

The Synchronization of Independent and Specific Models

Vincent Englebert
University of Namur (Belgium)
Email: vincent.englebert@info.fundp.ac.be

September 11, 2002

1 Presentation

CASE¹ tools are invaluable to support software engineers. Indeed, verification, generation of code, simulation, or reverse engineering tasks are very frequent and often cumbersome. Moreover, applications become larger, more complex and have to meet new requirements such as certification, metrics, etc. The apparition of UML² as a franca lingua has made companies more aware of the need of methodologies and of tools to carry out large projects. More recently the OMG has recently proposed the MDA for supporting the engineering of large applications. The MDA makes a clear distinction between independent/conceptual models (PIM³) and specific/physical models (PSM⁴). With respect to this decomposition, the conceptual model of some business activity can be implemented on multiple platforms in using a refinement process from some PIMs to PSMs. The inverse process could also be taken into consideration for reverse engineering tasks. Such process are made of transformations which are natural steps between distinct levels of abstraction (conceptual, physical, analysis, design, ...) [4, 7]. Many efforts have been done to identify or to formalize such transformations and they constitute a real added value for CASE tools [3, 7]. But in many cases, transformations pose a problem to ensure the traceability between origin and target models, since generally either the model is really transformed and the argument model is destroyed⁵ or we preserve the argument without keeping the links between them⁶.

This article exposes how some transformations (and more particularly refinement transformations) can be expressed without losing the trace between the source and the target models. Although this mechanism relies on a meta-repository of a meta-CASE that is not UML/MOF compliant, we believe that this could interest researchers working in the MDA realm.

The section 2 presents the concepts of our meta-repository that are necessary to understand our approach. The next section explains how it is possible to support simple transformations with this kind of repository. The main components of the meta-CASE are briefly presented in section 4 and the section 5 shows a case study about the modelling of distributed architectures.

2 Repository

The meta-CASE we have developed relies upon a meta-repository that can model meta-classes, meta-relations, meta-properties, and meta-models as well as their instances. The *meta* prefix is used in this article as a relative operator between distinct modelling levels⁷. Meta-models are aggregate concepts defined in terms of meta-classes and meta-relations which make up its domain. We note the instances of meta-classes and meta-relations respectively instances and relations.

¹Computer Aided Software Engineering.

²Unified Modeling Language.

³Platform Independent Model.

⁴Platform Specific Model

⁵Unless some kind of versioning mechanism has been activated [5].

⁶Some tools keep a journal that makes it possible to retrieve this link a posteriori [9].

⁷The instances of a meta-class can thus denote objects that are not classes.

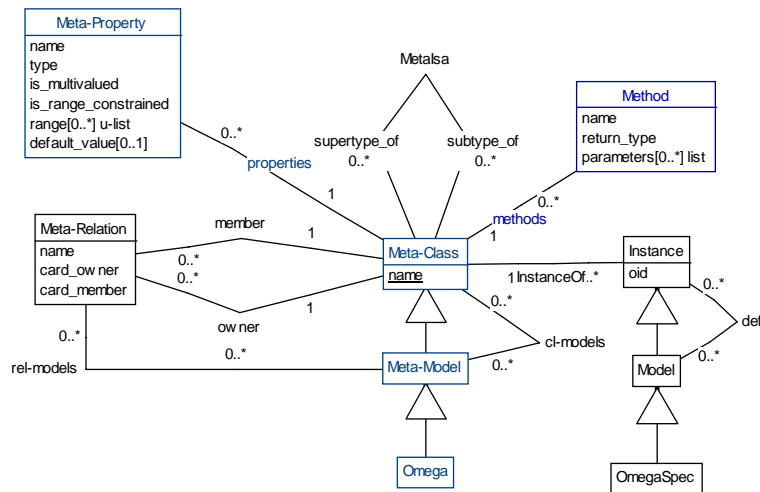


Figure 1: **Meta-Meta-Model** This schema shows the definition of our meta-meta-model. Technical details have been erased for pedagogic reasons. The `Model`s and `RelDomain` relationships denote the possible constituents of a meta-model.

Meta-relations denote possible one-to-many relationships between meta-classes. Meta-classes can inherit from several other meta-classes, and are described by meta-properties and methods⁸.

At the instance level (i.e., M1 level in the MOF), models are composed of instances and relations. Contrary to the MOF⁹, our models can share both instances and links between them.

In our approach meta-models and meta-classes are not independent concepts, indeed, we have modelled the `Meta-Model` meta-meta-class as a subtype of `Meta-Class`. This allows us to endow the meta-models of all the characteristics of a meta-class: meta-properties, methods, meta-relations, and inheritance. When a meta-model inherits from another one, its domain is composed of the domains of its supertype meta-models as well as of some additional meta-classes and meta-relations that are then specific to its semantics. However, meta-classes can not inherit from meta-models. This generalization is very useful in a lot of modelling tasks and that will be illustrated in the sequel.

The schema of Fig. 1 depicts the static diagram of our repository. The reader will find the main concepts explained so far, as well as a description of the instances (`Instance`, `Model` and `OmegaSpec`). The `Omega` meta-meta-class is a singleton and denotes a special meta-model which encompasses all the other ones, this corresponds to what OODBMS call a *root*. This special meta-model has only one instance (the `OmegaSpec`'s instance).

Figures 2 and 3 show how to use this meta-meta-model to define the meta-model of statecharts and how this can be instantiated to produce the statechart of a *switch*.

The generalization of meta-models to meta-classes makes it possible to define advanced concepts quite elegantly. Let us make our statecharts a little more realistic by adding `OR` and `AND` states. They are states that can be refined by either one or several parallel statecharts. It is easy to keep this requirement into consideration in our approach. Firstly, we define a new meta-model `OR-state` which is a subtype of the `state` meta-class and that encompasses meta-classes `state`, `init`, `final`, `transition`, `AND-state` and `OR-state` (i.e., it-self). Secondly, we define a new meta-model (`AND-state`) as a subtype of `state`. It is defined just in terms of statecharts. The resulting meta-model is depicted in Fig. 4.

The last example of this section shows other benefits of our approach. In complex architectures, a same concept can often appear at several places with distinct semantics. For instance, a table of a relational DBMS could be presented as a CORBA interface on some ORB bus and be implemented by a Java skeleton. We have thus three “*objects*” that denote essentially the same thing. Their definition must obviously be synchronized oneway or another. We can easily model the common

⁸The meta-repository has its own programming language.

⁹[8], page 1.15.

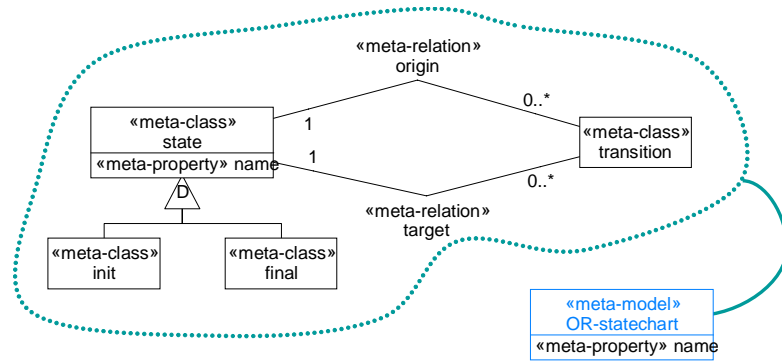


Figure 2: **Meta-Model** This meta-model describes simple statecharts in terms of concepts of our meta-meta-model. To help the reader to understand this diagram, we have added stereotypes to indicate the type of every element. The *lasso* shows the definition of the `Statechart` meta-model.

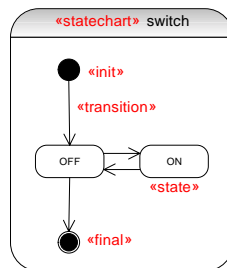


Figure 3: **Model** This diagram is a possible visualization of a model which complies with the `Statechart` meta-model.

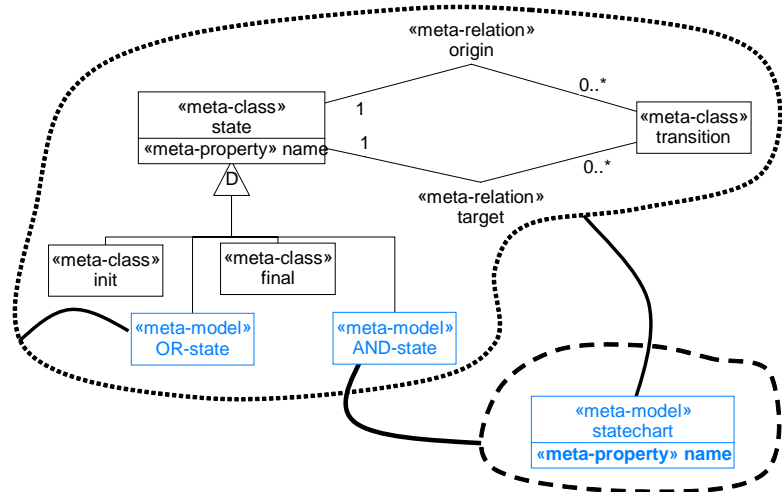


Figure 4: **Statechart Meta-Model** This schema depicts three meta-models: `OR-state`, `AND-state` and `statechart`. `AND-states` comprise only statecharts although `OR-states` may contain every kind of state.

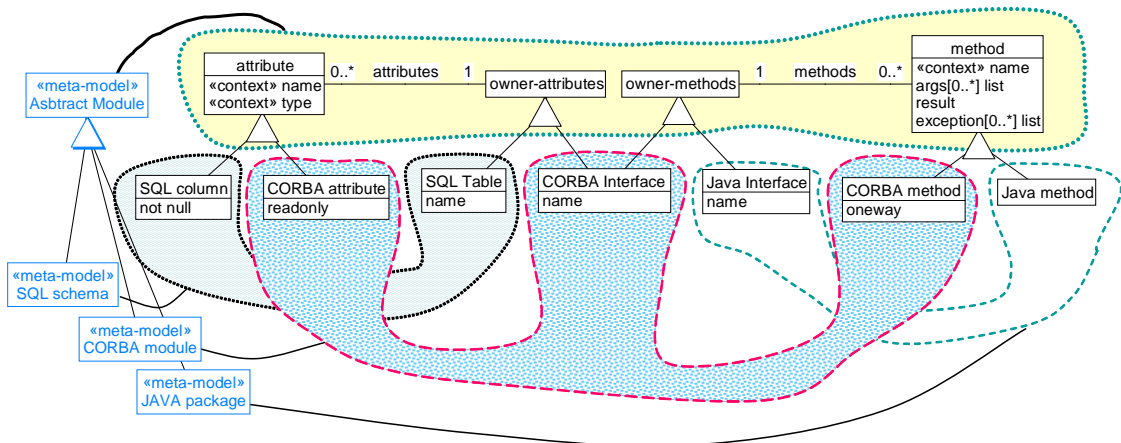


Figure 5: **Integration of meta-models**

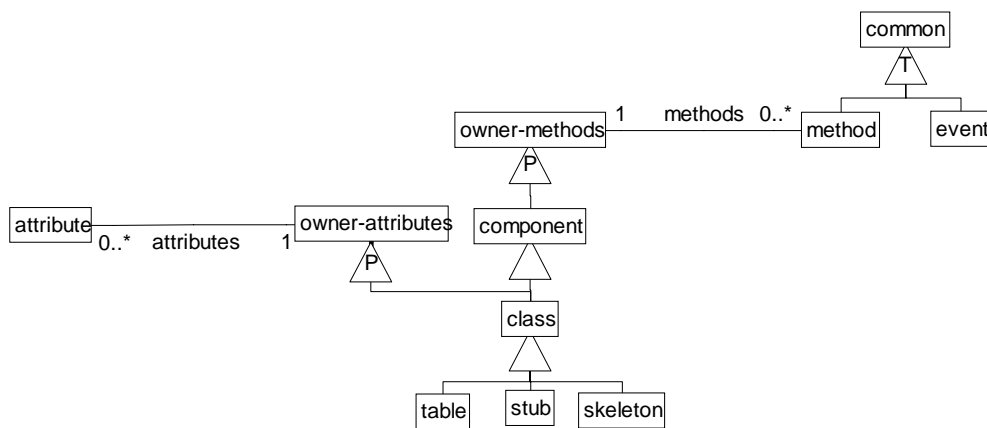


Figure 6: **Dynamic specialization and inheritance**

properties of these *objects* in a meta-model (**Abstract Module**) that will capture their essence. This meta-model can be declined/specialized in three versions: *relational*, *CORBA* and *Java*. Each one will extend the concepts of the abstract module with its own characteristics: the CORBA meta-model will precise if methods are *oneway* or synchronous and the relational meta-model will specify if columns are optional or *not null*. Figure 5 describes meta-models which permit us to define one or more models sharing common information such as the *customer* entity that can be “conjugated” in a SQL table, a CORBA interface or a Java interface.

Moreover, our meta-meta-model allows meta-properties to be contextual. This means that one instance can have several values for a same meta-property which depend on the model the instance belongs to. In our example, *attribute.name* and *attribute.type* are such meta-properties. Indeed, the type of an attribute could be *char(30)* in SQL, *wstring* in CORBA, and *String* in Java.

Of course, the semantics of this repository is more complex and we have just presented its main lines of force. Our aim is to show that it is possible to reach a great expressiveness with just few concepts (meta-class, meta-model, ...). To illustrate our proposition, we will explain in the next section how some transformations can be naturally expressed in our approach.

3 Transformations

In forward/reverse or re-engineering, transformations are natural steps between distinct levels of abstraction.

Our semantics allows the method engineer to specify many transformations as a dynamic specialization of objects inside the hierarchy of its meta-classes. Contrary to most modern languages, such specializations may occur a posteriori, once the object has already been instantiated. For instance, a concept that has been identified as a component must often be transformed to a class in the design phase, and to a stub (resp. a skeleton) in order to distribute it in the system and to a relational table to save its state. In the same way, in UML, a transformation could refine a static diagram to generate methods from all events that have been identified in a statechart or a collaboration diagram. For such transformations, the hierarchy of types defined in Fig. 6 suffices to support them automatically in our meta-CASE. Indeed, the type of an object (i.e. its meta-class) can mutate inside its type hierarchy, and moreover, an object can have distinct types in each model it belongs to. Hence, the same object (for instance the `customer` concept) could be defined as a component, and next be refined into a class in a static diagram, into a stub and a skeleton in some deployment diagram, into a table in a relational schema and some of its methods could be *merged* with events from a statechart in creating a common supertype. Moreover, the meta-CASE can exploit this semantics to propose automatically all the tools to help the engineer in his task (i.e. to transform).

So with a simple magic wand, the traceability is automatically maintained and the life cycles of these objects are synchronized. Of course, this mechanism is not the universal panacea. Many transformations are beyond this principle, but we believe that for simple transformations (and more particularly refinement transformations) this solution is simpler and more elegant.

4 The Meta-CASE Architecture

The meta-CASE we have developed consists of 3 components: the repository we have just described, an abstract machine to execute Voyager II+ programs, and an interpreter to display the models. Voyager II+ is a Pascal-like programming language with a garbage collector, object-oriented features, a lexical analyzer, declarative queries, and meta-facilities (reflexivity, ...). The last component is in charge to interpret Grasyla¹⁰ rules which define the presentation of the models on screen. This language is declarative and only a few lignes (about 20 or 30) have sufficed to produce the views of the figure 7. The graphical user interface is automatically deduced from the meta-models definition.

5 Case Study: modelling distributed architectures

Modelling distributed architectures is a crucial need and most methodologies still neglect this aspect [3, 1]. Indeed, we find either very informal descriptions or more formal descriptions such as the architecture description languages [6]. but that last approaches are mainly a posteriori techniques to model and to check systems once the architecture is already well defined. Our research group is investigating the definition of a methodology that would encompass the whole cycle from informal descriptions to design and implementation phases. We will use this context to illustrate a representative except that uses concepts from ADL languages to define component types (for instance pipes and filters), components (an analyzer, a parser and a semanticizer) that comply with the previous component types as well as their IDL interfaces and their deployment on the physical architectures (network and nodes). The component types and components would be part of PIMs and the IDL interfaces as well as their deployment would be part of PSMs in the MDA approach.

The four models shown in Fig. 7 illustrate how our approach allows engineers to specialize (and thus to transform) conceptual components (we have used the concepts of the ACME language [2] for this purpose – the “distributed parser” window) to IDL interfaces (the “compiler” window) that are next deployed on nodes borrowed from the model of some physical network (the “CNRS-FUNDP” window). All these views have been defined with Grasyla and any modification on one of these models is immediately propagated to the other models. This synchronization is enabled by the way concepts have been defined: they have been refined by specialization one from

¹⁰GRAphical SYmbolic LAnguage.

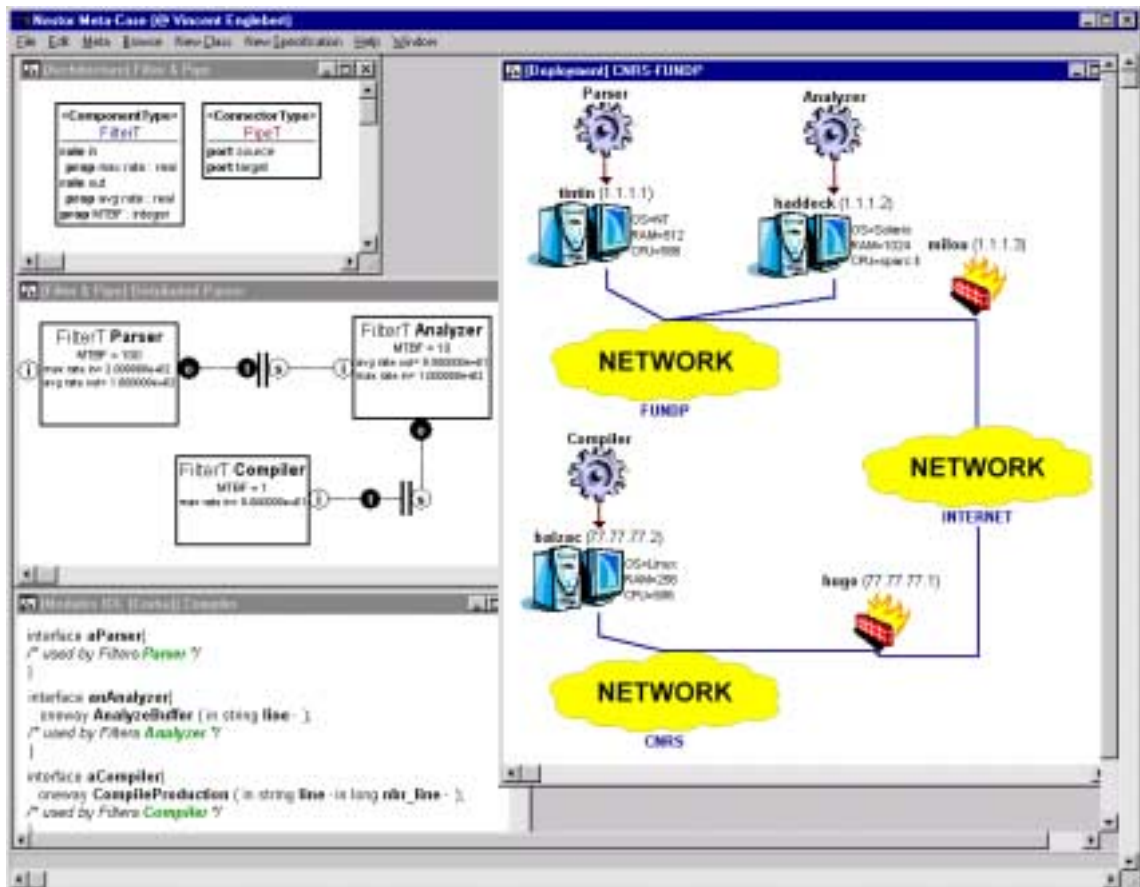


Figure 7: **Architecture** This figure illustrates how our meta-model allows us to synchronize data shared between PIMs and PSMs. The upper left model depicts a type of architecture whose components are either *filters* or *pipes*. The model in the window below describes an architecture that complies with the previous architectural style: there are three components that are linked together with two pipes. The large window on left shows the possible deployment of those components on a physical network (nodes, firewalls, ...). Finally, the last window is a possible implementation of the architecture where every component has been implemented as a producer of data.

another. For instance, the `Parser` filter denotes exactly the same concept than the `aParser` Corba interface or the `Parser` component on the `tintin` node. The IDL interfaces have been produced with the hypothesis that components produce the connections (i.e. the parser calls the analyzer), another interpretation would have been possible if the components were then just consumers and no more just producers (i.e. the analyzer would then call the parser). Our tool permits us to show several PSMs that come from one (or more) PIM according to distinct hypotheses. Of course, our principles are still valid and all those models (and their views) would again be synchronized.

Because PIM and PSM are closely related, we think that this kind of dynamic specialization can be useful to maintain the traceability and the synchronisation between them.

6 Conclusion

We have presented a meta-repository that is both very simple to understand and expressive enough for complex and realistic needs in the software engineering realm. The possibility to specialize dynamically objects inside their inheritance graph predisposes this mechanism to manage the traceability and the synchronization between models, and more particularly between PIMs and PSMs. Moreover, the generalization of meta-models into meta-classes allows one to model easily refinement process. Meta-modeling experiments have been done with database models, statecharts, ADL constructs, security models, organizational structures, etc. A prototype has been developed in C++ and the reader can view several screenshots on our site¹¹.

References

- [1] Garlan, D., S.-W. Cheng and A. J. Kompanek, *Reconciling the needs of architectural description with object-modeling notations*, Science of Computer Programming **44** (2002), pp. 23–49.
- [2] Garlan, D., R. T. Monroe and D. Wile, *Acme: Architectural description of component-based systems*, in: G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, Cambridge University Press, 2000 pp. 47–68.
- [3] Große-Rhode, M., F. P. Presicce, M. Simeoni and G. Taentzer, *Modeling distributed systems by modular graph transformation based on refinement via rule expressions*, in: A. S. M. Nagl and M. Münch, editors, *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, number 1779 in LNCS (2000), pp. 31–45.
- [4] Hainaut, J.-L., *Specification preservation in schema transformations – Application to semantics and statistics*, Data & Knowledge Engineering **16** (1996).
- [5] Jarke, M., R. Gallersdorfer, M. Jeusfeld, M. Staudt and S. Eherer, *ConceptBase – a deductive object base for meta data management*, Journal of Intelligent Information Systems, Special Issue on Deductive and Object-Oriented Databases **4** (1995), pp. 167–192.
- [6] Medvidovic, N. and R. N. Taylor, *A classification and comparison framework for software architecture description languages*, IEEE Transactions on Software Engineering **26** (2000), pp. 70–93.
- [7] Miličev, D., *Automatic model transformations using extended UML object diagrams in modeling environments*, IEEE Transactions on Software Engineering **28** (2002), pp. 413–431.
- [8] Object Management Group, *Meta object facility (MOF) specification*, Technical Report Version 1.3, Object Management Group (2000).
- [9] Roland, D., J.-L. Hainaut, J. Henrard, J.-M. Hick and V. Englebort, *Database engineering process history*, in: *Actes du deuxième workshop international sur les différentes facettes de l'ingénierie des processus (MFPE'99)*, Gammarth, Tunisie, 1999.

¹¹<http://www.info.fundp.ac.be/~ven/screenshots>.