An OO interpretation of graphs as meta-CASE's Meta-meta-model

Englebert, Vincent

Published in:
First International Conference on Graph Transformation

Publication date:
2002

An OO interpretation of graphs as meta-CASE's meta-meta-model

Vincent Englebert [1]

Computer Science Department
The University of Namur
Namur, Belgium

Abstract

This paper presents the meta-meta-model of a meta-CASE that is largely based on graphs. Nevertheless, the graph aspects have been hidden behind an extended object-oriented model that makes easier the modeling task: the generalization of meta-models into meta-classes as well as the possibility to edit the inheritance relationships at the instance level makes this approach both simple and expressive.

1 Introduction

CASE (Computer Aided Software Engineering) 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 UML (Unified Modeling Language) as a franca lingua has made companies more aware of the need of methodologies and of tools to carry out large projects. Nevertheless, CASE tools generally support a limited set of models and do not allow us to extend them with new notations, new models, and new functionalities. For this reason, researchers have proposed the concept of meta-CASE tools since the 90's to respond to those criticisms [1]. Such tools add an extra abstraction level in the general architecture of a CASE tool with the result that they are no more hard-coded, but are just the result of an interpretation of some higher level that can be edited on the fly, and by way of consequence, their specification can evolve. To date, about ten operational architectures are available [10,11,15,17,18].

[1] Email: `vincent.englebert@info.fundp.ac.be`

This paper describes a new architecture that makes it possible to represent the information managed by a CASE tool as a graph with an interpretation that is close to the OO concepts. This graph will be the core repository of our meta-CASE which is presented in section 2. Section 3 explains how can some transformations be subsumed in the repository semantics. Section 4 will describe the global architecture along with the Voyager II+ programming language and the Grasyla visual language. Before concluding this paper, we will show the expressiveness of our concepts on a case study (modeling distributed architectures).

2 Repository

Our experience with CASE tools [4,8] shows that a modeling task leads to a lot of specifications that can be related together to form complex graphs where some nodes can be exploded in other graphs. We explain in the next paragraphs how we have derived an OO meta-meta-model to represent these graphs.

If we investigate a single graph, we can observe a strong analogy between the concepts of node/edge and resp. meta-class/meta-relation. Adding meta-properties to meta-classes allows us to decorate the nodes with values. Meta-classes can also have methods written in Voyager II+ — the language is presented in the sequel. It remains to identify a graph with a meta-model to close this first step. So, a meta-model is made of meta-classes described with meta-properties and methods and linked together by meta-relations. To achieve an OO meta-meta-model, we endow the meta-classes with the possibility to inherit from other meta-classes (multiple-inheritance with possible disjoint specializations).

The second step consists in modeling the possibility to explode a node into another graph. Indeed, so far, we have just modeled “flat” graphs. The more natural way to proceed is to define meta-models/graphs as specializations of meta-classes/nodes. By doing so, meta-classes can be meta-models, that are themselves described in terms of meta-classes, and so on. Moreover meta-models can inherit from other meta-models or meta-classes[2].

Although the relationship between a meta-model and its constituents (i.e., its definition) is close to the concept of aggregation (cfr. UML), our semantics includes the meta-relations. Moreover, contrary to UML, a meta-model (say M) has one alone definition that can be extended to other meta-classes in possible subtypes of M. The aim of this new “aggregation” is to make possible the complete automation of its behaviour in the tool. For instance, only the more specialized definition can be instantiated. Let us imagine a `Temporal E/R`[3]

[2] Meta-classes can not inherit from meta-models.

[3] i.e. Entity/Relationship diagram with temporal annotations.

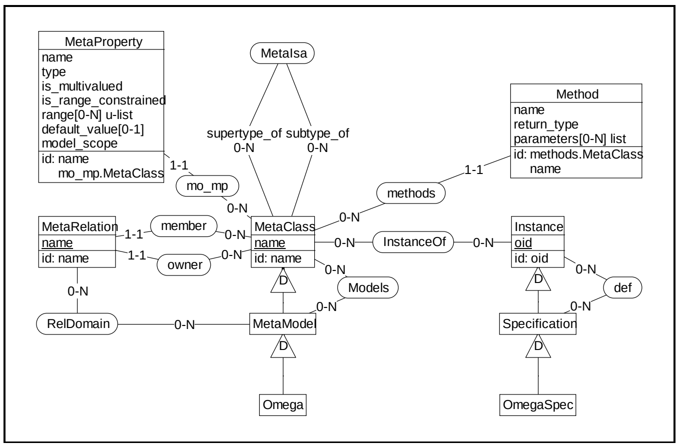


Fig. 1. **Meta-Meta-Model** This E/R schema shows the definition of our meta--meta-model. Technical details have been erased for pedagogic reasons. The `Models` and `RelDomain` relationships denote the possible constituents of a meta-model. The meta-relation instances are not shown, they have been hard-coded for efficiency purpose.

meta-model that would be a specialization of the `E/R` meta-model. Then, the tool would permit us to "project" temporal ER schemata to a "normal" `E/R` meta-model by removing all the concepts that do not belong to the definition of the `E/R` meta-model definition.

The E/R schema[4] 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`, `Specification` and `OmegaSpec`). The `Omega` entity 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 *lasso* (dotted and thick lines) shows the meta-model definition.

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 resp. 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`)

[4] We use the graphical notation of the DB-MAIN case tool (`http://www.db-main.be`). Triangles denote specialization relationships, and `P,D,T` letters stand for **P**artition, **D**isjoint, and **T**otal.

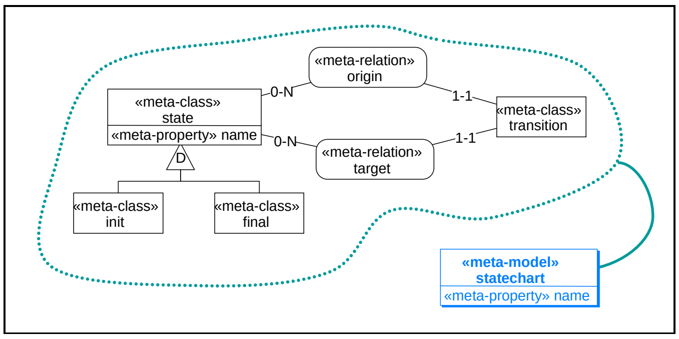


Fig. 2. **Meta-Model** This meta-model describes simple statecharts in terms of concepts of our meta-meta-model. To help the reader to understand this diagram, we have added stereotypes *à la* UML which shows the type of every element. The *lasso* shows the definition of the `Statechart` meta-model.

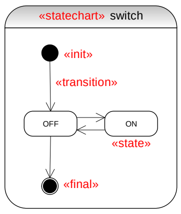


Fig. 3. **Specification** This diagram is a possible visualization of a specification which complies with the `Statechart` meta-model.

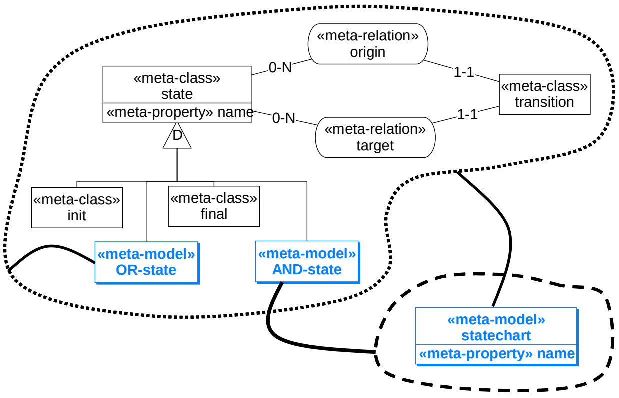


Fig. 4. **Statechart Meta-Model** This schema depicts three meta-models: `OR-state`, `AND-state` and `statechart`. AND-states comprise only statecharts although OR-states may contain every kind of state.

as a subtype of `state`. It is defined just in terms of statecharts. The resulting meta-model is depicted in Fig. 4. Our aim is not to present a brand-new statechart meta-model (the reader can read [2] or [14] for more information), but to show that usual notations can be easily modeled with our approach. Last but not least, our tool is able to exploit those concepts to manage such specifications automatically.

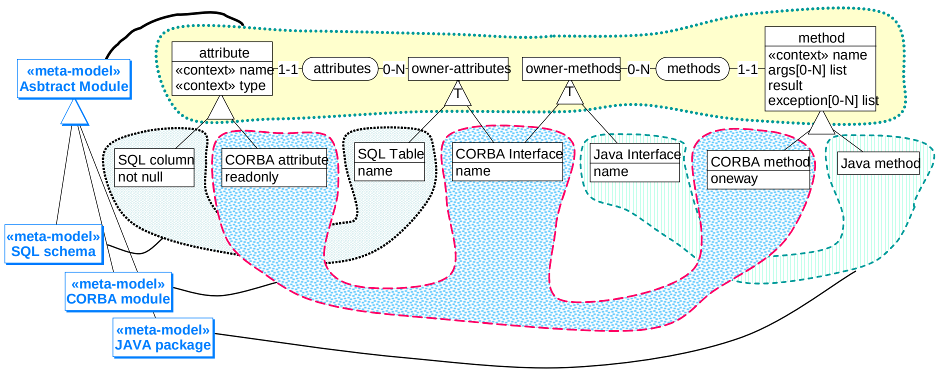


Fig. 5. **Integration of meta-models**

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 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 specifications 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 specification 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. [3] presents a more detailed definiton as well as a comparison with other meta-CASEs. Our aim is to show that it is possible to reach a great expressiveness with just few concepts (meta-class, meta-model, ...). In the scope of this workshop, it is also interesting to compare our framework with other approaches such as GXL[9] — even if their objective are different. Contrary to GXL, nodes can be shared between several graphs with specific information that depends on the graph it belongs

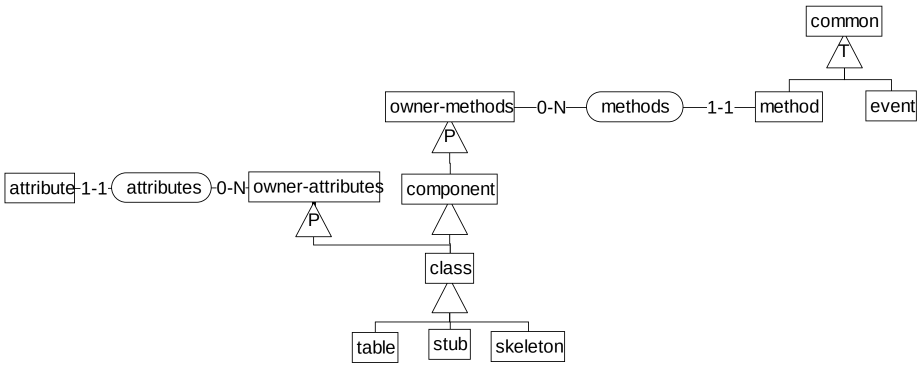


Fig. 6. **Dynamic specialization and inheritance**

to (context attributes, non-disjoint specialization) and can have a behaviour. Nevertheless, our model of "edge" is weaker, we do not propose hyperedge or explosion of edge into graphs, etc. Such features must be emulated.

The next section explains 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 (conceptual, physical, analysis, design, ...) [7,13]. Many efforts have been done to identify or to formalize such transformations and they constitute a real added value for CASE tools [6,13]. The OMG has recently placed this principle at the heart of its MDA (Model Driven Architecture) policy. But in many cases, transformations pose a problem to ensure the traceability between origin and target specifications, since generally either the specification is really transformed and the argument specification is destroyed [5] or we preserve the argument without keeping the links between them [6].

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 been 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

[5] Unless some kind of versioning mechanism has been activated [10].

[6] Some tools keep a journal that makes it possible to retrieve this link a posteriori [16].

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 specification 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 state-chart in creating a common supertype. Moreover, the meta-CASE can exploit this semantics to automatically propose tools to help the engineer in his task (i.e. to transform).

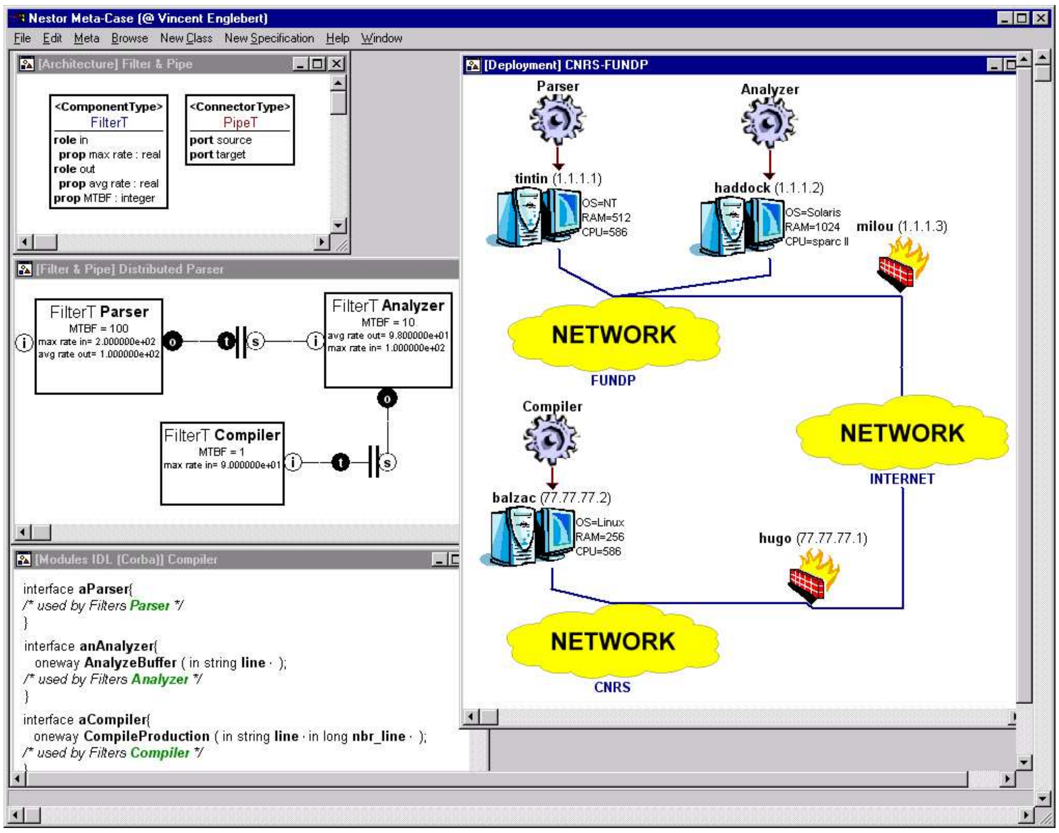
So with a simple magic wand, the traceability is automatically maintained and the life cycles of these objects are synchronized. Of course, this mechanism is not the universal panacea. Many transformations are beyond this principle, but we believe that for simple transformations (and more particularly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

4 Architecture

Although this article is focused on the repository, a presentation of the general architecture of our tool is necessary to understand its main strong points. The repository only describes the static part of a meta-model (as well as some constraints such as identifiers), we have not yet a specific language to describe advanced integrity constraints. Nevertheless, the tool is endowed with a programming language (Voyager II+) that permits us to write predicates, triggers, or any other program (import/export, checking, ...). Voyager II+ is a modular Pascal-like language endowed with garbage collected lists, declaratives queries, a lexical analyzer and meta-facilities.

Another key component in our architecture is the Grasyla (Graphical Symbolic Language) interpreter. The crucial idea behind this language is: whatever is the way the method engineer has defined a meta-model, it should be possible to define Grasyla rules to present specifications according to the requirements of another engineer or some standard. All the concurrent works lack this feature — they generally associate a meta-class with the concept of "visual node". This make their meta-models more sophisticated because they must reflect in some way visual criterions or requirements. In our approach, engineers can specify sophisticated views (such as the UML notations) by writing less than 50 lines of declarative code. The rules can be edited even while the tool is running.

Finally, the tool presents the meta-meta-model as a simple meta-model that acts as a proxy for the first one. This characteristic permits us to bootstrap all the tools that are required to edit/extend/visualize the meta-meta-model.



```
interface aParser{
/* used by Filters Parser */
}
interface anAnalyzer{
  oneway AnalyzeBuffer ( in string line · );
/* used by Filters Analyzer */
}
interface aCompiler{
  oneway CompileProduction ( in string line · in long nbr_line · );
/* used by Filters Compiler */
}
```

Fig. 7. **Case study.**

5 Case Study: modeling distributed architectures

Modeling distributed architectures is a crucial need and most methodologies still neglect this aspect [6]. Indeed, we find either very informal descriptions (UML devotes only a few pages to that [14]) or more formal description such as the Architecture Description Languages (ADL [12]), but that last approaches are mainly a posteriori techniques to model and 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 to define component types (for instance pipes and filters), components (an analyzer, a parser and a semanticizer) as well as their interfaces and their deployment on the physical architectures (network and nodes).

The four specifications 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 [5] for this purpose – the "`distributed parser`" window) to IDL interfaces (the "`compiler`" window) that are next deployed on nodes borrowed from the specification of some physical network (the "`CNRS-FUNDP`" window). The concept of "compiler" is thus unique (i.e. a single node) but has several specializations. All these views are obviously synchronized, since they have been obtained by specialization — the kind of transformation we wanted to illustrate.

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. Meta-modeling experiments have been done with database models, statecharts, ADL constructs, security models, organizational structures, etc. Moreover, the dynamic generalization of meta-models into meta-classes allows one to treat homogeneously advanced mechanisms such as refinement and explosion processes. A prototype has been developed in C++ and the reader can view several screenshots on our site[7].

References

[1] Alderson, A., *Meta-CASE technology*, in: A. Endres and H. Weber, editors, *Software Development Environments and CASE Technology*, number 509 in LNCS (1991), pp. 81–91.

[2] Ebert, J. and R. Süttenbach, *An OMT metamodel*, Fachberichte Informatik 13/97, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz (1997).

[3] Englebert, V., "A Smart Meta-CASE: Towards an Integrated Solution," Ph.D. thesis, The University of Namur, Computer Science Dept. Rue grandgagnage 21. 5000 Namur. Belgique (2000).

[4] Englebert, V. and J.-L. Hainaut, *DB-MAIN: A next generation meta-CASE*, Information Systems **24** (1999), pp. 99–112, special issue on meta-modelling and methodology engineering.

[5] 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.

[6] 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.

[7] `http://www.info.fundp.ac.be/~ven/screenshots`.

[7] Hainaut, J.-L., *Specification preservation in schema transformations – Application to semantics and statistics*, Data & Knowledge Engineering **16** (1996).

[8] Hainaut, J.-L., V. Englebert, J. Henrard, J.-M. Hick and D. Roland, *Database reverse engineering : from requirement to CARE tools*, Journal of Automated Software Engineering **3** (1996).

[9] Holt, R. C., A. Winter and A. Schürr, *GXL: Towards a Standard Exchange Format*, Fachberichte Informatik 1–2000, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz (2000).

[10] 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.

[11] Kelly, S., K. Lyytinen and M. Rossi, *MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment*, in: P. Constantopoulos, J. Mylopoulos and Y. Vassiliou, editors, *Proceedings of the* 8^{th} *International Conference CAiSE'96 on Advanced Information Systems Engineering*, LNCS **1080** (1996), pp. 1–21.

[12] 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.

[13] Milièev, D., *Automatic model transformations using extended UML object diagrams in modeling environnements*, IEEE Transactions on Software Engineering **28** (2002), pp. 413–431.

[14] Object Management Group, *OMG unified modeling language specification*, Technical Report Version 1.4, Object Management Group (2001).

[15] Parallax Software Technologies, "GraphTalk 2.5 Refrence Manual," (1994).

[16] Roland, D., J.-L. Hainaut, J. Henrard, J.-M. Hick and V. Englebert, *Database engineering process history*, in: *Actes du deuxième workshop international sur les différentes facettes de l'ingénierie des processus (MFPE'99)*, Gammarth, Tunisie, 1999.

[17] Sorenson, P. G., P. S. Findeisen and J. P. Tremblay, *Supporting viewpoints in Metaview*, in: *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints'96) on SIGSOFT'96*, San Francisco, CA, USA, 1996, pp. 237–241.

[18] Uhe, I., J. Ebert and R. Süttenbach, *Meta-CASE Worldwide*, Technical Report 24/98, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz (1998).