D3.1-Model Views Conceptual Approach

Model Virtualization

	NAME	PARTNER	DATE
WRITTEN BY	CLASEN C.	ATLANMOD	04/02/2011
	KLING W.	ATLANMOD	04/02/2011
REVIEWED BY			

RECORD OF REVISIONS

ISSUE	DATE	EFFECT ON		REASONS FOR REVISION
		PAGE	PARA	
1.0	02/11/2010			Document creation

TABLE OF CONTENTS

TABLE OF APPLICABLE DOCUMENTS

N°	TITLE	REFERENCE	ISSUE	DATE	SOURCE	
					SIGLUM	NAME
A1						
A2						
A3						
A4						

TABLE OF REFERENCED DOCUMENTS

N°	TITLE	REFERENCE	ISSUE
R1	Galaxy glossary		
R2			
R3			
R4			

ACRONYMS AND DEFINITIONS

Except if explicitly stated otherwise the definition of all terms and acronyms provided in [R1] is applicable in this document. If any, additional and/or specific definitions applicable only in this document are listed in the two tables below.

Acronymes

ACRONYM	DESCRIPTION

Definitions

TERMS	DESCRIPTION

1. INTRODUCTION

Modeling complex software systems involves a large number of heterogeneous models focusing on different aspects of the system at different abstraction levels. These models must be later combined in order to provide to each to each designer/developer of the system the specific view they need to perform their tasks. This problem is tightly connected to model composition, in which two or more models have their information captured to generate one single composed model.

However, current approaches present some important limitations concerning efficiency (due to the copying mechanism of model elements into the composed model), interoperability (when the composed model has a different *nature* than the contributing ones and needs to be manipulated using specialized tools) and/or synchronization (by failing to propagate changes from the generated model to the contributing ones, or the other way round) issues.

This document aims to describe a new model composition solution based on a model virtualization mechanism. Instead of generating a new composed model our approach generates a virtual model that provides the *illusion* to users (and tools) of working with a generated composed model while in fact, all model access and manipulation requests on the (virtual) composed model are directly translated to operations on the contributing models in a transparent way.

2. ON THE NEED FOR VIRTUAL MODELS

2.1 INTRODUCTION

Complexity of nowadays software systems is rapidly increasing and with this the difficulty to comprehend, develop and maintain them. To handle this complexity, software systems are usually represented by a series of models. Each model aims at representing a different aspect of the system at a different abstraction level. Models can be used for different activities: forward engineering, reverse engineering or even as run-time components of the system.

In either case, models must be combined to generate the most adequate view of the system for each user working with it, depending on his/her role (both to show only the relevant information to that user and to avoid errors due to improper manipulation of other aspects of the system in which the user is not involved). This is a complex and challenging problem due to the heterogeneity of the models and to the fact that most times several kinds of relationships exist between them, e.g., elements contained in one model may refine, extend or depend on other elements in a different model (e.g. a table in a relational database model refines a class in a UML conceptual model).

Several approaches for model composition have been developed so far. Basically, they all propose to generate a completely new composed model from a set of input models but differ on the language/technique used to define how to select and combine the elements from the source models that will be copied to the composed model. Fig. 1(a) illustrates the idea: models A and B are the input to a model composition mechanism that, accordingly to its composition rules, processes A and B and generates the composed model. Unfortunately, current approaches present some important limitations (linked to the fact that the composed model is generated as a separated new model by copying and merging pieces of information from the source models) regarding the performance (due to the time required to copy the model elements into the composed model), synchronization (due to the lack of propagation of changes from the generated model to the contributing ones or the other way round) or interoperability (sometimes the composed model has a different nature than the contributing ones and need to be manipulated using specialized tools) of the approach.

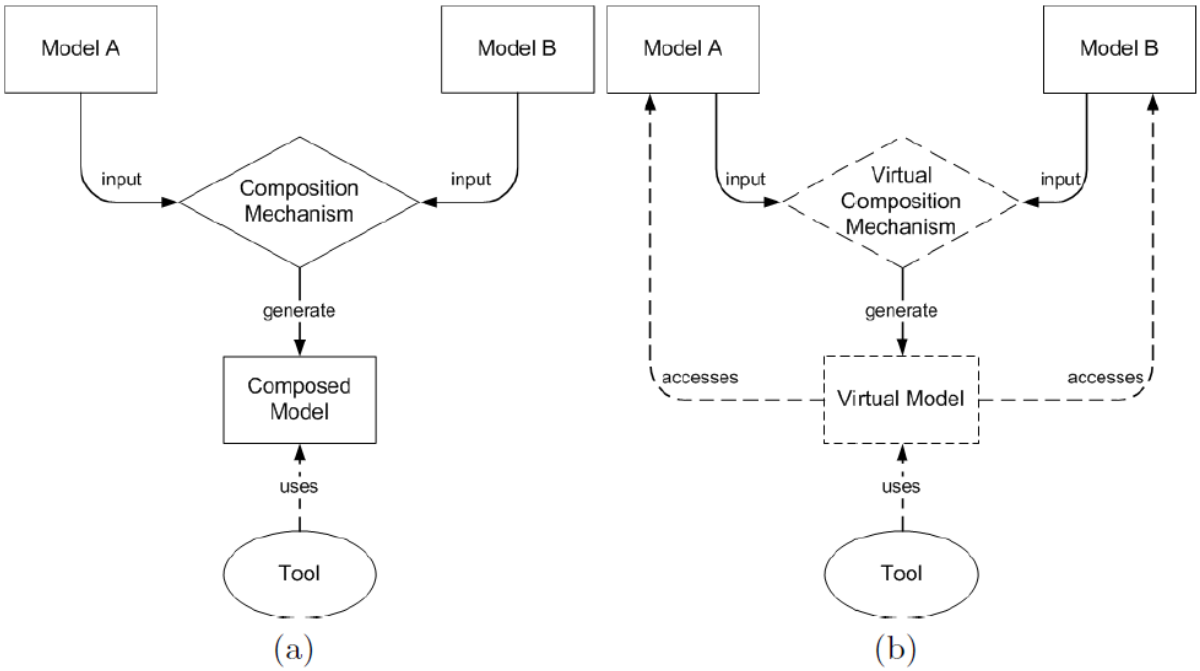

Figure 1: (a) Standard Model Composition; (b) Virtual Model Composition

This work has the purpose of creating a new model composition solution based on a model virtualization mechanism. Instead of generating a new composed model our approach generates a virtual model that provides the illusion to users (and tools) of working with a generated composed model while in fact, all model access and manipulation requests on the (virtual) composed model are directly translated to operations on the contributing models in a transparent way. This delegation avoids the problems mentioned above. Our solution is designed as a refinement of standard Model Access APIs, allowing our virtual model to be directly manipulated by current tools exactly in the same way as any other model.

2.2 MODEL COMPOSITION

Model composition is an emerging subfield in the area of Model Driven Engineering that has been studied from different perspectives such as aspect-oriented modeling[5], database schema integration[1], or model transformation[10]. In its simplest form, model composition is a modeling process that combines two or more input models. in order to generate a single output model that gathers the contents present in the input models[17]. The necessity of combining models come from the fact that systems are usually described by means of a large set of models, each one concerning different aspects of the system. Therefore, combining elements spread across different models allows:

1. to analyze a system from different viewpoints;
2. to obtain a cross-domain view of a set of models;
3. to manipulate heterogeneous information in an integrated manner;
4. to check the consistency between a set of models;
5. to establish relationships between different views of the system.

A formal definition on model composition and its semantics may be found in [9]. Nevertheless we provide here definitions on the most important elements involved in a composition process. We will use this terminology throughout the paper:

Definition 1. *A **Contributing Model** is a model used as input in a model composition process. Its elements are named contributing elements.*

Definition 2. *A **Composed Model** is the output of a model composition process, generated from a set of contributing models according to the rules specified in a model composition operator* $\otimes$*. Its elements are named composed elements.*

Definition 3. *An **Inter-Model Link Set** is a set of predefined links between different contributing models (describing relationships between their elements) that can be used as additional information in a composition process.*

Definition 4. *A **Model Composition Operator*** $\otimes$ *is a function that receives a set of contributing models as input and produces a composed model as output such that* $\otimes$*:* M x M → M*, where* M *is the universe of models.*

As seen from the previous definitions, the key element in a model composition process is the composition operator. It has mainly two tasks: (i) defining the semantics of the composition, and (ii) defining what we call the nature of the composed model. The first task is related to how the contributing models and its elements are combined to generate the composed model. For instance, the composition operator could simply define that the composed model is the union of all elements, or discard those satisfying a given condition, apply a partial merge of similar elements (where the similarity could be computed, e.g. by name matching, or predefined in a link set used as additional input for the operator),...

Our mechanism is independent of how this first task is accomplished. We focus on the second one: depending on the model composition operator applied, a composed model may have different natures. Typically, the selected elements of the contributing models are copied (i.e. cloned) into the composed one. In this case, the composed model is a concrete model, with exactly the same nature of the contributing ones (and completely disconnected from them once the creation process has finished). This is not the only possibility. As we will see, we propose an approach in which the composed model has a virtual nature.

2.3 QUALITY INDICATORS IN MODEL COMPOSITION

Depending on the kind of composition operator provided, the composition process will showcase a different set of functional and non-functional properties. We believe that the six most important ones are the following:

1. *Creation time*: time required to create the composed model;
2. *Manipulation time*: time required to perform read and write operations in a composed model;
3. *Memory usage*: memory requirements when performing a model composition. Especially important when dealing with large-size models;
4. *Interoperability*: capability of a composed model to be handled in the same manner a normal model would (i.e. this implies that we do not require special tools for manipulating the composed model);
5. *Modification capabilities*: the set of operations that can be applied on a composed model (e.g. read-only, establishing inter-model links, etc.);

6. *Synchronization*: automatic propagation of changes between composed and contributing models, ensuring their consistency.

These properties will be used when evaluating the existing approaches and when discussing the benefits of our alternative in the next sections.

2.4 STATE OF THE ART

A considerable number of works on model composition can be found in the literature (though there is not yet a consensus on its formal definition and terminology). In what follows we summarize the main "families" of approaches in the area.

Several works such as [17], [21] focus the formal semantics of model composition and discuss the possible types of composition operators (e.g., merge, expand, diff, etc.). These differences can be used to compare different model composition solutions [3], [2].

Other approaches as [8], [18] and [20] target the automatic composition of models proposing algorithms to identify, select and combine elements from contributing models. Composition frameworks, as in IBM Rational Software Architect[14], CCBM[16] or languages, as the Epsilon Merging Language[11] may help during the composition process by, for instance, facilitating the identification of the elements to merge or helping to compare the contributing models before deciding the elements to merge [12].

Other solutions apply model composition to target a specific kind of problems, as in [6]. Research has been also when considering the composition of a specific type of models, like ADLs[19], Statecharts[15] or UML[22][4].

What the vast majority of the proposals for composing models have in common is the fact that most of them completely generate a new (composed) model after the composition process (an exception is [13] but it focuses on *decorating* a single model with new information and not on composing models), completely discarding the contributing models once the new one was generated. Another common approach was not to generate a new model, but rather use one of them as a pivot that will receive the contents from the other. In a way or another we believe that all these approaches share one or more of the following three main problems:

1. Approaches that generate a new composed model by copying elements from source models do not scale well. The creation of the composed model may take too long and, more severely, the memory usage could be a serious bottleneck during the creation process because of the need of having available at the same time both the new instances of the composed model and the existing instances of the contributing models;

2. Approaches that do not provide propagation mechanisms to ensure the consistency between the contributing and the composede models. If synchronization is lost, tool using the composed model will present users with an outdated version of the model;

3. Approaches that generate a composed model with a different format than that expected by standard modeling tools. Visualizing and/or manipulating the composed model requires the use of a different API or toolset.

These problems hamper the usefulness of these methods as the basis of a model composition process. Next section presents our alternative model composition mechanism to overcome these limitations. Note that our alternative mechanism changes the way the generated model is built but not how to define the composition rules that specify how to select and merge the elements of the

contributing models. Our method is agnostic with respect to this. Any existing method could be used for this purpose.

3. MODEL VIRTUALIZATION (CONCEPTUAL)

In this section we present the conceptual idea of our solution for model composition in order to overcome the problems mentioned before. The main difference with relation to previous works is the role contributing models play in the composition process. In our solution the main idea is to consider the contributing models not only as simple input for the generation of the composed model, discarded once the composed model has been created, but rather as core elements during its whole lifecycle. This is achieved by adding virtualization techniques to the composition process that integrate the contributing models within the composed model.

3.1 THE NOTION OF A VIRTUAL MODEL

The main innovation of our solution is the virtualization of the composed model. In our approach, the composed model is a virtual model. They key difference between a virtual model and a concrete model is that a virtual model is made of virtual elements, which are proxies to actual elements contained in other models (usually concrete ones but we could also have a composition of virtual models). Virtual model elements are perceived and manipulated by a tool in the same manner as a normal model would, but in fact the actual elements being accessed (through the proxies in the virtual model) are the elements contained in the contributing models from which the virtual model was generated.

> **Definition 5.** *A **Concrete Model** is a model whose model elements hold concrete data. Its elements are named concrete elements.*
>
> **Definition 6.** *A **Virtual Model** is a model whose (virtual) model elements are proxies to elements contained in other models. A virtual model delegates the access to its elements to the models it references. Its elements are named virtual elements.*
>
> **Definition 7.** *A **Virtual Reference** is a reference in a virtual model that links two model elements contained in different concrete models.*
>
> **Definition 8.** *A **Virtual Composition Operator** $\oplus$ is a model composition operator that produces a virtual element as output, such that* $\oplus : M \times M \rightarrow VM$, *where* VM *is the universe of virtual models.* $\oplus$ *does not duplicate elements from contributing models; it creates virtual elements (i.e. proxies) to them.*

To make this idea clearer — and denoting the universe of models as *M*, the universe of concrete models as *CM*, and the universe of virtual models as *VM* (where *CM* and $VM \in M$) —, let's consider two models $m_a = \{a_1, a_2\}$ and $m_b = \{b_1, b_2\}$ (where $m_a, m_b \subset CM$). In a traditional composition process, and considering the simplest composition algorithm possible (i.e., to generate a composed model with the union of all elements present in the contributing models), the result would be a (concrete) composed model m_{ab} whose model elements would have the same values as the elements present in the contributing models but without being really the *same* elements, that is:

$$\otimes : M \times M \rightarrow CM$$

$$m_a \otimes m_b = m_{ab} \qquad \text{where} \qquad m_{ab} = \{a_1', a_2', b_1', b_2'\}$$

Alternatively, what we propose in our solution is a new model composition operator $\oplus$ that produces as result a virtual composed model vm_{ab} that, instead of being populated with mere

copies of the elements from contributing models, would actually use the real elements contained in those, avoiding thus the need of duplicating elements.

$$\oplus \ : \ M \times M \rightarrow CM$$

$$m_a \oplus m_b = vm_{ab} \quad \text{where} \quad m_{ab} = \{a_1, a_2, b_1, b_2\}$$

Both m_{ab} and vm_{ab} conform to the same metamodel mm_{ab}. This metamodel defines the set of concepts that can appear in the composed model, i.e., which elements from the contributing models may be part of the composed model. In the simplest scenario, mm_{ab} would be a subset of the metamodels of the contributing models but our approach allows as well the definition of new inter-model links or virtual attributes in the composed metamodel (that then becomes richer that the union of the contributing ones) as will be detailed further on.

In fact, since a metamodel is also a model it can be also virtualized, allowing for instance vm_{ab} (or even m_{ab}) to conform to a virtual metamodel vmm_{ab}. Similar to a virtual model, a virtual metamodel would not actually contain the metamodel elements itself but reference to the metamodel elements of the contributing metamodels. Nevertheless, virtualizing metamodels is not usually interesting because the size of metamodels is relatively small and therefore the benefits of virtualizing them are smaller compared with those we get when virtualizing models.

It is also important to note that once virtual models are perceived as regular models, nested virtualization may be achieved without further effort, i.e., the contributing models of a virtual composition may be also virtual models.

3.2 CONCEPTUAL ARCHITECTURE

To be useful, virtual models must appear as normal (i.e. concrete) models to the user. Therefore, we need to provide a mechanism to transparently use the elements from contributing models in the composed model. In short, we must modify the way model elements are accessed or viewed in a modeling environment, so that we can redirect an operation on a virtual element to the corresponding concrete one it refers.

As examples of modeling frameworks we can cite the *Eclipse Modeling Framework* (EMF), the *NetBeans Metadata Repository* (MDR), and Microsoft's *DSL Tools* and *SQL Server Modeling* (formerly *Oslo*). Each one of those has its own particularities but they all provide a *Model Access API* to allow the creation and manipulation of its models.

> **Definition 9.** *A **Model Access API** is the component of a modeling framework that provides an API that can be exploited by tools and users to access and manipulate models and their elements in that specific modeling framework.*

Although Model Access APIs present some differences (to adapt to the specificities of each modeling framework), they all share common functionalities. All of them must, for instance, provide operations to load and save models, or to get and set their model elements (and their properties). In this section, we will present our solution in terms of these generic operations to facilitate the implementation of our approach in any modeling framework. Next section presents the implementation for a specific framework (the EMF modeling framework in Eclipse).

The typical signature of these basic model manipulation operations is the following:

Method	Behaviour

Model loadModel(String uri)	loads a model from its persistence location (i.e., an XMI file, a relational database, etc.) identified by *uri*
void saveModel(Model m, String uri)	saves a model m to the location identified by *uri*
Element getElement(Model m, Object id)	gets, from the model *m,* the model element corresponding to the identifier *id*
void setElement(Model m, Element e)	sets, in model *m*, the model element *e*
Object get(Element e, Property p)	gets, from a model element *e*, the object corresponding to property *p*. The returned object may a simple data type value or another model element (e.g. when the property is an association/reference to another element)
void set(Element e, Property p, Object value)	sets the value of property *p* in model element *e* with the value *value*. As before, *value* can be a primitive type or an element

Table 1 : Model Access API's methods.

Sometimes *getElement* and *setElement* are not directly offered as separate operations but overlap with *get* and *set*. In that case, the framework offers an operation to get the root element of the model; using *get* calls we can navigate from that root object to the contained ones until reaching the desired object.

3.2.1 Model Virtualization API

The Model Virtualization API implements the interfaces present in a Model Access API and refines the model management operations in order to deal with virtual models. In a traditional model composition solution, where the composed model is a concrete model, a standard Model Access API simply accesses directly the element from a model. What we propose by virtualizing model composition is seamless integration between contributing and composed models by modifying the way the access to model elements is performed. The fact that the Model Virtualization API just provides an implementation of a standard Model Access API also means that a virtual model may be handled by any modeling tool, which is not aware of the underlying implementation (in fact it would not even be aware that is dealing with a model of a different nature). The Model Virtualization API was also built with a concern to make it easily extensible, making it possible to be implemented in different manners. The implementation of the Model Virtualization API will also change accordingly to the modeling framework it targets.

> **Definition 10.** *A **Model Virtualization API** is an API that implements a standard Model Access API modifying its behavior in order to allow the manipulation of virtual models by delegating the access to its referenced contributing models.*

When a given tool accesses a element from a virtual model it is in fact accessing a virtual element, which, as previously said, is a proxy to a concrete model element. The Model Virtualization API must then be able to navigate these proxies in order to reach the desired concrete element. The solution to handle this is through the use of a series of mappings relating virtual and concrete model elements (and virtual and concrete metaelements) that are built during the loading phase of a virtual model, allowing the API to know to which concrete model element a given virtual element

points (and consequently, to which concrete model it belongs). The Model Virtualization API then uses the standard Model Access API to access the elements from concrete models.

Fig. 2 depicts the relationship between the Virtualization API and the other APIs of the modeling framework. Tools access the virtual model using the standard methods in the Model Access API. This API may have different implementations (e.g. to access models stored in an XMI format, or in a database) and our Virtualization implementation. Calls to virtual models are automatically redirected to this Virtualization implementation (e.g. in Eclipse you can register what kinds of models each API implementation should handle). When processing the request the API will identify the referenced element(s) and request a get (or set or ...) operation on the contributing model where the element belongs. This request will be processed using the right API for the contributing model. If the requested feature is a virtual reference (see definition 7), the Virtualization API will use the Linking API (detailed in the next subsection) to handle it.

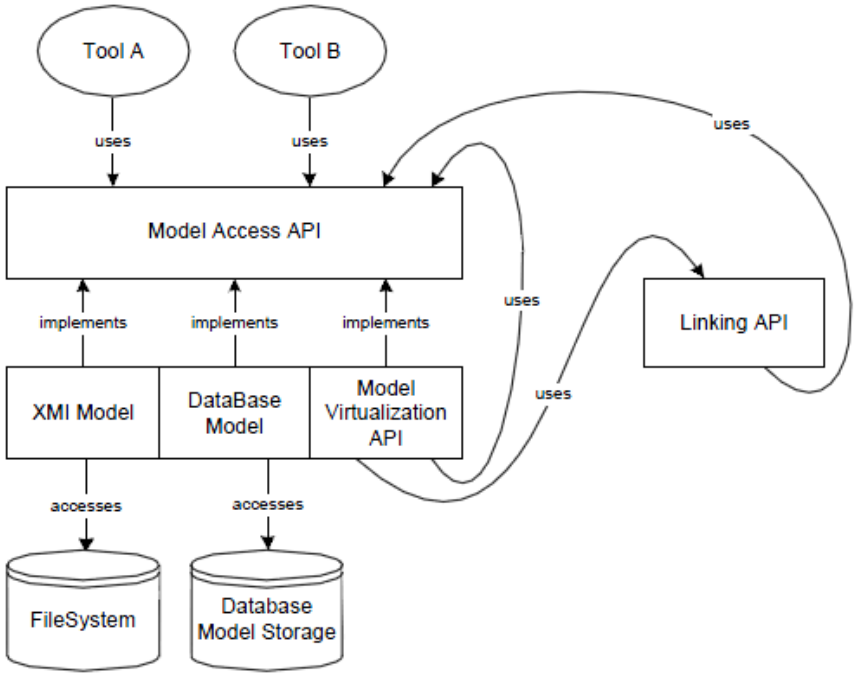

Figure 2 : API Relationship for Model Virtualization

In the following we describe the implementation of the model manipulation methods by the Virtualization API.

3.2.1.1 Load and Save Models

As virtual models do not hold concrete data, they just store the paths to the resources involved in the composition (concrete models and metamodels and, optionally, an inter-model link set detailing the virtual references). Therefore, when a *loadModel* operation is invoked, the Virtualization API performs the following tasks (Fig. 3): (i) load the virtual model file to get the paths to concrete resources, (ii) load all concrete metamodels and models, (iii) load the composition metamodel (or build it in case it is not specified), (iv) build the mappings that will assist the navigation from virtual to concrete elements, and (v) load the Linking API (that can also load an inter-model link-set, if available). For (ii) and (iii) the *loadModel* method of the standard Model Access API would be used. The save operation behaves analogously: all contributing models are saved (since they may

have been modified through the virtual model) using the standard *saveModel* method. A Linking API operation to save the virtual references is also called, in case they were updated.

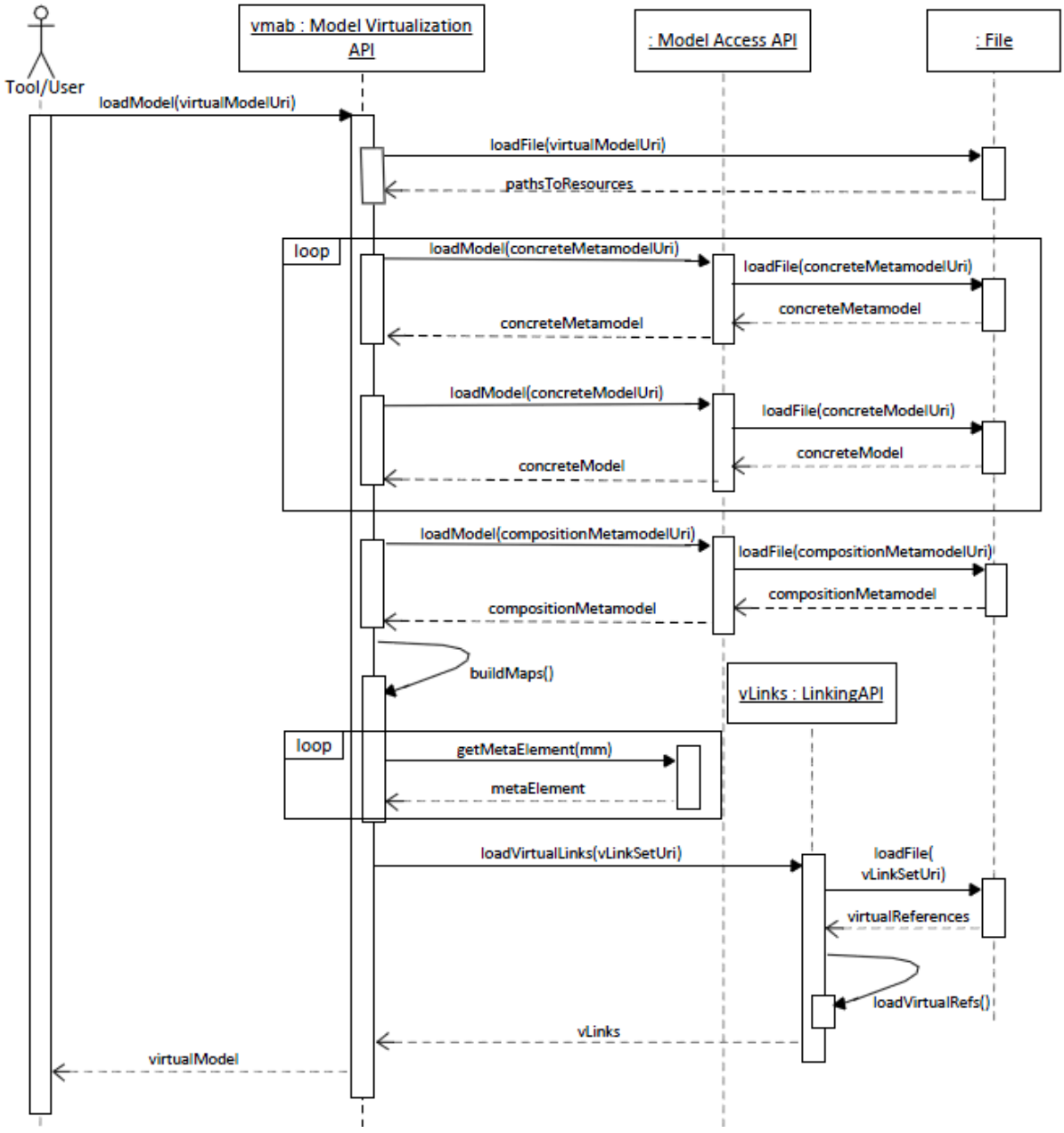

Figure 3 : *loadModel* operation of a virtual model.

Due to space limitations, the same lifelines and loops are used to represent different objects: contributing metamodels, models and the composition metamodel.

3.2.1.2 Get and Set Model Elements

The mappings creating during the loading phase allow the identification of the contributing element each virtual element represents. First, by comparing the type (metaelement from the composition metamodel) of a given requested (virtual) element, the Virtualization API can locate the

corresponding metaelement of the referenced concrete element (with Virtualization API's internal *virtualToConcreteMetaElement* method), and then use it to identify the container contributing model (with internal *getContainerModel* method). From there it can retrieve the concrete element itself by using the standard Model Access API's *getElement* on the concrete contributing model. There are several ways to implement this last step. The simplest option is to assign the same id to the virtual and concrete element. The behaviour for *setElement* follows the same pattern.

3.2.1.3 Get and Set Properties

Once retrieved a concrete element from its corresponding virtual one, it is straightforward to get or set its properties. Fig. 4 shows how the Virtualization API process a *get* request (property *name* of an element vb_1 in a virtual model vm_{ab}). First it needs to recover the correct concrete element containing the property (cb_1). As for the *getElement* description above, it does so by first discovering the metaelement of vb_1 (i.e. $vmeb_1$) and then by recovering its concrete version ($cmeb_1$), the container contributing model and finally the actual concrete element cb_1. Then, the Virtualization API uses the standard Model Access API to retrieve the name property of cb_1. The *set* operation behaves in the same way, only replacing the last get call by a set call.

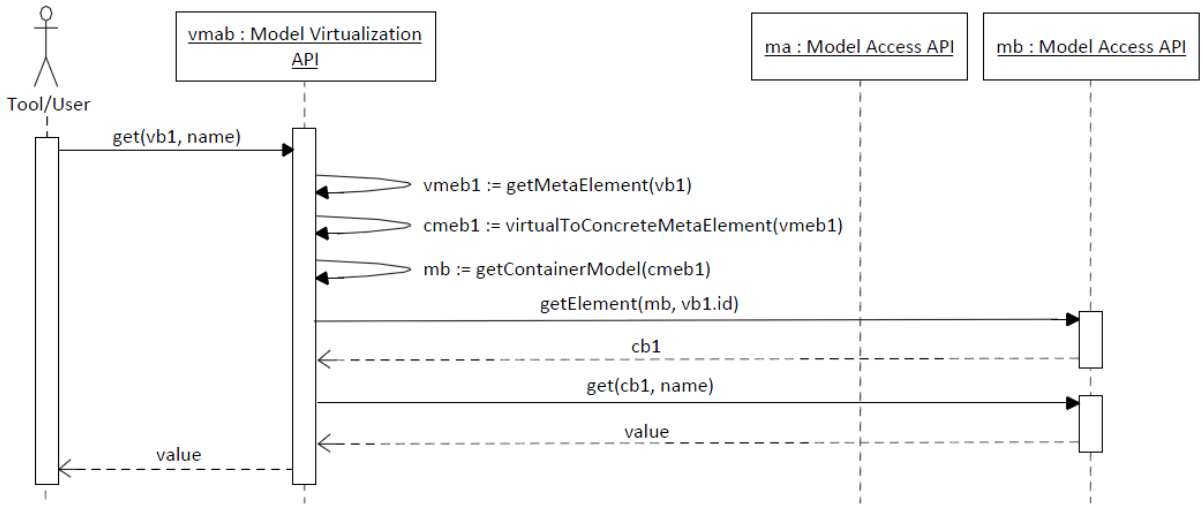

Figure 4 : A *get* operation on a virtual model

3.2.2 Linking API

An important aspect in model composition, besides the simple union of elements from contributing models, is the possibility of creation of relationships between them. To cover this aspect of model composition we proposed the use of a separate Linking API in our solution. The decision for a separate API to handle inter-model relationships was made in order to gain in modularity, since different implementations based on different techniques may be used in different contexts without the need of modifying the core of the Model Virtualization API. For instance, algorithms for automatic model composition based on different criteria (e.g., name matching) may be built.
The Linking API is used by the Model Virtualization API to navigate references (which we call virtual references) between elements contained in different models and, based on the type of the reference, provide the correct visualization on the composed model. Alternatively, it also provides the possibility to create model views (i.e., by hiding elements) and virtual attributes. Although model views and virtual attributes exactly inter-model relationships, a decision has been made to include these elements in the linking API since this information is also store in separate models (in order to preserve the original contributing models intact).

Definition 1. *A **Virtual Reference** is a reference in the virtual model that links two model elements contained in different concrete models.*

Definition 2. *A **Virtual Attribute** is an extra attribute in the virtual model that is not contained in any of its concrete models.*

The different types of virtual references we propose are:

- *Inter Model Associations*: similar to regular references between model elements, but between model elements contained in different concrete models. May have or not an opposite reference and different multiplicities;
- *Merging*: allows to indicate that elements in different models correspond to the same element in a system (semantical overlapping). After composition, only one the merged instance is presented to the user/tool and updates in it are propagated to all merged elements in the concrete models. It allows also to specify (through *FeatureMerged*, which feature from the merged element should be merged into a single feature;
- *Extension*: allows to specify that an element in one model extends an element in a different model, inheriting its properties;
- *Expansion*: allows to expand the virtual model with model elements (virtual attributes) not contained in any of the concrete models. It does not modify the original concrete models, and are stored in a separate model;
- *Model Views*: allows to view a model from a specific perspective, i.e., to hide elements not relevant for a given application. This is particularly important when dealing with large models. A user may, for instance, only desire to see model elements conforming to a specific metaelement, or to browse only elements whose values satisfy a given condition.

These different kinds of references may be implemented in different manners (e.g., one may implement it directly in different java classes), but we propose the adoption of models in every aspect of our solution and therefore we chose to represent the references in a weaving metamodel[7] (see Figure 5). Each reference between concrete models will then be an instance of one of the metaclasses in the weaving metamodel and will be stored in a weaving model conforming to this metamodel. By performing like this when the Model Virtualization API requests to the Linking API the access to one of the virtual references, it will check the type of the reference (e.g., merge or extension) and will perform differently according to it.

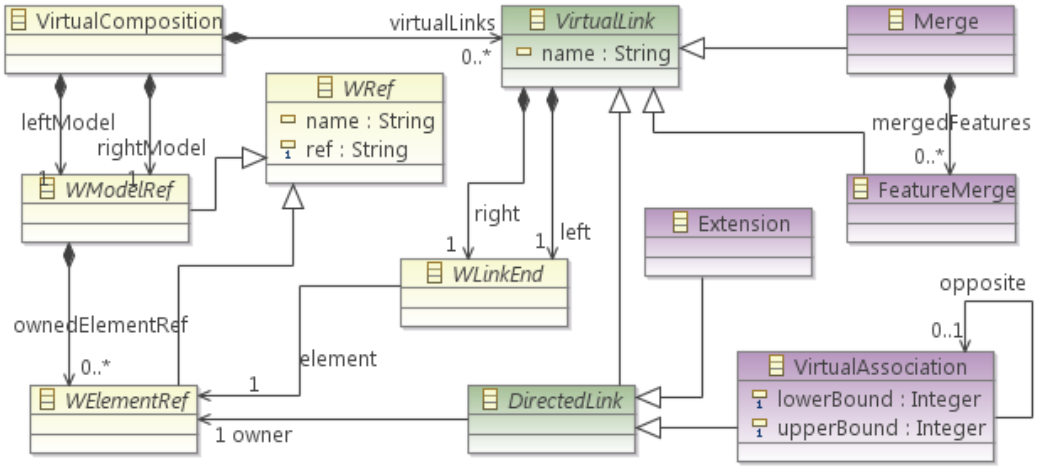

Figure 5 : Simplified Class Diagram of the Virtualization Weaving Metamodel.

For instance, when a virtual reference of type association is requested on one model element, the Linking API will identify to which model element (in a different concrete model) this element is linked, and return it. For two merged elements, one pivot element is chosen whose element will be displayed in the virtual model (with the other hidden). But when a *setElement* operation is requested to this merged elements, both will be modified in order to maintain consistency. For extension, the attributes of one element will be added to the attributes of the element that inherits from it.

For the case of virtual attributes the solution is a little different. Once a virtual attribute does not conform to any metaelement present in the contributing metamodels (we we do not want to modify the contributing metamodels since it goes completely against the philosophy of the solution of maintaining the consistency of contributing (meta)models for use in different scenarios), and it is mandatory that a metamodel describes all the possible concepts a given model may have, we handle this problem by adding a metaelement to which the virtual attribute will conform directly in the metamodel of the virtual model. But a problem remains when deciding how to persist the values of virtual attributes. To handle this we have opted to save virtual values in a separate extra model, completely dedicated to save their values. This keeps the elegance of the solution without interfering in any of the quality aspects we desire.

The main methods provided by the Linking APi are the following:

- *void loadVirtualReferences(String weavingModel-Uri)*: loads the weaving model based on the given *weavingModelUri* by invoking the *loadModel* method from the Model Access API. Also navigates the values the create the mappings between the elements participating in the virtual references;
- *void saveVirtualReferences(String weavingModel-Uri, Model m)*: saves *m* to the location specified by *weavingModelUri*;
- *Boolean isVirtualReference(Object o, Property p)*: checks if the property *p* of Object *o* is a virtual reference;
- *Object getVirtualReference(Object o, Property p)*: gets the referenced Object of element *o* by the (reference) property *p*;

- *void getVirtualReference(Object o, Property p, Object referencedObject)*}: sets the reference *p* in Object *o* to Object *referencedObject*.

Fig. 6 shows the use of the LinkingAPI to set a virtual reference. The first part of the operation (retrieving the involved concrete models) follows the same pattern as before (Fig. 4). Once it is detected that the feature link to set is in fact a virtual reference (operation isVirtualReference) the Linking API uses the operation setVirtualReference to set b2 as the value of r b2 for a1. Again, the implementation of this operation can be done in several ways. For instance, the Linking API could store the links as additional information in the contributing models or persist/load virtual references to an external location (we encourage the use of weaving models[7] for this, as storing links in a separate model avoids polluting the contributing models with extra information).

This is precisely one of the reasons to separate the Linking API from the Model Virtualization API: better modularity and easier creation of alternative implementations. At this stage of the research we only consider simple association links (e.g. element a in model ma references element b in model mb) between model elements in different models, but more complex ones could also exist (e.g. inheritance, merging,...). Support for these additional relationships could be easily integrated by providing a more powerful implementation of the Linking API without having to change the Virtualization API.

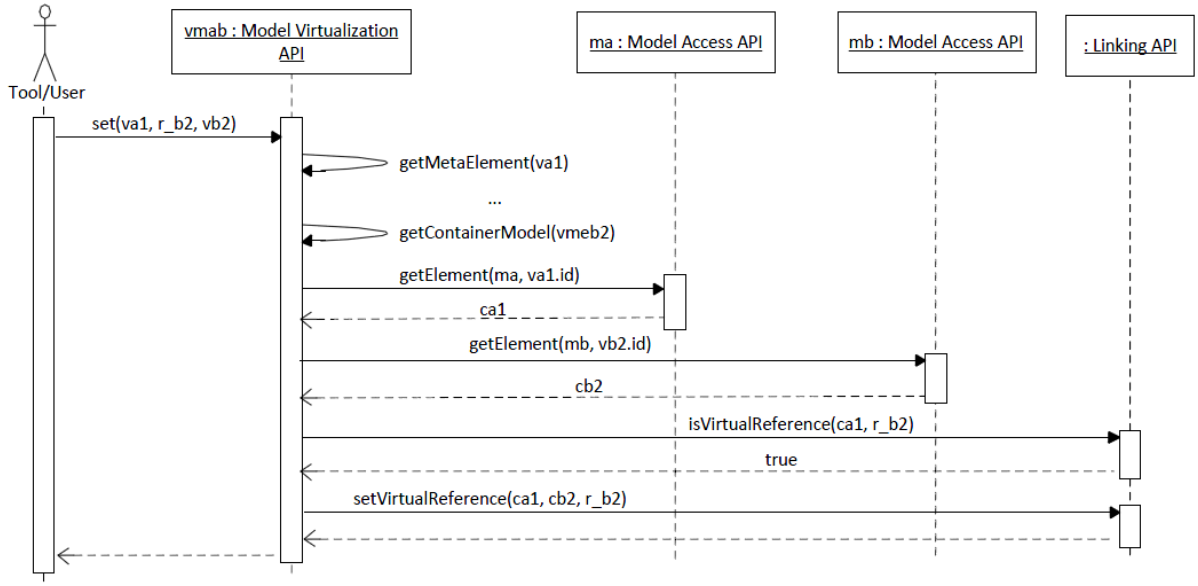

Figure 6 : A *set* operation for a virtual reference

4. CONCLUSION

In this document we have presented a new approach for model composition based on a model virtualization mechanism that offers a direct and transparent access to the contributing models used in the composition process. This is achieved by means of a specific Model Virtualization API that takes care of translating the model manipulation requests on the virtual model to appropriate operations on the elements of the comntributing models. Users of the model (and tools manipulating it) are not aware of this indirection when manipulating the virtual model. We have shown that this approach provides additional benefits to traditional model composition approaches.
As further work, we plan to extend the list of possible inter-model relationships to enrich our model virtualization mechanism and experiment with the virtualization of metamodels and with the composition of virtual models. For the latter, we plan to implement an existing model composition solution on top of our Model Virtualization solution, to prove that our virtualization mechanism is extensible and that virtual models can take the role of contributing models in other model composition process. We will also continue the development of our prototype to provide an adequate tool support to these extensions and to facilitate the experimentation of this approach with real end users.

5. REFERENCES

[1] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. ACM Comput. Surv., 18(4):323-364, 1986.

[2] J. B_ezivin, M. Barbero, and F. Jouault. On the applicability scope of model driven engineering. In MOMPES '07, pages 3-7.

[3] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In GaMMa '06, pages 5-12.

[4] S. Clarke. Extending standard uml with model composition semantics. Sci. Comput. Program., 44(1):71-100, 2002.

[5] T. Cottenier, A. van den Berg, and T. Elrad. Modeling aspect-oriented compositions. In MoDELS Satellite Events, pages 100-109, 2005.

[6] J. S. Cuadrado and J. G. Molina. A model-based approach to families of embedded domain-speci_c languages. IEEE Trans. Software Eng., 35(6):825-840, 2009.

[7] M. D. D. Fabro and P. Valduriez. Towards the e_cient development of model transformations using model weaving and matching transformations. Software and System Modeling, 8(3):305-324, 2009.

[8] F. Fleurey, B. Baudry, R. B. France, and S. Ghosh. A generic approach for automaticmodel composition. In MoDELS Workshops, pages 7-15, 2007.

[9] C. Herrmann, H. Krahn, B. Rumpe, M. Schindler, and S. V□olkel. An algebraicview on the semantics of model composition. In ECMDA-FA '07, pages 99-113.

[10] F. Jouault, F. Allilaire, J. B_ezivin, and I. Kurtev. Atl: A model transformationtool. Sci. Comput. Program., 72(1-2):31-39, 2008.

[11] D. S. Kolovos, R. F. Paige, and F. Polack. Merging models with the epsilon merginglanguage (eml). In MoDELS '06, pages 215{229.

[12] D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: a foundationfor model composition and model transformation testing. In GaMMa '06, pages13{20. ACM, 2006.

[13] D. S. Kolovos, L. M. Rose, N. D. Matragkas, R. F. Paige, F. A. C. Polack, and K. J. Fernandes. Constructing and navigating non-invasive model decorations. InICMT '10, pages 138{152.

[14] K. Letkeman. Comparing and merging uml models in ibm rational software architect v7.0 part 7: Ad-hoc modeling - fusing two models with diagrams. developer-Works, pages 454{470, 2007.

[15] S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave. Matching and merging of statecharts speci_cations. In ICSE '07, pages 54{64.

[16] Occello, A.-M. Pinna-Dery, M. Riveill, and G. Kniesel. Managing model evolution using the ccbm approach. In ECBS '08, pages 453{462.

[17] K. S. F. Oliveira and T. C. de Oliveira. A guidance for model composition. In ICSEA '07, page 27.

[18] R. Pottinger and P. A. Bernstein. Merging models based on given correspondences. In VLDB '03, pages 826{873.

[19] D. D. Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. Developing next generation adls through mde techniques. In ICSE '10, pages 85{94.

[20] M. Sabetzadeh, S. Nejati, S. M. Easterbrook, and M. Chechik. A relationship driven approach to view merging. ACM SIGSOFT Software Engineering Notes '06, 31(6):1{2.

[21] Vallecillo. On the combination of domain speci_c modeling languages. In ECMFA '10, pages 305{320.

[22] Z. Xing and E. Stroulia. Umldi_: an algorithm for object-oriented design differencing. In ASE '05, pages 54{65.