$ represented in the figure as black lines with circles in their ends, contains a relationship with links to the entity *Project* and the resource *Project*. Additionally, the Attribute *dueDate* is related in the CM to the AttributeResource *dueDate*. The modeler creates these correspondence links because he knows the meaning of the relationships between elements.

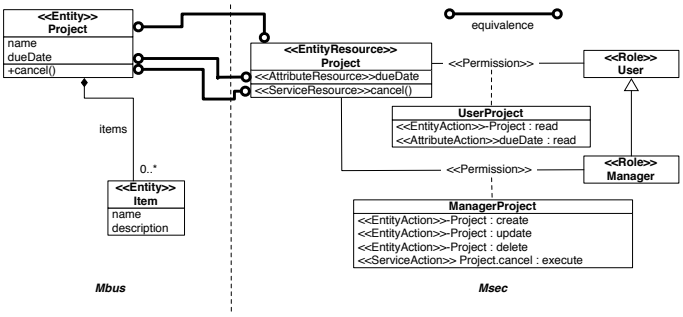

Figure 3: High-level View Models.

3.2 Heterogeneous composition

A heterogeneous composition is the integration of two models expressed in two different languages. This means that is necessary to define the compositional semantic for every pair of concepts in both languages. In our case, to compose the business view model and the security view model will be a heterogeneous composition. For instance, if we have the concept *Entity* that belongs to a Business modeling language, and the concept *Resource* that belongs to a Security modeling language, it is necessary to define what it means to compose them, e.g., if every *Service* and *Attribute* of the *Entity* will be protected too. Similarly, if a third language is added, it will be necessary to define a composition semantic for each pair of languages, increasing to the complexity of implementing a multi-view solution.

Our strategy is to avoid the heterogeneous composition transforming each view model into a low-level model defined using a common modeling language. After the composition we perform a composition between homogeneous concepts, for instance, to compose two *Classes*. Therefore, if the complete set of view-models is transformed in to a common low-level language, a homogeneous composition can be applied. However, a new problem appears: how to identify the element to compose in the low-level models. In other words, how to derivate the CM between the low-level models.

4 Approach overview

The overall approach is to transform both high-level models into low-level models that conform to the same existing metamodel (e,g,. Java metamodel), or conform to an extension of it. We align both high-level models by using a CM, which needs to be propagated through the complete transformation chains. The main challenge is to define a mechanism to automatically derive the

new correspondence relationships, having in mind that the TC increments the complexity of the models by adding elements at each step.

To derive a low-level CM it is necessary to trace back the elements of the low-level models and to check if they come from pairs of related elements in the high-level. With a trace model (*TM*) [ANRS06] we determine the elements in both low-level models that come from a couple of related elements in the high-level. For instance, an *Attribute* in the business view model is transformed in the low-level model into: an *Attribute*, a *GetterMethod* and a *SetterMethod*. In the security model a *ResourceAttribute* with a *ReadPermission* is transformed in the low-level security model into: a private *Attribute* and an annotated *ReadMethod*. Therefore, we have to trace back all these low-level elements and verify that the high-level source element (e.g., *Attribute*) from which they originate, is related with a correspondence relationship to the high-level concern-specific element (e.g., *ResourceAttribute*).

Once, we establish which elements in the low-level models came from a pair of correspondent elements in the high-level models, we have to relate them by identifying the *correct match* for each one. For instance, a *GetterMethod* (in the low-level application model) can be related to a *ReadMethod* (in the low-level security model) but not to a *WriteMethod*. The modeler has to specify constraints, to avoid erroneous correspondences. A constraint is a relationship between two metaclasses that defines if the correspondence link between concepts that conform to them can be established or not. In our solution this set of constraints is called a Derivation Model (*DM*). This approach is presented too in the context of transformation chain evolution [YCWD09].

Figure 4 presents the general schema of our approach. In the left, the business view model is transformed into a Java model (MM_{bus}, M_{bus}, MM_{java}, M_{java}, T_1)[3]. In the right the security view model is transformed into a Java model too (MM_{sec}, M_{sec}, $MM_{sec-java}$, $M_{sec-java}$, T_2). $CM_{high-level}$ is the high-level correspondence model that aligns the two high-level models. TM_A and TM_S are the trace models that relate the high-level models with the low-level models. The DM relates the low-level metamodels with constrains between their metaclasses. The DM is used to generate the transformation T_3, that uses the trace models and the $CM_{high-level}$ to generate the $CM_{low-level}$.

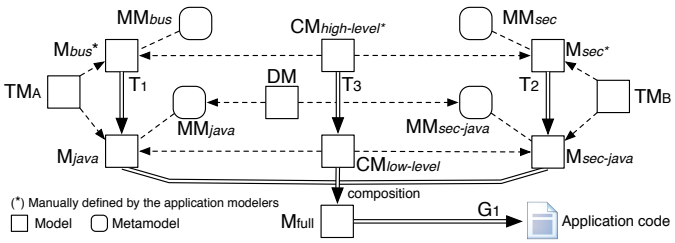

Figure 4: General Schema

Finally, the low-level models are composed and transformed into code by the original model-to-text transformation (G_1).

[3] *MM* = Metamodel, *M* = Model, *T* = Transformation Chain

5 Derivation of Correspondence Model

To implement our strategy, low-level correspondence relationships have to be derived automatically. The transformation (T_3), produces the low-level CM.

5.1 Derivation

Two elements a' and b', from application and security models respectively, will have a correspondence relationship if: 1) There is a CM relationship at the higher level between a and b, where a' was produced from a by T_1, and b' was produced from b by T_2. 2) The metaclasses ma' and mb' where a' conforms to ma' and b' conforms to mb', allow for correspondence relationship between their instances. Intuitively, the first condition establishes that elements a' and b' trace back to a pair of elements that have a high-level correspondence relationship between them. The second condition means that the metaclasses ma' and mb' are the same metaclass or extensions of the same one. Therefore, it is permitted to define correspondence links between their instances and finally to compose them. If both conditions are satisfied for an element a' and b', T_3 will produce a correspondence link between a' and b'.

In figure 5, a correspondence link is created between the GetterMethod *getDueDate* in M_{java} and the annotated ReadMethod *readDueDate* in $M_{sec-java}$ because they satisfy both conditions. First, *getDueDate* traces back to the Attribute *dueDate* in M_{bus} when T_1, transformed it and *readDueDate* trace back to the AttributeResource *dueDate* in M_{sec} when T_2 transformed it. Additionally, the Attribute *dueDate* and the AttributeResource *dueDate* were related by the high-level CM ($CM_{high-level}$). Second, the GetterMethod and ReadMethod the metaclasses are allowed to be related. However, is not possible to create a correspondence link between the SetterMethod *setDueDate* and the annotated ReadMethod *readDueDate* because these metaclasses do not have a constraint that allows to relate them.

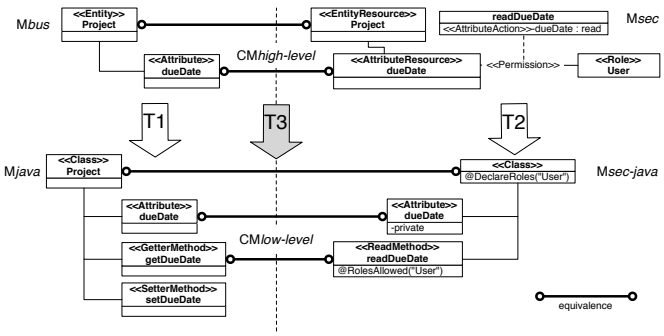

Figure 5: Detailed schema

5.2 Traceability

When T_1 is applied to the Attribute *dueDate*, it is transformed into the Attribute *dueDate*, the GetterMethod *getDueDate* and the SetterMethod *setDueDate*. In order to make this information available to T_3, we generate trace links between output elements and input elements. The same

happens in the T_2 side, T_3 needs to know if the ReadMethod traces back to a related AttributeResource. Once T_1 and T_2 are executed, two tracing models are generated (TM_A and TM_S). With these two tracing links, T_3 can find the elements in both lower-level models that trace back to the pair of related elements in both higher-level models.

We generate a tracing model when each transformation (i.e., T_1 and T_2) is executed. This model conforms to a Traceability Metamodel and it has links between every source element and its target elements.

5.3 Derivation Model

The modeler has to define a Derivation Model (*DM*) to make explicit if the instances of two metaclasses are allowed or not to be related by a correspondence link. In the same way, the modeler has to decide about the propagation of the compatibility relationships. If two metaclasses are compatible, are their submetaclasses compatible too? Are the composites compatibles too?

We have defined different types of constraints in the Derivation Metamodel. These types are: *Inheritable constraint* (to allow submetaclasses), *Final constraint* (to reject submetaclasses), *Incompatible constraint* (to explicitly reject two metaclasses, and *Composition constraint* (to allow composites). Due to space restrictions the details of the semantics of these constraints are out of the scope of this paper.

5.4 Generating the CM Transformation

T_3 receives as input the high-level CM, but T_3 cannot use the DM as input. The DM is defined at metamodel level and a transformation rule only receives models as inputs. Therefore, it is necessary to express the constraints as part of the transformation rules. For this reason we, use a High-Order Transformation (HOT) that receives the DM as input and produces T_3.

In our case study, the GetterMethod *getDueDate* in the application side can be connected to the ReadMethod *readDueDate*, because the metaclasses GetterMethod and ReadMethod are compatible. As it was presented before, there is a constraint that allows these two elements to be connected. The case of GetterMethod and WriteMethod is clear that they are incompatible metaclasses and a correspondence link cannot be defined between their instances. In this way it is possible to relate only the compatible pairs of elements and not every generated element in each side.

6 Composition

The final step is to compose both low-level models to obtain a complete low-level model of the application. In this composition, the generated CM ($CM_{low-level}$) is an essential input. This model has the information of *what* will be composed. In *Changeset*, the Classes in the application low-level model (M_{java}) will be composed with the annotated Classes in the security low-level model ($M_{sec-java}$), the Attributes in (M_{java}) with the private Attributes in ($M_{sec-java}$), and the Methods in (M_{java}) with the annotated methods in ($M_{sec-java}$). By using the correspondence links it is possible to identify every pair of elements to be composed. To do the actual composition we use a mechanism based on the *UML Package Merge* [DDZ08].

7 Related work

In our approach we integrate several of the ideas presented in the following works. These ideas help us to model the application and its concerns in independent models and delay the composition to the lower-level of abstraction where a heterogeneous composition is performed.

Following the multi-modeling principle, Cicchetti and Di Ruscio present in [CD08] a proposal for modeling a web application using three independent models: a data model, a composition model, and a navigation model. A weaving model (correspondence model) is defined between the data model and the composition model, and an additional weaving is defined between the composition model and the navigation model. These three models are composed using the CMs into a high-level model. This high-level model conforms to a platform independent web metamodel, such as WebML or Webile. These web modeling languages try to integrate all the required concepts for web application modeling in a common metamodel. However, this research does not try to generate the final application code, only the complete high-level model. Nevertheless, it is not always possible to find a common high-level metamodel that includes all the required concepts of the different concerns to model an application. In our research, we want to bring the independent models to the lowest-level taking advantage of the reduced semantic gap between the metamodels at this level and the models richer in implementation details.

In [CD06], Cibran et al. present an approach to define business rules on an application as aspects. The business rules are modeled using a DSML. The relationships between the business rules and the application are defined using a connection DSML. This DSML abstracts the different patterns of how the business rules are connected with the application code. This connection language can be seen as a CM that relates two different models of the application. Both models are transformed to low-level models. The connections are also transformed and aspect code is generated from them. In our approach, using the DM and the HOT we automate the generation of the transformation that creates the relationships between the low-level models.

8 Conclusions

Our approach facilitates the modeling of different concerns using separated view models each one close to the problem domain. The different view models are aligned using correspondence relationships between their elements. These correspondence relationships explicitly capture the overlapping and dependencies among their elements. Additionally, our approach offers an automatic derivation mechanism to identify the elements to compose in the low-level models based on the correspondence relationships defined between the high-level models. With this mechanism, it is possible to maintain both models aligned from the high-level until the lowest-level through the transformation chain. This is the main difference of our work with other approaches where these relationships are only defined as an input, but not maintained during the transformation chain. As a result of delaying the actual composition to the lowest-level where all the models conform to the same metamodel or to an extension of it, it is possible to perform a homogeneous composition using a single composition mechanism.

We implemented the derivation mechanism and the composition mechanism using the Atlas Modeling Language (ATL) [JK06] and in our current work we are exploring the modeling of

an application in three or more models and bringing theses models to the lower level, where they will be composed. This is a complex problem, because the interactions among the different modeled concerns can generate several inconsistencies.

Bibliography

[ANRS06] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, Y. Shaham-Gafni. Model traceability. *IBM Systems Journal* 45(3):515–526, 2006.

[BBD+06] J. Bézivin, S. Bouzitouna, M. Del Fabro, M. P. Gervais, F. Jouault, D. Kolovos, I. Kurtev, R. F. Paige. A Canonical Scheme for Model Composition. *Model Driven Architecture Foundations and Applications*, pp. 346–360, 2006.
doi:10.1007/11787044_26
http://dx.doi.org/10.1007/11787044_26

[CD06] M. Cibran, M. D'Hondt. A Slice of MDE with AOP: Transforming High-Level Business Rules to Aspects. *Proceedings of the 9th International Conference on MoDELS/UML*, 2006.

[CD08] A. Cicchetti, D. Di Ruscio. Decoupling web application concerns through weaving operations. *Science of Computer Programming* 70(1):62–86, 2008.
doi:http://dx.doi.org/10.1016/j.scico.2007.10.002

[DDZ08] J. Dingel, Z. Diskin, A. Zito. Understanding and improving UML package merge. *Software and Systems Modeling* 7(4):443–467, October 2008.
doi:10.1007/s10270-007-0073-9
http://dx.doi.org/10.1007/s10270-007-0073-9

[JK06] F. Jouault, I. Kurtev. On the architectural alignment of ATL and QVT. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*. Pp. 1188–1195. ACM, New York, NY, USA, 2006.
doi:http://doi.acm.org/10.1145/1141277.1141561

[LBD02] T. Lodderstedt, D. Basin, J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. *UML 2002 - The Unified Modeling Language : 5th International Conference, Dresden, Germany, September 30 - October 4, 2002. Proceedings*, pp. 426–441, 2002.
http://www.springerlink.com/content/a82dhdta602g43r5

[Van00] H. Vangheluwe. DEVS as a common denominator for multi-formalism hybrid systemsmodelling. *IEEE International Symposium on Computer-Aided Control System Design, 2000. CACSD 2000*, pp. 129–134, 2000.

[YCWD09] A. Yie, R. Casallas, D. Wagelaar, D. Deridder. An approach for evolving transformation chains. *MoDELS '09: Proceedings of the twelfth ACM/IEEE international conference on Model Driven Engineering Languages and Systems*, pp. 551–555, Oct 2009.