Extending UML for Model Composition

Jacky Estublier
LSR-IMAG
220, rue de la Chimie BP53
38041 Grenoble Cedex 9
France
jacky.estublier@imag.fr

Anca Daniela Ionita
LSR-IMAG
220, rue de la Chimie BP53
38041 Grenoble Cedex 9
France
Anca.Ionita@imag.fr

Abstract

The success of UML and more generally, of the model driven approach, has led to a proliferation of models, representing various systems, but the description of large applications may actually be composed of several system models. Therefore, the challenge is to determine how to compose independent system models, in order to build large applications.

We present in this paper the software federation approach developed by our team. A federation relies on the concept of a domain, which describes a specific application domain, represented with a specific meta-model. Building a software federation means composing independently developed domains, by composing their meta-models.

The UML standard, which is well supported by tools, has been found convenient for describing our meta-models, but it turns out that it does not contain sufficient modelling capabilities for the composition of these models. This paper analyses the need for model composition, the available UML concepts related to model composition and proposes an UML profile allowing the composition of models in software federations.

1. Introduction

A current trend in software engineering consists of managing programs at the level of their concepts, using modeling languages, in order to simplify their design and maintenance and to increase their robustness against the rapid change of technologies. This trend proposes to raise the level of abstraction at which programs are designed and developed, from code to models. This change has been compared to the jump from assembly language to the third-generation programming languages [1]. However, some basic conditions, for working exclusively with modeling languages, are their ability to describe in detail a large area of applications and the possibility to generate code.

Typical of this new trend is Model Driven Architecture (MDA) ([2],[3]) currently developed by OMG (Object Management Group), which is based on a four layer architecture, where each level is considered a language of the level below. MDA is supported by an extensible set of modeling standards, two of them being important for our work: the Meta Object Facility (MOF) [4] which is the top level formalism and the Unified Modeling Language (UML), described in MOF. Even though UML has been created as a general modeling language, it provides a profile mechanism [5], allowing specialization to specific domains, by providing a meta-level description of domain specific concepts.

More generally, Model Driven Engineering (MDE) [6] considers not only MOF, as the top level, but also other technologies like EBNF (Extended Backus-Naur Form) for languages, normal forms for data bases, DTDs (Document Type Definition) for XML (Extensible Markup Language) and so on. Domain Specific Languages (DSL) are representative of this kind of evolution toward many different and specialized metamodels.

Since software application requirements are ever increasing, the natural tendency is to develop ever bigger pieces of code, leading to huge software systems – that are not only difficult to build and maintain, but also to learn and customize. This is exemplified by the evolution of the available software tools (like Microsoft Word, Microsoft Office Project etc), which are growing in size, complexity and functionality.

An opposite trend tries to build software applications by allowing pieces of existing software to interoperate. The purpose is to build new applications with rich functionalities, while not increasing the dimensions of
individual tools, but making them inter-operate with either new or existing programs. Our work on the federation of components presented below belongs to this last trend, which places a strong emphasis on reusability. It offers an architecture and an integrated environment, capable of composing non-homogenous types of tools, like COTS (Commercial Off The Shelf), legacy software or any other products available on the market, designed to interoperate or not.

The federation approach relies on conceptual domains - high-level, platform-independent, object-oriented models, capturing the core domain concepts, which are translated then into Java classes. A conceptual domain is executable, and is the first reusable component produced in the development of a federation. The challenge is twofold:

- create conceptual domains capable of modeling a wide set of applications pertaining to the domain;
- compose conceptual domains in order to model the target application.

In order to create widely scoped domains, the conceptual domain is not a model of the domain, but a meta-model that defines the relevant concepts and the constraints that must be satisfied by each valid model in the domain. Therefore, the conceptual domain defines a family of models, all conforming to the same meta-model, i.e. using the same concepts and relationships. The domain designer imposes the meta-model and its associated tools (analyzers, interpreters, editors and so on), but the user defines his or her own model; it is a compromise between product lines [7][8][9], which are closed worlds and generic components [10], which have a wide reuse scope, but a low reuse rate.

Our system was designed to provide software developers with tools to easily define federations, from the definition of conceptual domains, to their composition and implementation based on Java. The UML modeling standard has been useful for defining the conceptual domains, but it does not contain enough elements to model the composition, so one should go higher, at the meta-modeling level and define appropriate UML extensions for the federation architecture (section 4). Some other works, attempting to give a more general, yet precise UML-based modeling for the composition of models are analyzed in section 5.

2. High Level Modelling in the Federation Architecture

The federation approach relies on a library of reusable application domains, each one described by its conceptual model, which is a small static model, containing its essential concepts. Unlike other approaches based on the separation of concerns, the domains are autonomous and independent. Vertically, a domain is structured into layers, separating the high-level executable model from the implementation tools, which may be non-homogenous - COTS, legacy components, applications available on the market or in-house developed programs. For each domain, the architecture allows either bottom-up development, if all the tools are already available, or top-down development, if it is necessary to build new tools.

The top abstract level contains a conceptual model, that represents the class diagram of the essential concepts and excludes implementation details. If complete models were used for describing the tools, as is usually done, then the result would be a huge composition model (especially if the federation is formed of many domains) – which is difficult to work with. The solution is to select just a few concepts, common to all the domain tools and useful for the interoperation with other domains. These essential concepts are grouped into the Conceptual Model (Figure 1), which is described as a class diagram.

![Figure 1. The architecture of the federation of components](image)

In order to allow the selection between different equivalent tools, the conceptual model is not connected directly to the concrete level; other intermediate levels are introduced for mapping between them [11]; they contain abstract roles, connected through wrappers to specific tools. Due to the separation between the domain abstractions and their implementation, it is possible to compose the domains irrespective of the selected tools.

The above described domains may be composed and configured in terms of the client’s specific requirements, creating composite domains, which are also reusable. A tool was developed for creating and composing domains. This paper addresses the way composition can be
performed, using an editor that allows the definition of the relationships between domains - at the level of their concepts. In this way, one can select the composition of the appropriate domains and the tools preferred by the client.

In order to analyze this particular type of model composition, consider Figure 2 - that describes the conceptual models of two domains: Project Management and Process. Such a composition might be necessary, because each domain has a set of features which are missing in the other. For example, the project management domain manages time, with all the information related to a task start date, end date, duration, delay etc. In fact, the purpose of project management is to relate tasks with a schedule and to monitor work progress. On the contrary, the process domain [12] has no information about time, but it automates task execution, starting activity as soon as its initial conditions are satisfied. Aspects of the organization of processes and their relations to other meta-models are also presented in [13], in a different context.

Figure 2. The conceptual models for project management and process domains

There are two important problems that must be solved, in order to make the composition of domains in a federation:

- **The synchronization between abstract and concrete levels**

  The tools work at the concrete level, but they should be permanently synchronized with their correspondent conceptual model. More precisely, the tools synchronize with an instance of the conceptual model, called the common universe. When a tool performs an action that affects the common universe, the synchronization intermediate layer is responsible for modifying the common universe accordingly. Conversely, a change in the common universe may have impact on the tools state; the synchronization layer must also notify the tools in order to synchronize with their local state.

  The synchronization layer is not the subject of this paper, but its presence has a fundamental consequence: the common universe behaves really as an executable abstraction of the domain. It is enough to reason at the conceptual level; the tools and the synchronization layer may be forgotten.

- **The synchronization between abstract levels of different domains**

  The conceptual model of a domain is supposed to be invariant, and the composition of conceptual domains also has its invariant side. Therefore, the domain composition should be carefully designed. Two important views of compositions can be identified:
  - the relationships between concepts from different domains; this composition is invariant and is part of the composition meta-model;
  - the collaborations between instances from different domains, describing the composite behavior.

  The former composition view is addressed in this paper.

  Why is it necessary to relate concepts from different domains in a static way? If one analyses the example from Figure 2, for a Project instance from the project domain, there should be a corresponding Process instance in the process domain. However, the project has the attributes: Name, StartDate, EndDate, Cost, Duration, while Process has the attributes Name and IdentificationCode. The concepts are not the same, the instances are independent, but they influence each other. There are many correspondences between their operations, e.g. AddTask() from Project and AddActivity() from Process. However, FindCriticalPath() from Project has no correspondence in Process, so the concepts are not identical, but are overlapping.

  The relationship between Project and Process involves then other relationships between their associated classes (figure 2) for instance Task and Activity, CompositeTask and CompositeActivity or even Interdependency (which specifies the way the tasks follow each other) and DataFlow (that makes the connection between Activity instances). There is also a relationship between Resource and Responsible, which states that, among the Resource instances allocated to a Task instance, one and only one is
linked to the Responsible instance of the correspondent activity.

These relationships do not represent just collaboration links, but associations between concepts from different domains, some of them between overlapping concepts. This composition between the static views of different domains should be well defined, to form a meta-modelling language that is capable of describing any domain and any composition of domains. UML has been studied as a possible meta-model for defining the composition but, as shown in section 3, it does not contain the proper modeling elements for describing federation peculiarities. However, in an effort to leverage existing UML tools, we elected to define our new meta-modelling language as a UML profile (section 4), which contains only the concepts necessary for defining the domains and their composition inside a federation.

3. Model Composition in UML and its Profiles

In order to work at the abstract level in a federation of components, it is necessary to have a language capable of modelling the domains and their composition. This would actually be a domain specific meta-modelling language that describes domains and domain compositions. The Unified Modelling Language has been studied from the point of view of composing elements that belong to different models, which is needed in our architecture.

UML models generally represent different views of the same system, using graphical diagrams to describe: use cases, classes, behaviour and implementation; all of them constitute the system model. Generally, an application contains only one system model, but large systems might be composed of subsystems, each one with its corresponding system model, not necessarily at the top of the hierarchy. There are two types of model compositions:
- Composing different views of the same system,
- Composing the same view of different systems

A) Composing models representing different views of the same system

Generally, the interdependency between UML diagrams that represent different views is implicit, it is not realized with relationships, but with constraints and well defined rules, in terms of semantics. Class diagrams represent the structure on which other diagrams are based; e.g. interaction diagrams contain the instantiated objects, while a statechart diagram describes the life cycle of a class with dynamic behavior; its activities should be present as operations in the corresponding class diagram; similarly, the operation calls from the interaction diagrams should also be consistent, meaning that the class of the target object should contain them.

However, for relating model elements from different models, the only specific relationships that are available are two abstraction dependencies, with the stereotypes <<trace>> and <<refine>> [5]. Trace relates model elements representing the same concepts found in different views, while refine shows correspondences between different levels of refinement, like analysis and design models. However, the UML standard states that the trace relationship “is rarely computable and is usually informal”, so it is difficult to use it in models intended to be executable.

B) Composing models representing the same view in different systems

The standard UML agrees with composing models that represent entire and different system models, but no special relationship is defined for this complex situation. How do we actually build the composition? Do we need something more than putting the pieces together and how do we specify it? Because elements are encapsulated in models, one should establish dependencies to import other model elements in a model namespace, or to be able to access them. Still, if a composition model is defined, it would contain all inter-model relationships and one would have the right to add relationships between any model elements. So, one might use generalizations and dependencies between models.

A useful kind of dependency called abstract reference, with the stereotypes <<Reference>> and <<ReferenceForCreate>>, has been introduced in the UML Profile for Relationships [14] for referencing an instance and changing the property values of the source with respect to the current values of the target. The reference dependency could be used for the synchronization between model elements from different systems, but with no possibility to specify which properties should be changed, if the elements do not correspond entirely.

The Enterprise Collaboration Architecture [15] gives a good answer for a composition seen as a collaboration between components, interacting with each other through ports. For example, a collaboration diagram for process component composition is very close to what is necessary for representing correspondences between federation roles, for the vertical mappings.

What is still missing is a possibility to explicitly define relationships between the system concepts, which are not changing with respect to a particular objects behaviour. Analyzing the Project and Process classes presented in section 2, their composition can be described as a collaboration, but there are also clear correspondences between their attributes and operations, at the static level, which do not depend on a specific scenario. The federation of components requires the definition of such
relationships between elements from models representing different systems.

It might be seen as an exception to the general principle of encapsulation, since big applications are currently built on the assumption that the parts have a hidden structure and they should only be composed or integrated with well-defined interfaces. However, the federation also hides the detailed internal structure by only allowing transparency at the conceptual level, that contains the fundamental concepts and relationships, but no implementation details; experience shows there are implementation details that need to be hidden. The federation makes visible the information necessary for reuse, in order to avoid rigid encapsulation, which is sometimes an obstacle for reusability and extensibility [16].

4. UML Extensions for the Federation of Components

A federation meta-model was elaborated as a basis for defining specific models, corresponding to specific applications based on this architecture. The elements useful for the composition of domains are presented below, as extensions of UML Core.

4.1. Extensions for the Federation Architecture

A FedeDomain is a package, containing the model elements that describe the layered structure of a domain. Each FedeDomain contains a FedeConceptualModel - a model described by a class diagram, representing concepts that are common to different tools that might implement the domain. They were modelled as stereotypes of Package and Model classes from UML Core (Figure 3). A FedeDomain can be either atomic or composite; a FedeCompositeDomain contains other domains, either atomic or composite and it also adds all the necessary elements for realizing the composition.

4.2. Extensions for the Model Composition

The FedeConceptualModel of a domain contains 4 types of modelling elements dedicated for composing domain models:

- Associations between concepts (classes) from different domains – modelled with Association from UML Core
- Correspondences between features, generally operations, from classes located in different domains, modelled with FeatureCorrespondence, a stereotype of Dependency from UML Core;

Figure 3. UML Extensions for the Federation Architecture

- Correspondences between overlapping concepts (classes) from different domains – modelled here with ConceptOverlap, a stereotype of AssociationClass from UML Core
- Emerging concepts in the composition domain, which generally model complex composition relationships, with more than two parts, modelled with class from UML core.

The previous sections have proved that domain composition in the federation architecture involves correspondences between similar concepts from different models and often simple associations are not enough. A specific definition is necessary, because the concepts do not correspond perfectly, they are just overlapping, so a <<trace>> dependency cannot be used. A possibility to model might be given by the <<Reference>> or <<Reference For Create>> stereotypes, but they have no way to specify which properties reference each other.

Figure 4. UML Extensions for the Composition Relationships inside the Federation

The correspondence between attributes and operations of different classes was modelled as a relationship

![Diagram of UML Extensions](image-url)
between features (because Attribute and Operation are derived from Feature in UML Core). This is a rich dependency, with the stereotype FeatureCorrespondence, that allows tagged values to specify, for example, the correspondence between the operations parameters. The constraint of this stereotype is that features should have the same type, they should be either two attributes or two operations; in OCL (Object Constraint Language) it becomes:

\[
\text{context: FeatureCorrespondence} \\
\text{inv: (self.supplier.oclIsTypeOf(Attribute) and} \\
\text{self.client.oclIsTypeOf(Attribute)) or} \\
\text{(self.supplier.oclIsTypeOf(Operation) and} \\
\text{self.client.oclIsTypeOf(Operation))} \\
\text{inv: self.supplier.classifier <> self.client.classifier}
\]

“Supplier”, “client” and “classifier” conform to the UML Core meta-model; there will also be other terms having the same source, in other OCL constraints defined in this section.

The relationship between concepts was modelled as an association class, with the stereotype ConceptOverlap; it is richer than a simple association, because its semantics include as tags feature correspondences between part or all of the attributes and operations of the classes representing similar concepts. As a constraint, these classes should belong to different system models, because a well-defined model could not be defined with such redundancies. The OCL representation is:

\[
\text{context: ConceptOverlap} \\
\text{inv: self.assocEnd->at(1).classifier.} \\
\text{fedeConceptualModel <> self.assocEnd->at(2).} \\
\text{classifier.fedeConceptualModel}
\]

The class overlap contains correspondences between features, but, if one of its attributes has the type of another class, it involves another correspondence at the level of classes. So, a class overlap may aggregate another class overlap.

Moreover, an important constraint is that a class overlap should have at least one correspondence between its features, either as a tag, for simple features, or as an aggregation of another class overlap. In OCL:

\[
\text{context: ConceptOverlap} \\
\text{inv: (self.featureCorrespondence->notEmpty()) or} \\
\text{(self.conceptOverlap->notEmpty())}
\]

Correspondences between features may be contained directly by the composite domain, even for classes which are not overlapping concepts. A FeatureCorrespondence is not always a part of a ConceptOverlap.

\[
\text{context: FedeCompositeDomain} \\
\text{inv: (self.featureCorrespondence –} \\
\text{self.conceptOverlap.featureCorrespondence)} \\
\text{->notEmpty())}
\]

These stereotypes may be considered as a basis for defining correspondences between concepts; the current studies are oriented towards the specification of particular details for the federation architecture, regarding:
- rules and constraints for their instantiation, to create correspondences at the model and object levels;
- managing the relationship life cycle;
- introducing design elements related to the model transformation into code.

From a structural point of view, correspondences have been studied in length by the database community, for schema migration and for federated data bases. Indeed, in both cases, two (or more) database schemas represent, at least partially, the “same” information [17], [18]. A correspondence must be explicitly and manually defined.

The definition and control of behavioural consistency for the correspondences between the states of different domains and the interactions between concepts across system models, are difficult research topics, not addressed in this paper.

5. Related Work

Several other approaches based on UML extensions are analysed in this section, because they are similar to the federation of components from the point of view of their structure (based on the separation of concepts) and/or their attempt to resolve composition at the metamodeling level.

A very important UML profile, which should be compared to the federation, is Executable UML [19]. It also decomposes the system into domains, by sorting the requirements, expressed as use-cases. Both approaches define the mission and preserve the autonomy of the domains. The major difference is the unit of invariability of a domain; in Executable UML, the invariability is the domain mission: a domain could be replaced by another one with different concepts, but the same mission; in the federation of components, the invariability is the conceptual model, which can be implemented by different tools. Both of them compose domain models, even if in different ways, but the federation imposes a layered architecture, to make the separation between the models and their implementation tools. While in Executable UML the domains which are already realised are not modelled, in the federation any composition of domains is based on its high-level model, containing its essential concepts. Similar to how the federation defines new relationships for composing domain models, Executable UML also allows the definition of bridges, as asymmetrical dependencies, or as joint points between aspects.

Another composition relationship was introduced by Siobhán Clarke [20] as a new kind of relationship for UML; it assumes that the application has been previously decomposed in terms of its requirements, to obtain the
separation of concerns, and one has to do the composition. Overlapping concepts are also identified, but they are treated differently than in the federation of components. The federation keeps the models unchanged, while including them into a composite domain, which also contains all the new concepts and relationships added for the composition. On the contrary, Clarke’s approach attempts to merge the models. There are two kinds of compositions presented in the paper: override (everything is replaced by a new version) and merge (if there are corresponding elements, the composition contains a single element and if they do not correspond exactly, reconciliation strategies should be proposed with respect to the application).

A merge-by-name scheme was also used for creating composite templates that wove different concerns, in the model-driven development approach presented in [21], while overlapping between concepts from different models was also studied for meta-model-based model transformations in [22]. However, overlapping is discussed in this paper for composition only, not for model transformation, it can be seen as a synchronization between similar concepts from different domains. Our approach can also be considered as another way of implementing a domain-specific model-driven architecture than the one described in [23], where “bridges” between models of different nature and maintenance of consistency between models are also discussed.

The meta-model composition from [24] was designated for creating generic modelling environments and a toolkit was supplied for defining and interpreting the models (generating some source code, configuration files and reports). UML was also completed with new operators to support meta-model composition:

- the equivalence operator – which makes a union for the properties of two UML class objects, similar to merge composition from [20];
- two kinds of inheritance operators – interface and implementation inheritance, that differentiate between inheriting a class role or its structure.

Even though these elements cannot be applied for the composition in the federation architecture, the goal is quite similar: composing domain-specific meta-models and interpreting them.

Another approach for integrating and reusing conceptual models can be found in [25], where a joint action model is proposed for managing the relationships between requirements and discipline-specific system views. The course-grain architecture is similar to the federation, with its vertical links for synchronizing two levels of abstraction: conceptual models and engineering models, plus horizontal synchronization between models, performed at a high level only.

6. Conclusions

The new focus on models and the persistent necessity to compose new applications from existing “components”, clearly stress the need for model composition. Indeed, domain-specific meta-models are published as a way to simplify the reuse of their domains. For large applications, it is clear that they focus the contributions of different domains, each one represented by its meta-model; the issue is how to compose these domains at the meta level. We claim that these composition mechanisms are absolutely necessary for allowing model engineering to become an industrial reality.

The federation environment has already explored these model engineering issues for several years and has identified different kinds of compositions: meta-model and model compositions between overlapping or non-overlapping concepts, hierarchical composition, vertical composition and so on. The federation environment originally developed ad-hoc tools for supporting composition design and implementation. When the issues were clear enough, we tried to use UML - in order to integrate our approach with the many tools around it. This led us to realise that UML focuses on composing different views of a unique model, and does not offer facilities for composing models or meta-models coming from independent software.

The contribution presented in this paper consists of describing the federation experience about model composition and proposing an UML extension that allows the composition of different system models, inside the conceptual framework of UML.

More generally, we believe that model engineering will not rely on UML only, seen as the unique and universal modelling language, as proposed by OMG, but more likely on a set of domain-specific meta-models, seen as Domain Specific Languages. These domain-specific meta-models, sustained by specific editors to allow easy design, will considerably simplify the realisation of the associated code generators. Creating an application that spans several domains, will therefore involve composing their meta-models. Even though this paper presents a model composition profile for UML, the proposed solution can be adapted as a meta-model directly based on MOF as well as for most technological spaces [3], therefore it can be considered as a general approach for meta-model composition.

The experience acquired by building composite federations has revealed the need for describing in detail specific mechanisms of model and meta-model composition and the work presented here shows a way to perform these compositions.
Acknowledgements: This research was partly supported by a Marie Curie Intra-European Fellowship within the 6th European Community Framework Programme.

References


We are grateful to the whole Adele team that designed and developed the Melusine federation environment, most notably J. Villalobos, T. Le and G. Vega.