Pergamon

Information Systems Vol. 19, No. 4, pp. 1–54, 1994
Copyright© 1994 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0306-4379/94 $7.00 + 0.00

0306-4379(94)E0004-9

DB-MAIN: A NEXT GENERATION META-CASE

VINCENT ENGLEBERT AND JEAN-LUC HAINAUT

Computer Science Department
University of Namur
Rue grandgagnage 21
B-5000 Namur

Email: {ven,jlh}@info.fundp.ac.be

(Received 18 June 1993; in final revised form 1 November 1993)

Abstract — This paper describes the DB-MAIN meta-CASE architecture that attempts to conciliate seemingly contradictory goals concerning efficiency, ergonomics, evolution, reuse and ontologies integration. The architecture is based on two layers, the first of which includes built-in basic concepts and functions that should be part of every Information System oriented CASE tool. The second layer comprises four meta-mechanisms intended to reuse, extend and specialize the basic constructs: ontology definition (through a dynamic repository), graphical interface definition, functional definition and methodology definition. These mechanisms are activated through four cooperating languages that allows the CASE engineer to customize the DB-MAIN environment from merely adapting the graphical interface to defining new modeling domains. The paper shows how these mechanisms meet a hierarchy of requirements and how they compare with the current state of the art.

Key words: Meta-CASE, CASE tool, graphical visualization, dynamic functionalities, meta-modelisation.

1. INTRODUCTION

The functions provided by current commercial CASE environments generally fall into seven categories: *a)* collecting, storing and managing specifications *b)* querying, visualizing and exploring specifications *c)* generating reports and code *d)* exchanging with their environment *e)* transforming specifications according to different abstraction levels and paradigms *f)* evaluating specifications against definite criteria and *g)* methodological control and guidance. These functionalities are defined in terms of four major components: *1)* their meta-models (ontology and repository) *2)* their interface (specification representation and control) *3)* their functions/processes (transformations, evaluation) and *4)* their methodologies (the reasoning and activity guidelines). Therefore, developping a meta-CASE entails the modelisation of these four concept domains that will serve as arguments to generate a dedicated CASE tool. Moreover, users expect the meta-CASE technology underlying the generated CASE tool to allow them to incrementally and dynamically extend this CASE tool whenever their needs evolve.

Meta-CASE technology exists for several years and products like ConceptBase [14], Kogge [4], MetaEdit+ [15], MetaView [7, 22] or RAMATIC [20] can be considered as the state of the art in this area[†]. They all propose the modelisation of the four components described above, at least to some extent. They also have enough industrial strength to be compared with some hand-written CASE tools or with component thereof.

1.1. *Requirements Analysis*

We can classify the requirements for user customization according to the four CASE components mentioned above, and by increasing degree of complexity as far as meta-CASE building is concerned.

[†]Some meta-CASE's were omitted because references were too old or because their description was too vague to allow a comparison (ToolBuilder/Lincoln Software, Paradigm/Platinum Technology, GraphTalk/Parallax System, …

At the *interface level*, the user needs customized representations of the specifications stored in the repository. For instance, a conceptual database schema can be displayed according to the ERA, OMT or UML graphical conventions.

Customizing the *meta-model level* allows the *CASE engineer* to modify or to extend the modeling scope of the tool. It can involve four degrees of extension, ranging from adding new properties to an existing meta-class to define a new meta-model. In this discussion we consider that any meta-model comprises meta-classes, relationships and properties.

Adding a property to a meta-class. Existing classes and relationships are left unchanged, but properties are added. For instance, new properties *Owner* and *ManagementCost* can be associated with class *EntityType* to allow analysts to specify the owners and the management cost of each data unit.

Adding a new class or relationship. An existing meta-model is extended to cover the modelisation of new concepts. For instance, a processing meta-model that already comprises class *Process* can be extended by the inclusion of the *Actor* class and of the *Perform* relationship linking it to existing class *Process*.

Adding an inter-meta-model relationship. This is a more complex extension since it defines a bridge between two distinct meta-models. A nice example is that of an *include* relationship from the *DataStore* class of a Data Flow meta-model to the *Attribute* class of the Entity-Relationship meta-model.

Adding a new meta-model. This allows a tool to encompass a new modeling domain. Let us consider a tool that includes the ERA conceptual database meta-model. We want to add a new meta-model that represents the security aspects of information systems. This meta-model should include such concepts as *Users*, *Group of users*, *Privileges* and *Operations*, that can be represented by classes and relationships. Since *Privileges* apply on data units, a new inter-meta-model relationship *on* is defined from *Privilege* to *EntityType* of the ERA meta-model.

The *function level* concerns the extension of the processing power of the tool. The simplest functions include input/output processors that generate external data (reports, code, inter-tool information exchange) or that accept data from the environment (code analyzer, natural language analyzer, import function). However, more knowledge-based processors can be developed, such as transformers, evaluators, optimizers, normalizers or specific inference engines. These extensions can be complex and require a rich meta-programming environment.

The *method level* is seldom proposed in current (meta-)CASE tools, since each tool supports an implicit family of methods that cannot be changed easily. A method is dedicated to an engineering activity such as database design, program development, reverse engineering or system integration. It specifies what kinds of specification documents are coped with and which processes can be enacted to transform documents. We will let this component aside in this paper (see [11, 19])

1.2. The Architecture Spectrum

The spectrum of CASE architectures spreads between two extreme technologies, namely completely dedicated CASE tools (the mainstream of the commercial offering) and fully generic meta-CASEs. The first architecture provides no means to extend, or even to merely customize the tool. However, such of-the-shelf tools are ready to use and they generally offer excellent performance since the repository and the functions have been hardwired. The second architecture acts as an empty shell that forces the CASE engineer to specify every aspect of the target CASE tool, including the most common ontologies such as popular data and process meta-models. The performance of such tools often are poor due to the interpretation techniques they are based on. We postulate that the right solution to the user expectations is a trade-off between these two extreme approaches. Accordingly, this paper develops the DB-MAIN approach, an intermediate, two-layer architecture, that uses the best of these extreme technologies.

The basic layer of DB-MAIN appears as a large spectrum OR[†] CASE tool dedicated to information system engineering. It provides developers with generic data and process meta-models through which they can build specifications at all standard levels of abstraction (conceptual, log-

[†]**O**bject-**R**elationship, which encompasses both Entity-Relationship and Object-oriented meta-models.

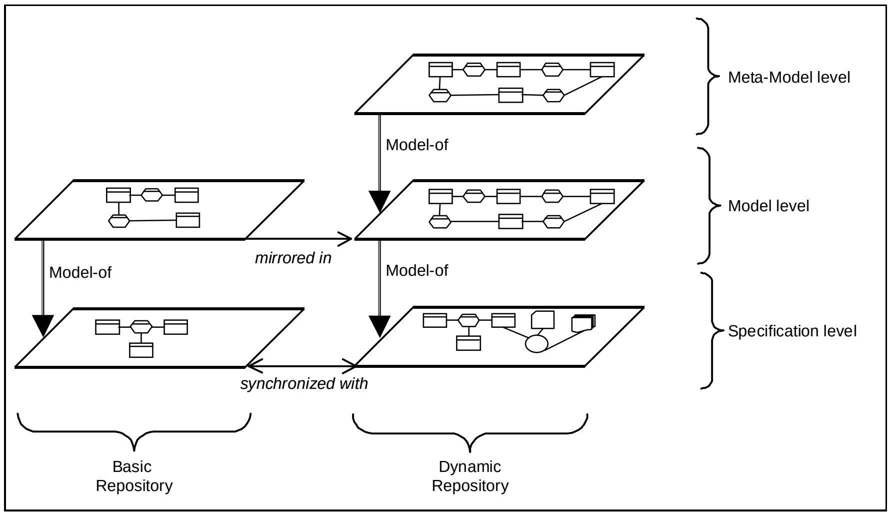

Fig. 1: **[The Basic and Dynamic Repositories]** The left column denotes the usual structure of a CASE tool i.e. its meta-model (upper plane) and its stored specifications (lower plane). The right column denotes the usual structure of a meta-CASE defined with a three levels architecture. The "*mirrored in*" and "*synchronised with*" arrows denote correspondences between "concepts" in the basic and dynamic repositories.

ical, physical, coding) and according to all major modeling paradigms (ERA, OO, NIAM, RDB, standard files, etc.). This layer also includes powerful functions and processors that make the tool invaluable in such complex activities as logical/physical database design, reverse engineering, system evolution, integration and migration [13]. Just like any other CASE tool, it includes a built-in repository (*basic repository*), dedicated functionalities, sophisticated interfaces, and other standard characteristics and functions.

The meta layer of DB-MAIN relies on its own repository (*dynamic repository*) that supports the creation/modification of new meta-models and the common editing facilities of the meta-models instances (the specifications). Although this repository is distinct from the one used in the basic layer, the basic repository is modelled by the meta layer and hence, specifications of the basic layer can be duplicated in the meta layer and both specifications are synchronized. Figure 1 illustrates the use the both repositories with their own abstraction levels.

The motivation for this layered architecture is three-fold.

Firstly, the basic layer is in itself a complete, stand-alone CASE tool that supports forward and reverse engineering activities. Despite its neutrality (it does not comply with any CASE standard, but rather encompasses all of them), it can be used as any commercial tool.

Secondly, one can observe that all the CASE tools currently available, whatever the modeling paradigm they are based on, offer concepts and functions related to data structure and data processing specification. Therefore, providing the CASE engineer with neutral concepts and functions that can easily be tailored to fit a definite paradigm, gives him/her an invaluable help when building a dedicated CASE tool. For instance, the basic DB-MAIN meta-model includes the concepts of *entity type* and *attribute* data objects, which can be mapped to, e.g., object class and attribute in OO meta-models, record type and field in standard file and DBTG structures, table and column in RDB or message and field in communication meta-models. As another example, the concept of *processing unit*, consuming and producing data objects, can model a workflow process, a programming module, an object class method, a trigger or a C++ statement. On the functional side, DB-MAIN includes a rich transformational toolset that can be used to process any data structure specification, whatever the model in which it has been expressed [12].

Finally, the meta layer can use all the investments made for the basic layer. All the processors dedicated to the basic layer are available in the meta layer "for free". For instance, a new CORBA IDL generator or a new PL/1 analyzer that would be developped for the basic layer would automatically become a functional part of the meta layer.

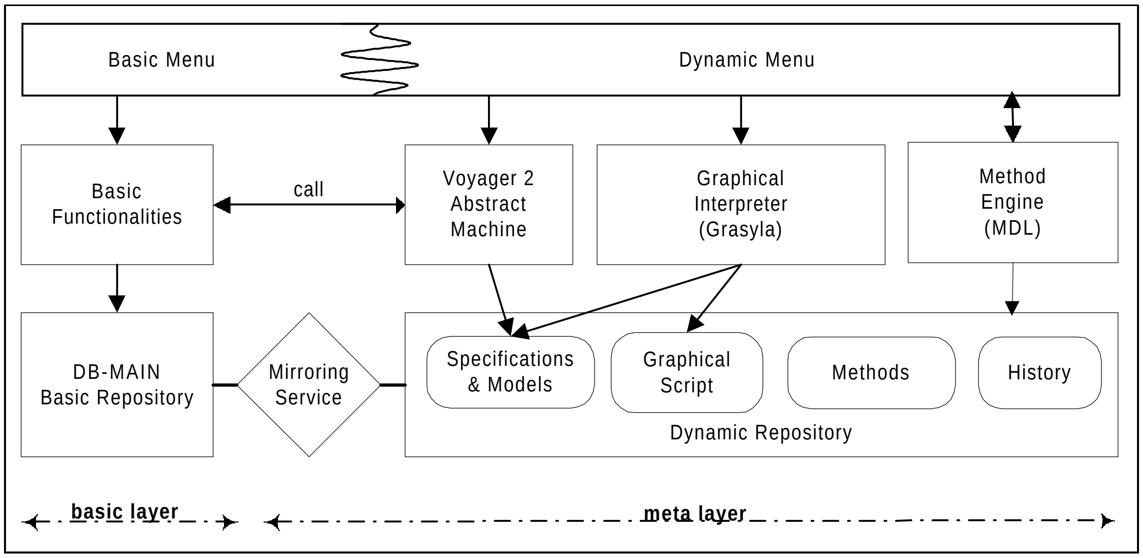

Fig. 2: The global architecture of the meta-CASE tool.

1.3. Organization of the paper

The development of the meta-CASE components was guided in order to answer the requirements explained in section 1.1. Each component will be presented after a general presentation of the meta-CASE architecture that describes both the basic and the meta layers (section 2), section 3 presents the major concepts defined in the repository, section 4 sets out the Grasyla language that permits CASE engineers to define complex interfaces in a very simple way. The Voyager 2 functional language is described in section 5. This language allows the CASE engineer to define new functionalities as well as complex integrity rules. Section 6 explains how our definition of inheritance can be used to integrate meta-models. Evolution problems are adressed in section 7. We close with a conclusion of the main ideas presented in this paper.

2. THE META-CASE ARCHITECTURE

The DB-MAIN tool is composed of two fully integrated layers: the basic and the meta layers. Its general architecture is depicted in figure 2. Each layer has its own repository. We name these repositories respectively *basic* and *dynamic*. The basic repository is written in `C++`, is fast and proposes built-in and advanced functionalities that are common to most OR meta-models. The dynamic repository is generic, extensible and stores both meta-model definitions and application domain specifications (i.e. meta-model instances). The structural part of the basic repository is mirrored in the dynamic part and is maintained by the *mirroring service*, in such a way that each basic object belongs in a seamless way to the dynamic repository as well. The CASE engineer can reuse this image in the dynamic part to use, to extend or to specialize the meta-model built in the basic repository. He can of course define his own meta-model independently of the basic meta-models.

The meta-layer of the tool includes three machines: the Voyager 2 abstract machine, the Grasyla[†] interpreter and the MDL[‡] engine. The first one executes Voyager 2 programs (see section 5), and accesses the dynamic repository (and thus transitively the basic part as well). Voyager 2 programs can call built-in functionalities offered in the basic layer such as: specifications management, generators, transformations, metrics as well as assistants/wizards dedicated to OR schemas processing. Futhermore, the basic layer can itself use the Voyager 2 abstract machine to offer dynamic functionalities above its basic repository such as in some advanced transformational toolkits

[†]**Gra**phical **Sy**mbolic **La**nguage.
[‡]**M**ethods **D**escription **L**anguage

or some reverse engineering assistants. The second abstract machine is the *graphical scripts interpreter* (see section 4) that displays graphical representations of stored specifications. This abstract machine also manages the contextual menus attached to each meta-class/meta-model. The DB-MAIN menu is composed of both basic items and dynamic items that can be customized depending on the context (meta-model, methodology).

As stated in section 1.2, this architecture offers important advantages such as: efficiency, standardization of the common and shared meta-models, easier maintenance and evolution. Reuse is also a major benefit.

The meta-CASE tool has four definition languages: the concepts definition language (to describe both the meta-classes and the classes), the graphical representation language (Grasyla), the functional language (Voyager 2), and the method definition language (MDL). The latter is out of the scope of this paper, but an overview of this language has been presented in [19].

3. META-MODELS AND REPOSITORY

The repository definition was guided by user concerns and the main considerations from earlier experience with the basic DB-MAIN CASE tool. When building a meta-CASE tool, architects design a meta-meta-model. Such a design is a compromise between simplicity and semantic richness [24]. Excessive simplicity prevents the tool to automate tasks[†] that could be derived from the semantics. On the other side, rich meta-meta-models have more complicated semantics and are much more difficult to implement. When defining the meta-meta-model, our motto was: "*Keep its semantics as simple as possible as far as we do not loose crucial advantages — i.e. automated tasks*". Hence, the concepts[‡] proposed in our meta-meta-model are also present in many other tools. The main differences are the generalization of a meta-model as a meta-class and the support of dynamic inheritance.

The main constructs of the repository are depicted in the schema of figure 3, expressed in some kind of OR meta-model. Elements of the population of entity-types `meta-model`, `meta-class`, `meta-relationship` and `meta-property` are respectively called `meta-models`, `meta-classes`, `relationships` and `properties`. So, for instance, the *data flow diagram* meta-model will be an instance of a `meta-model` and the *datastore* in this meta-model will be called a `meta-class`. Instances of these concepts will be called respectively `specifications`, `classes`, `relations` and `attributes`. This example illustrates the use of the previous concepts:

$$\texttt{``\#31XII1999''} \mapsto \texttt{Orders} \mapsto \texttt{DataStore} \mapsto \texttt{meta\_class}$$

where the $\mapsto$ symbol denotes the *instance-of* relationship.

A meta-model is defined in terms of the meta-classes belonging to its ontology. If m is a meta-model, then we note $\Delta(m)$ the set of meta-classes that define it. One meta-class can belong to several meta-models. Meta-classes have a unique name and can have properties. When a meta-class is *virtual*, it depends on the existence of at least one subtype (cfr. section 6 for more information on the inheritance).

Each property has a type (integer, real, char, string, boolean, bitmap, sound) and can be single/multi valued. Meta-classes can participate in named one-to-many relationships. Inheritance hierarchies can be defined amongst meta-classes. Multiple and dynamic inheritance is allowed.

One notes Γ:`meta-class`$\mapsto$`class` the materialization function. This function is represented by the relationship `materialization` in figure 3.

The *Entity-Relationship Schema* and *Database Security Diagram* (DSD) meta-models will be used to illustrate the definitions given above . The OR schema of the DSD meta-model is shown in figure 4 and the definition of the meta-models is depicted in figure 5. There are two meta-models, each one has a definition composed of meta-classes and they share both one meta-class: *`data_object`*.

A possible materialization could be:

[†] i.e. integrity constraints, triggers, dialog boxes, explosion/decomposition, ...

[‡] meta-classes, binary meta-relationships, inheritance, simple/multivalued meta-properties and meta-models

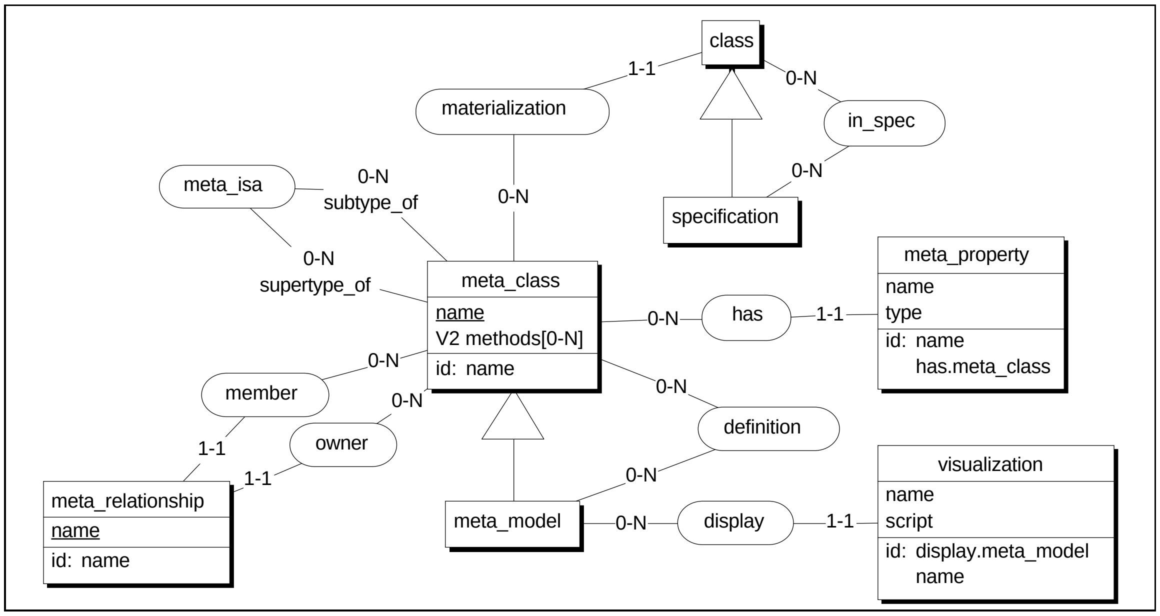

Fig. 3: **[The Meta-Meta-Model schema definition]** △ symbols denote **isa** relations.

Γ(*entity-relationship schema*)= { *Order schema*, *Stock Conceptual* }

Γ(*DSD meta-model*)= { *Order Security*, *Global Security* }

Γ(*data_object*)= { *customer*, *order*, *product*, *warehouse*, ...}

Γ(*individual*)= { *bond*, *columbo*, *kojak*, *maigret* }

Γ(*operation*)= { *all*, *create*, *delete*, *update*, *read*, *copy*, *open* }

Since a meta-model is itself a meta-class due to the ISA-relationship between `meta-model` and `meta-class`, it inherits all the meta-class properties. A meta-model can thus have properties, take part in relationships[†] with meta-classes (or meta-models) or inherit from other meta-classes (or meta-models). Since a meta-model is defined in terms of meta-classes, a meta-model (i.e. a meta-class) can belong to its own definition. This permits to represent the "*becomes*" relationship between a specification and its explosion/refinement as described in [9] and [15].

Each specification is defined by a set of classes belonging to the meta-classes that define the meta-model of the specification. Distinct specifications can share common classes in such a way that classes edited in a specification are necessarily visible in the other ones (several examples are illustrated in the screen shot figure 10).

A meta-model can inherit from meta-classes and, hence, from other meta-models. A meta-model that is a subtype of a meta-class, inherits all its properties and roles, a meta-model that is a subtype of a meta-model inherits all its properties, its roles and its definition. Let us suppose that meta-model M inherits from meta-models M_1, ..., M_n, then M is defined by the union of the definitions of each M_i and its own definition.

A meta-class can have methods (i.e. Voyager 2 procedures/functions) that define the behaviour of its instances. Some predefined methods, such as the following, specify the default behaviour of the classes:

`CanCreate()`$\rightarrow$`Boolean`: Is it permitted to create an instance of this meta-class.

`Precondition()`$\rightarrow$`Boolean`: Are the preconditions fulfilled?

`CanDelete()`$\rightarrow$`Boolean`: Can the user delete this instance?

`OnDelete()`: The action to be carried out when the current class is deleted.

[†]This possibility permits to make explicit relationships between meta-models as it is suggested in [17].

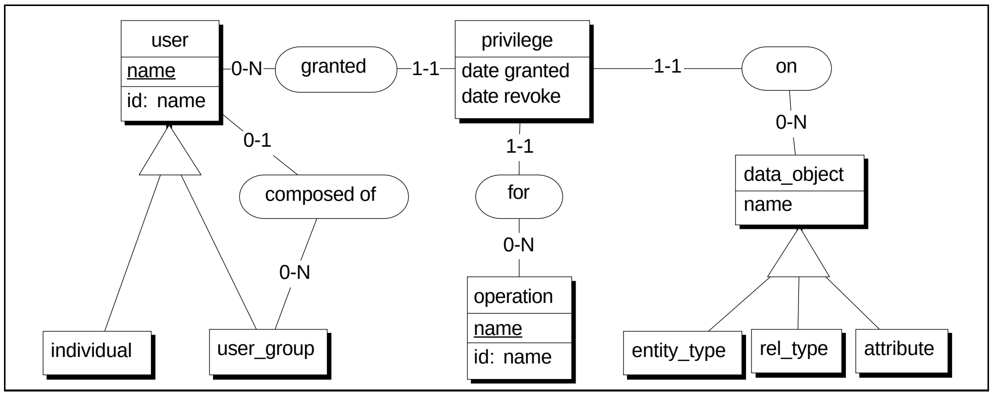

Fig. 4: **[Database Security Diagram Meta-Model]** This schema modelizes the right accesses of users to database objects such as *entity-types*, *attributes* and *rel-types*. A *user* is either a *user_group* or an *individual*. A group is itself composed of users. Users can have *privileges* on some *database objects* for specific *operations*. The `data_object`, `entity_type`, `rel_type` and `attribute` meta-classes are defined in the DB-MAIN basic layer.

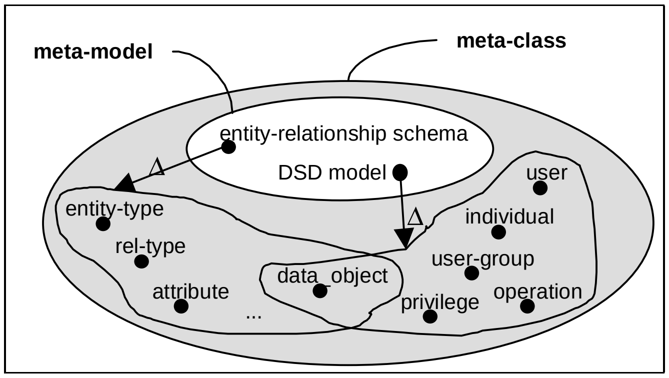

Fig. 5: **[Venn's diagram illustrating the DSD and ER meta-models]** Venn's Diagram denoting the population of the meta-class and meta-model entity-types of the repository. The meta-model set is a subset of the meta-class set. Arrows annoted with Δ denotes the mapping Δ from meta-model to the power set of meta-class.

The user can overload or refine these predefined methods. Others can also be defined in order to endow each meta-class with intelligence. Validation, transformation, metrics and export methods are common uses. Each user method can be executed from a contextual menu.

In the same way, predefined methods exist for all the transformations available in the DB-MAIN basic layer. These methods allow the CASE engineer to define evolution rules for new meta-models depending on concepts that can be affected by transformations. For instance, splitting entity type E into two fragments requires the distribution of the privileges that concern E.

Finally, each meta-class can have an identifier composed of single or multivalued meta-properties and roles. There are two kinds of identifiers: *local ID* and *global ID*. Let C be a meta-class (resp. meta-model) and x_1, x_2 be two classes (resp. specifications) $\in \Gamma(C)$. If C has a local ID, then if x_1 and x_2 belongs to one specification s, they must have distinct values wrt. the properties/roles that form the ID. If C has a global ID, then x_1 and x_2 must have distinct values wrt. the properties/roles found in the ID whatever the specifications they belong to. If C is a virtual meta-class, then C can have no explicit ID, and the identifier can be looked for in the subtypes. Otherwise, IDs are inherited from supertypes.

Identifiers are not required by the repository technology but are offered both for methodological issues and to provide textual denotation of classes. For instance, an entity type can be designated by its name (local ID), a SQL column by its name and the table it belongs to (local ID), and a software engineer by its first and last names (global ID).

Although the repository power may seem less expressive than in concurrent meta-CASEs, complex concepts like polymorphic objects and relationships [8] (for instance) can be transformed into the concepts explained above without loss of expressiveness.

4. THE GRAPHICAL ENVIRONMENT (GRASYLA)

The graphical representation of meta-models is a crucial issue. Indeed, as underlined by Findeisen [8], "*[...] the final result was sometimes lacking user-friendliness*" and "*Many of the difficulties encountered by the user are associated with the graphical representation of the SDM*[†] *and the consistency constraints*". Our experience lead us to consider two kinds of specification visualization: graphical diagrams (graph, tree, table, matrix, ...) and textual views (report, code, ...). Software enginers often require several views of a same specification depending on the process to complete. Graphical views are prefered for teaching or for validation although textual views[‡] are prefered to edit huge specifications. Meta-CASE tools have thus to offer these views.

Some tools propose either a sophisticated definition language (MetaView's EDL/GE [10]) or a graphical editor (MetaEdit+) to define the shape of the concepts and generally propose script languages to generate the textual views. MetaView also has a constraint language to force compliance of diagrams with rules[§] and Hardy [21] proposes hypertext facilities.

In the DB-MAIN approach, each meta-model (and thus each specification) can have several graphical representations (cfr. the "`visualization`" entity-type in the repository of figure 3). Each one is defined by a Grasyla script (set of symbolic statements) explaining how to represent meta-classes (and the properties) in terms of their characteristics (properties, super-types and roles). Each statement can be preceded by an identifier/functor so that meta-classes can have distinct graphical representations inside a diagram. Hence, the *display processor* (DP) is governed by a set of equations. When displaying a class, the DP uses the "best" equation describing the look of its meta-class. Grasyla can be used to define graphical and textual views[¶]. Although the Grasyla semantics is very simple, advanced possibilities are proposed such as: colors, fonts, handles, arrows, various shapes, bitmaps, video, sound, graphical alignment and aggregated forms.

Each meta-model has a default DP to place and build the representation of their meta-classes. The behaviour of this default DP suits graph-like specifications very well. However, some views require special algorithms that can not be modeled directly in Grasyla: Matrices, Tables, Browsers, Sequence Diagrams, Screen Layouts, ... For this reason, the meta-CASE architect can implement hard-wired graphical processors dedicated to some meta-models acting like *meta-model-patterns*. This meta-model pattern can be specialized into other meta-models with their own graphical statements. Hence, the hard-wired DP will use user-defined statements to display the specifications.

Let us examine the DSD meta-model (cfr. figure 4). Users are denoted by small bitmap icons topping their name, groups of users are represented by boxes comprising their composition tree. Each privilege is displayed as a labelled node linking a user with a data object. Figure 6 illustrates the graphical representation of a DSD specification excerpt. Items were positioned by the user. The figure expresses facts such as: *a)* colombo and kojak are people, and form the employees group, *b)* group staff is made of groups managers and employees, *c)* members of group managers are allowed to read attribute address and *d)* members of group staff are allowed to read entity type customer and to delete entity type order.

The Grasyla script advicing the display processor on the way to display each concept (*users*, *user_groups*, *data_objects*, *privileges*, ...) is reproduced in figure 7. The general pattern of equations is "*\$ concept = symbolic expression describing how displaying the concept in terms of its properties and its roles*".

The graphical language is powerful enough to bootstrap a complete meta-model editor in about twenty lines. The graphical language meets important requirements such as complex graphical notations and multiple views. This problem has been addressed in [1]. Some advanced views are shown in the screen shot illustrated in figure 10.

[†]Software Development Model.

[‡]Textual views can have facilities like: sort algorithms, hypertext, ...

[§]For instance: a supertype must be placed above its subtypes.

[¶]Textual views defined with Grasyla look like a text but are still graphical objects that are distinct of ascii characters. Reports must be generated by Voyager 2 programs.

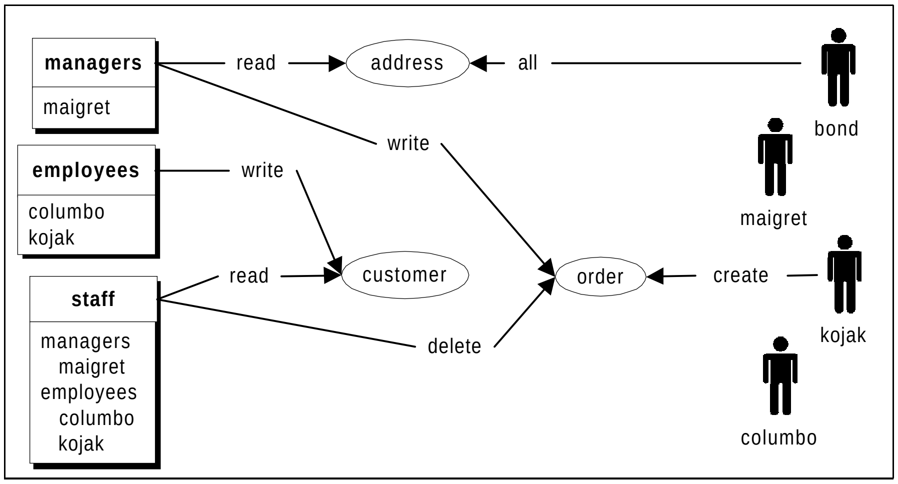

Fig. 6: Graphical view of a DSD specification.

5. THE META-LANGUAGE (VOYAGER 2)

Many programs owe their success story to their capability given to users to add new functionalities. EMACS and AutoCAD[†] are such programs. To our knowledge, KOGGE is the only meta-CASE offering a complete meta-language (KOGGE-modula) to give access to its repository. As to MetaEdit+, it proposes interfaces with foreign languages like `C++` or SmallTalk.

DB-MAIN is endowed with a meta-language (Voyager 2) [6]. Its concepts are sufficiently clean and simple to teach it in a two-day seminar. Voyager 2 is a C/Pascal-like language and is thus complete. So far, about one hundred medium to large applications have been written in Voyager 2 such as: COBOL/SQL/CODASYL/C++/O2 generators, IMS/COBOL/SQL extractors, RTF file and report export functions, transformation toolkits, metrics and statistics programs.

The main paradigms found in Voyager 2 are: • *Elementary types.* Besides the classical types (integer, real, char, string, file), Voyager 2 supports lists as first-class values[‡]. Lists are automatically "garbage-collected" by the abstract machine. Special operators are provided such as `+`[§], `*`[¶], ... and there is no restriction on the use of lists. The `lambda` type allows a program to load functions at the execution time, to store them in lambda-variables for a further use. • *Lexical analyzer.* Programs can read a text file one token at the time. Tokens definitions can be specified at the execution time. • *I/O statements.* • *GUI functions.* • *Procedure and function definition* with recursive calls. • *Dynamic linked libraries* • *Communication with Windows programs.* • *Weakly typed language.* Programs can dynamically create (or delete) meta-classes and, next, create new instances of these ones. The type-checking is thus delayed until the execution time. • *Meta-Homogeneity.* The meta-meta-model can be queried in the same way than a meta-model.

Voyager 2 has special constructs to query the repository . Simple queries are based on this pattern: *Let R be a relationship, then find the list* $[b_1, \ldots, b_n]$ *such that each* b_i *plays a role of R with classes of a list* $[a_1, \ldots, a_m]$ *and satisfies some boolean user-defined expression.* Since queries use lists and return lists, they can be fit together and be mixed with list dedicated operators (`*`, `+`) to form complex queries in an easy way.

Voyager 2 is used to extend the standard functionalities (editing) as well as to define complex integrity rules. It can be used to develop complex programs or to define methods and standard predicates/triggers attached to meta-classes. Some methods can be executed from a contextual menu.

[†]AutoCAD is a trademark of Autodesk

[‡]Example: `[1,2,3]`, `[[1,"one"],[2,"two"]]`, `[5,[3,[1],[4]],[8,[6],[10]]]` are syntactically correct lists in Voyager 2.

[§]i.e. list concatenation

[¶]i.e. list intersection

```
01 root :  user, data-object, privilege ;
02 $user_group = boxV { boxH { handle spring bold { $name } spring handle }
03                      ruleH
04                      boxH { in_group($compound_of) spring }
05                    } with { frame=shadow color=black }
06 $individual = boxV { boxH { handle spring bitmap("/draw/man.bmp") spring handle }
07                      boxH { spring $name spring }
08                    }
09 $privilege = boxV { boxH { spring handle spring }
10                     boxH { handle spring $about spring handle }
11                     boxH { spring handle spring }
12                   }{ frame=simple color=black }
13 $operation = $name
14 $data_object = ovalH { handle $name handle }
15 in_group( list $user ) = boxV { $head in_group( $tail ) }
16 in_group( $individual ) = boxH { $name spring }
17 in_group( $user_group ) = boxV { boxH { $name spring }
18                                  boxH { horiz(10pt) in_group( $compound_of ) spring }
19                                }
20 $granted = connect{void}{void}{color=black width=2pt}
21 $on = connect{format=arrow color=black}{void}{color=black width=2pt}
```

Fig. 7: **[Grasyla Script of the DSD Diagram]** The directive at line 1 expresses that all the instances of the `user`, `data-object` and `privilege` meta-classes must be displayed. Statements at lines 2, 6, 9, 13, 14, 15, 16 and 17 define how displaying instances of the corresponding meta-classes in terms of geometrical forms (box, oval, line, ...), letters and their characteristics (properties and roles). The assembling of the boxes obey more or less the same principles that in TEX[16]. Statements at lines 20 and 21 define the arcs denoting relations between classes.

To illustrate the use and the semantics of the Voyager 2 programming language, one will use the DSD meta-model defined in figure 4 to implement two new functionalities listed in figure 8. Lines 1–11 denote the (`retrieve_users`) function. The `let` statement (lines 1–4) links identifiers to meta-classes such that they are interpreted as types in the body of the statement. In our example, `Privilege` is a type as `integer` and `string`. The engineer can also associate identifiers with relationships (lines 5–7) and properties. The body of the `let` statement is a function declaration (lines 8–10). The `retrieve_users` function returns a list and takes two arguments (an operation and a data object). It computes the list of all the users having some privileges for this operation on this data object. The `Privilege{@forop:[op]}` query/expression computes the privileges (i.e. a list) about the `op` operation. The `Privilege{@priv_on:[data]}` expression retrieves the privileges defined on the `data` operation. The `User{@granted:...}` expression retrieves the users having the privileges common to the both previous lists (the `*` operator denotes the list intersection). Lines 13–22 denote the overloading of a predefined method[||] that is automatically triggered when `UserGroup` classes are deleted. Line 17 declares a new local variable. The query (line 18) retrieves all the users that compose the group to delete (denoted by `this`), and the loop (line 18) on this list will delete each user. If members of this group are themselves `UserGroup` classes, they will be deleted in the same way.

6. INTER META-MODEL COORDINATION

In meta-CASEs, distinct meta-models can coexist in the same workspace. Each meta-model has its own ontology, its own semantics and graphical standards. Studies like [3] and [23] show that meta-models can have very complicated definitions even when very expressive concepts are used. For instance, the integration of the “Object”, “Dynamic” and “Data Flow Diagram” meta-models in OMT is often achieved by explicit relationships that establish correspondences between “similar” meta-classes (`datastore` $\leftrightarrow$ `class` , `actor` $\leftrightarrow$ `class`, `event` $\leftrightarrow$ `operation`). Each meta-model

[||]The body of this predefined method is empty.

```
let Privilege = meta-class("privilege") and
    DataObject = meta-class("data_object") and
    User = meta-class("user") and
    Operation = meta-class("operation") and
    granted = relationship("granted") and
    priv_on = relationship("on") and
    forop = relationship("for")
in { function list retrieve_users( Operation op,DataObject data ){
       return User{ @granted:Privilege{@forop:[op]} * Privilege{@priv_on:[data]} };
     }
   }

let UserGroup = meta-class("user_group") and
    User = meta-class("user") and
    composed_of = relationship("composed_of")
in { method UserGroup::on_delete()
       User:  usr;
     {  for usr in User{@composed_of:[this]} do {
          delete usr;
        }
     }
   }
```

Fig. 8: User-defined Voyager 2 functions/methods

being assembled as the integration of sub-meta-models and so on. [1] and [2] present nice examples with some guidelines for the integration. Unfortunately these "correspondence" relationships must be managed explicitely by the CASE engineer.

DB-MAIN proposes an implicit integration of *similar* concepts in distinct meta-models through the use of the `isa` relationship. The `isa` relationship between two meta-classes A and B (A `isa` B) is defined as a partial function $\Gamma(A) \rightarrow \Gamma(B)$. Let us suppose that meta-model M_1 (resp. M_2) is defined as $\Delta(M_1) = \{V_1, \ldots, V_m\}$ (resp. $\{W_1, \ldots, W_n\}$) where V_i, W_j are meta-classes. Then if meta-classes/concepts V_k and W_l have similar semantics, one can create common supertype C that factorizes all the common properties/roles of both meta-classes V_k and W_l. Now, let us take two specifications $s_1 \in \Gamma(M_1)$ and $s_2 \in \Gamma(M_2)$, if classes $v \in \Gamma(V_k)$ and $w \in \Gamma(W_l)$ denote the same application domain concept, then one can create a class $c \in \Gamma(C)$ such that v and w are both subtypes of c. These classes are now synchronized by the way of a common parent and each modification operated on v will be propagated to w and reciprocally. Since the `isa` function is partial, some classes could have no counterparts in other specifications.

This mechanism can be illustrated (see figure 9) with the (`datastore`$\leftrightarrow$`entity-type`) correspondence. We could create a common supertype `data` in the OMT meta-model with `datastore` and `entity-type` as subtypes. In the same way, if two instances `d`$\in \Gamma($`datastore`$)$ and `e`$\in \Gamma($`entity-type`$)$ in their respective specifications are similar, then one can create new instance `f`$\in \Gamma($`data`$)$ such that `f` is a common supertype of both `d` and `e`.

7. THE EVOLUTION OF META-MODELS

Schema evolution is a crucial problem in software engineering dedicated applications [5]. A meta-CASE that would prohibit any modification of the meta-model definitions will have the same failing than classical CASE tools. On the other side, schema evolution in OODBMS is still an open problem. Because meta-CASE's have their own meta-model (i.e. the meta-meta-model), meta-model evolution must be addressed at the application level (i.e. the meta-CASE level) and not at the DBMS level. On the other side, meta-CASE's often have rigorous control over the developped applications. For this reason, programs that would crash when run against a modified schema in classical OODBMS environment, would simply cause an exception in the worst case in meta-CASE's environnement.

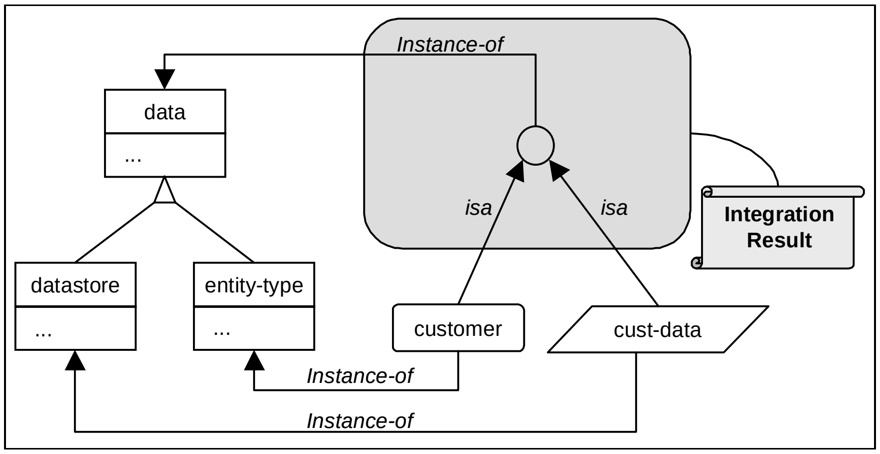

Fig. 9: **[Meta-Model Integration]** Two meta-classes from distinct meta-models are integrated in creating a common supertype. Their instances are integrated in the same way.

In the DB-MAIN approach, CASE engineers can modify the meta-model definitions without restriction. The repository ensures the coherence of the instances. Users can add/remove supertypes, add/delete meta-classes, add/delete relationships, edit the meta-model definition, add/remove/edit Grasyla definitions, . . . Most of these modifications can occur when meta-classes are populated. The interface of Voyager 2 programs with the meta-models and their specification is dynamic. This feature prevents the meta-CASE from propagating the meta-model modifications to the programs. On the other side, when such programs use a bad† meta-model definition, they stop with a warning message.

8. CONCLUSION

The DB-MAIN approach to CASE extendability is based on the idea of conceptual and functional reuse. Every CASE tool intended to help developers in Information System engineering should include basic concepts modeling data objects and process specifications. It should also include basic functions for the management, visualization, transformation and evaluation of IS-related specifications. Therefore, developing a new CASE tool consists in customizing and extending the basic DB-MAIN environment through four mechanisms with which four specific languages have been developed. The latter allow the CASE engineer to augment the modeling domains addressed by the tool, to define the graphical user interface, to develop specific functions and to built methodologies for each engineering process such as analysis, design, reverse engineering, integration. At the current time (version 4), we have experimented the customization of the repository through the addition of properties to basic meta-classes and through the development of Voyager 2 programs and functions. We observed that the DB-MAIN users encounter few problems in customizing the tool. For instance, a sophisticated Sybase generator was developed in 5 days, following a one-day introduction to the repository and Voyager 2 language. However, the other mechanisms will be available in version 5 only. The DB-MAIN meta-CASE is developed in `C++` for MS-Windows workstations.

Acknowledgements: We thank D. Roland, P. Heymans, J.F. Raskin and P. Thiran for their help and comments. We also thank J-M. Hick and J. Henrard, the other architects of the DB-MAIN CASE environment.

REFERENCES

[1] Mike Brough. Methods for CASE: a generic framework. In Loucopoulos [18].

†In this case: a definition that was modified.

[2] Jürgen Ebert and Alexander Fronk. Operational semantics of visual notations. Technical Report 8/97, Fachberichte informatik, Koblenz, http://www.uni-koblenz.de/universitaet/fb/fb4/ (1997).

[3] Jürgen Ebert and Roger Süttenbach. An OMT metamodel. Technical Report Fachberichte Informatik 13–97, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, (1997)., http://www.uni-koblenz.de/universitaet/fb/fb4/publications/GelbeReihe/RR-13-97.ps.gz (1997).

[4] Jürgen Ebert, Roger Süttenbach, and Ingar Uhe. Meta-CASE in practice: a case for KOGGE. In A. Olivé and J. A. Pastor, editors, *Advanced Information Systems Engineering, , CAiSE'97*, number 1250 in LNCS, pp. 203–216, Barcelona, Catalonia, Spain (1997).

[5] W. Emmerich, W. Schäfer, and J. Welsh. Databases for software engineering environments – the goal has not yet been attained. In I. Sommerville and M. Paul, editors, *4th European Software Engineering Conference – ESEC'93*, volume 717 of *LNCS*, pp. 145–162, Garmish-Partenkirchen. Springer-Verlag, ESPRIT-III Project GOODSTEP (6115) (1993).

[6] Vincent Englebert. *Voyager 2. Version 3 Release 0.* FUNDP - DB-MAIN, FUNDP, Rue grandgagnage 21. 5000 Namur. Belgium, http://www.info.fundp.ac.be/~dbm (1998).

[7] Piotr Findeisen. MGED reference manual – release 2.0. Technical report, University of Alberta (1993).

[8] Piotr Findeisen. *The EARA model for Metaview. A reference.* University of Alberta (1994).

[9] Dinesh Gadwal, Piotr S. Findeisen, Paul G. Sorenson, J. Paul Tremblay, and L. Beth Millar. *Generating customizable software specification environment using Metaview.* University of Saskatchewan and University of Alberta (1994).

[10] Dinesh Gadwal, Pius Lo, and Beth Millar. EDL/GE users's manual. Technical report, University of Alberta and University of Saskatchewan (1994).

[11] Georges Grosz, Samira Si-Said, and Colette Rolland. MENTOR: un environnement pour l'ingénierie des méthodes et des besoins. In *INFORSID'96*, pp. 33–51, Bordeaux (1996).

[12] Jean-Luc Hainaut. Specification preservation in schema transformations – Application to semantics and statistics. *Data & Knowledge Engineering*, **16**(1) (1996).

[13] Jean-Luc Hainaut, Vincent Englebert, Jean Henrard, Jean-Marc Hick, and Didier Roland. Database reverse engineering : from requirement to CARE tools. *Journal of Automated Software Engineering*, **3**(2) (1996).

[14] M. Jarke, R. Gallersdorfer, M.A. Jeusfeld, M. Staudt, and S. Eherer. ConceptBase – a deductive object base for meta data management. *Journal of Intelligent Information Systems*, **4**(2):167–192 (1995).

[15] S. Kelly, 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* 8th *International Conference CAiSE'96 on Advanced Information Systems Engineering*, volume 1080 of *LNCS*, pp. 1–21, Heraklion, Crete, Greece. Springer-Verlag (1996).

[16] Donald Ervin Knuth. *The TEXbook.* Addison-Wesley, nineteenth edition (1990).

[17] V. Lalioti and P. Loucopoulos. Visualisation of conceptual specifications. *Information Systems*, **19**(3):291–309 (1994).

[18] P. Loucopoulos, editor. *Advanced Information Systems Engineering*, volume 593 of *LNCS*, Manchester. 4th International Conference CAISE'92, Springer-Verlag (1992).

[19] D. Roland and J-L. Hainaut. Database engineering process modeling. In *International Conference on The Many Facets of Process Engineering*, Gammarth, Tunis (1997).

[20] Matti Rossi, Mats Gustafsson, Kari Smolander, Lars-Ake Johansson, and Kalle Lyytinen. Metamodeling editor as a front end tool for a CASE. In Loucopoulos [18].

[21] Julian Smart and Robert Rae. *Hardy. User Guide.* Artificial Intelligence Applications Institute, 80 South Bridge, Edinburgh EH1 1HN. UK (1996).

[22] Paul G. Sorenson, Jean-Paul Tremblay, and A. J. McAllister. The Metaview system for many specification environments. *IEEE Software*, **5**(2):30–38 (1988).

[23] Roger Süttenbach and Jürgen Ebert. A Booch Metamodel. Fachberichte Informatik 5–97, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, http://www.uni-koblenz.de/universitaet/fb/fb4/ (1997).

[24] X. Wang and P. Loucopoulos. The development of Phedias: a CASE shell. In Hausi A. Müller and Ronald J. Norman, editors, 7th *International Workshop Computer-Aided Software Engineering (CASE'95)*, Toronto, Ontario, Canada. IEEE Computer Society Press (1995).

APPENDIX

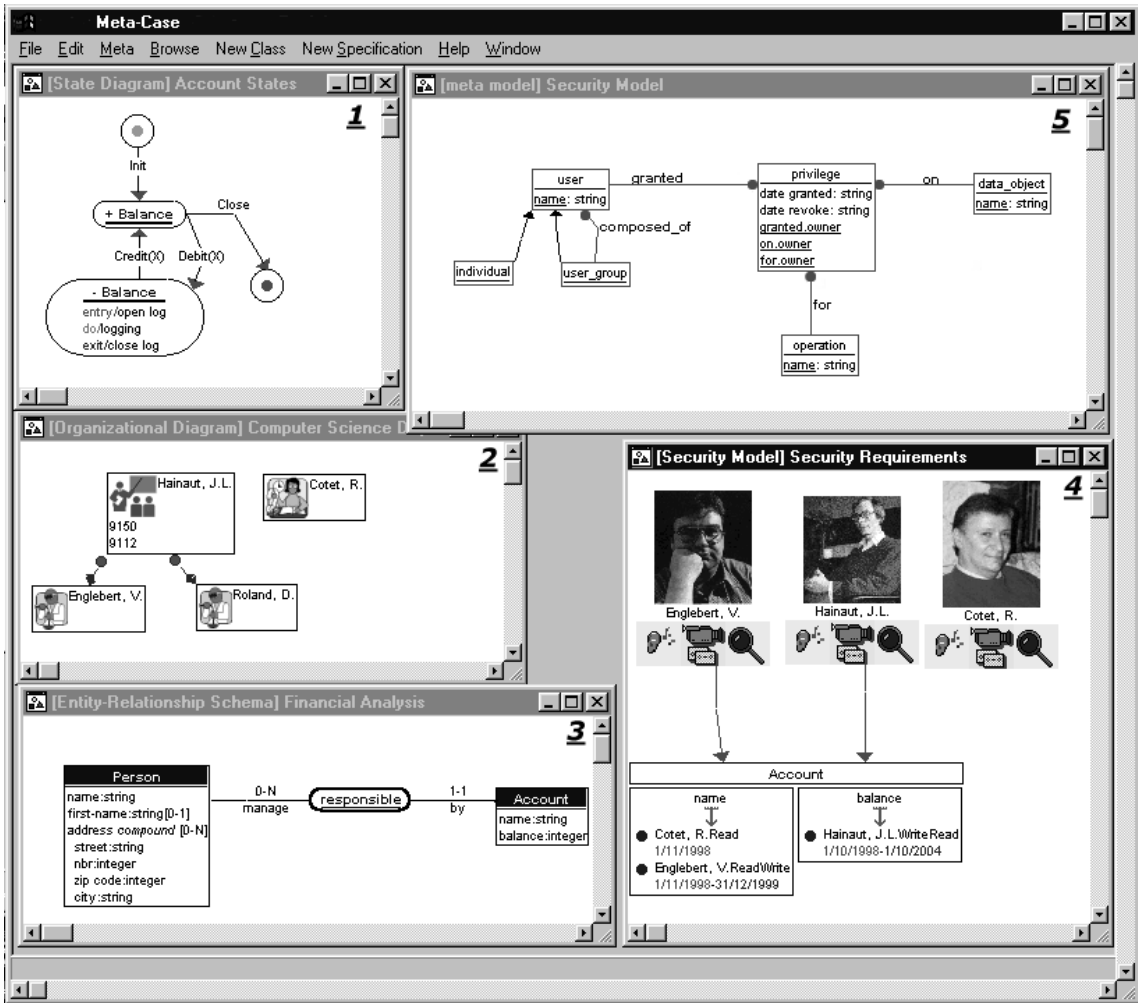

Fig. 10: **[Screen Shot]** Four specifications and one meta-model definition are visualized in this screen shot. Windows denote respectively *1)* a state diagram *2)* an organizational diagram *3)* an entity-relationship schema *4)* a visualization of a DSD specification wrt. a richer Grasyla script than the one described in this article and *5)* the visualization of the DSD meta-model definition. In window *4*, the `account` entity-type is displayed like a table *à la* MS-Access, and each column contains the grants (who, what and when). The arrows denote privileges on the whole table/entity type. Let us remark that the `account` concept is shared by specifications *3* and *4*, and that the `hainaut` person is itself shared by specifications *2* and *4*. Icons (sound/camera/magnyfying glass) are active and a double click on them shows a multimedia document.