Chapter 1

GRASYLA: MODELLING CASE TOOL GUIS IN METACASES

Vincent Englebert
Jean-Luc Hainaut
University of Namur
Rue Grandgagnage 21
B-5000 Namur
Belgium
{ven,jlh}@info.fundp.ac.be
http://www.info.fundp.ac.be/∼dbm

Abstract Meta-CASEs are CASE (Computer-Aided Software Engineering) tool factories. For some years, much effort have been spent in this realm to propose a competitive alternative to the traditional CASE framework. Meta-CASEs now benefit from efficient and rich meta-repositories, they support several methods and are multi-user. However, all current meta-CASEs share the same approach of the Graphical User Interface modelling task. We analyze here the new challenges to take up and we propose a new graphical symbolic language (Grasyla) to model the CASE's GUI.

Keywords: MetaCASE, CASE tool, graphical symbolic language, meta-modelling, Grasyla

1. INTRODUCTION

CASE tools are programs that support software engineers[1] activities during the software life cycle. They can automate some stages (code generation, metrics, model checking, . . .) or help the software engineers to follow/respect some methodologies. Meta-CASEs are high level compilers/interpreters that produce CASE tools that meet specific models and engineering processes. They use an abstract description of a CASE tool (i.e., its models, its behaviour, its processes, . . .) to produce a ready-to-use product. Of course, such products cannot be ex-

[1]Software engineers use CASE tools to build software.

pected to enjoy the same qualities than hand-coded dedicated CASE tools. The current situation exhibits an interesting dilemma. From one side, CASE tools become ineluctable. Indeed, the market is inundated with a plethora of models[2] and paradigms[3], so that programmers cannot keep using mere text editors or vendor-dependent IDE[4]. On the other side, as Lending and Chevarny [11] write: "*Few organizations use CASE tools [2, 16]; organizations abandon the use of the tools [2, 21, 23]; and organizations that do use CASE tools contain many systems developers who do not actually use the tool [14] "*.

Meta-CASEs attempt to reduce the distance between the programmers/analysts needs and the CASE functionalities. Indeed, meta-CASEs allow analysts to define their own models (i.e., meta-models), to visualize their specifications in accordance with their requirements, and to apply their processes (metrics, transformations, report generation, ...). Moreover, this technology is ideal to integrate independent meta-models and can act like an *enterprise's meta-model-warehouse*. Some research [7, 13, 24] concentrated on the knowledge representation aspect and deliberately left aside the graphical representation of the specifications, while other groups [1, 9, 18, 19, 20, 22] investigated the whole problem. In all the cases, the graphical representation plays a crucial role since it is often the only way to visualize and edit the stored specifications. It will also be a major criterion for deciding on a meta-CASE tool.

In order to generate/simulate the GUI[5] component of a CASE tool (i.e., the description of the CASE tool's GUI), every meta-CASE has its own GUI meta-model that allows method engineers[6] to specify the interface they want. The current research all share the same approach, that is, method engineers[7] associate, with each concept of a meta-model, a form built from the concept's characteristics. This form is described either with a formal text (MetaView [5]) or via a graphical editor (MetaEdit+ [9, 15], GraphTalk [19]).

In this article, we will review some important but still unexplored challenges. They are dictated by two goals: *a)* meta-CASEs GUIs should be as good as hand-coded CASE tools and *b)* they should be independent of the model of the meta-repository (i.e., the meta-meta-model). That is, whatever the way engineers will model a methodology, it should be possible to define any GUI over its meta-model.

We will present the Grasyla language: a graphical symbolic declarative language to define the graphical representation of a specification (i.e., an instance

[2] OSA, SADT, UML, OML, ERA, NIAM, ...
[3] Object Oriented Analysis, Distributed Systems, Workflow, Client/Server, Real Time, ...
[4] Integrated Development Environment
[5] Graphical User Interface
[6] Method engineers define the models and methodologies.
[7] The engineer who edits and defines the meta-models.

of a meta-model). One will present the motivations and the strengths of this approach with two case studies.

Section 2. will present a résumé of the main concepts of our meta-repository. The next section analyses the challenges of the graphical visualization task in meta-CASEs. Grasyla is presented in section 4. We will describe how Grasyla helps the method engineer and takes up the challenges. After a brief description of the implementation, a case study will be discussed.

2. META-REPOSITORY PRESENTATION

The meta-repository stores information about both types and their instances. Because concepts stored in this meta-repository will describe CASE tools, we will call types and instances respectively *meta-classes* and *classes*. The example below shows the different levels of abstraction: `Customer` is the name of a class of the application domain, and is a component of the specifications built by the analyst; (`''Smith''` is an instance of this class and concerns users only); `Customer` is an instance of meta-class `entity-type` that is a component of standard entity-relationship models.

`''Smith''` $\rightarrow$ `Customer` $\rightarrow$ `entity-type` $\rightarrow$ `meta_class`

The $\rightarrow$ symbol denotes the *instance-of* relationship.

The meta-classes can have attributes (*meta-properties*) and participate in one-to-many relationships (*meta-relations*) with other meta-classes. Meta-classes can inherit characteristics from several meta-classes (*multiple inheritance*). Meta-classes are grouped together to define ontologies (*meta-models*). For instance, these sentences could describe very basic entity-relationship diagrams that are made up of entity-types and attributes:

```
meta-class <entity-type> {
  string:  <name>;                      ‡a meta-property
  owner of <has>;                       ‡ the ∞-role of a meta-relation
  local identifier = { <name> }};       ‡name identifies entity-type
meta-class <attribute-type> {
  string:  <name>, <type>;
  integer:  <min>, <max>;
  member of <has>;                      ‡the 1-role of a meta-relation
  local identifier = { <name>, owner of <has> }};
meta-model <ERA diagram> {              ‡a meta-model
  string:  <name>;
  document:  <author>;                  ‡URL of the author
  list video:  <minutes>;               ‡a multivalued video meta-property
  components = { <entity-type>, <attribute-type> }};
```

Some details have been omitted for simplicity sake. For instance, the meta-repository includes characteristics such as *1)* multivalued meta-properties and multimedia types[8]; *2)* meta-models are themselves meta-classes and share their

[8] `picture`, `audio`, `video`, `document`.

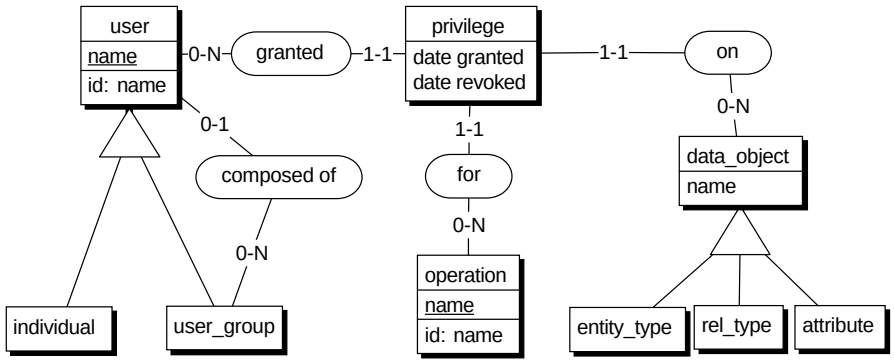

Figure 1.1 **[Database Security Diagram Meta-Model]** This schema models the access rights granted to users for operations on database objects such as *entity-types*, *attributes* and *rel-types*. A *user* is either a *user_group* or an *individual*. A group is itself composed of users.

semantics; *3)* the member roles can be mandatory and *4)* meta-models can share one or more meta-classes in their definitions[9]. The meta-repository and its semantics are described in [4].

The meta-CASE is endowed with a meta-language (Voyager 2) that allows the method engineer to define new functionalities [3, 4]. Voyager 2 is an object-oriented C/Pascal-like language with meta-types, predicative queries, garbage collected lists, GUI functions, dynamic linked libraries, I/O statements, communication with Windows programs and meta-homogeneity[10]. Each meta-class has predefined triggers and predicates that can be overloaded by the method engineer.

The ERA schema depicted in fig. 1.1 describes the meta-model of a "*Database Security Diagram*" (DSD). This example will be used throughout this article.

3. THE CHALLENGES

The graphical representation of specifications is a crucial issue when building CASE tools. As for many programs, the graphical interface will be a major criterion (both subjective and objective) in a CASE tool evaluation and acceptance. Norman *et al* [17] argues that the "*customization without loss of functionality* " and "*customizable user interfaces* " were needed in CASE tools and should be investigated in the 1990's. Moreover, recent studies [8, 12] explain that this gap

[9]The `components` clause in the extract.
[10]The meta-meta-model can be queried in the same way as a meta-model.

is not yet filled "*A well-designed user interface should create a metaphor that bridges the conceptual gap between a computer system and human thinking. Such a metaphor is not the strength of current CASE tools*".

From our experience in the building process of the DB-MAIN tool [6], we observed these criticisms and we have deduced some important rules concerning the graphical user interface, and more specially the graphical representation:

1) One needs both textual and graphical views. Our experience leads us to consider two kinds of specification visualization: graphical diagrams (graph, tree, table, matrix, ...) and textual views (report, code, hypertext, ...). Software engineers often require several views of a same specification depending on the process to perform. Graphical views are preferred for teaching or for validation while textual views[11] are preferred to edit huge specifications. Meta-CASE tools have thus to offer these views.

2) The graphical representation of a concept is contextual. Its representation depends on its use. For instance, in ERA diagrams, attributes can be indented (resp. underlined) depending on their belonging to compound attributes (resp. to identifiers).

3) The shape of a concept is polymorphic. In OO meta-models, concepts are modelled by inheritance hierarchies. Although this graph denotes an atomic concept, its representation will depend on its position in the graph. For instance, if one encounters this case: "`weak-entity-type isa entity-type`" in a meta-model, software engineers will expect to have distinct shapes for them.

4) The graphical representations must evolve. Scientists have certainly not yet discovered the ultimate methodology; new ones appear every year. However, they generally share common ontologies like statecharts, static model (ERA/NIAM/class/...), use cases, ... Semantics change but the structural meta-model does not vary very much. The main changes we observed are about the graphical representation[12]. Moreover, in meta-CASEs, method engineers can specialize and edit the meta-models to adapt them to their company requirements. These modifications must often be reflected in the graphical representation.

5) CASE tools must support multimedia data. CASE tools must support informal processes[8] and must accompany the software engineer in its rigorous steps as well as in "softer" aspects of software development. Multimedia data can enrich specifications with informal information such as interviews, requirement documents, imported diagrams, ... and bridges a gap between CASE tools and non-standard tools[13] in the software engineer environment.

[11] Textual views can have facilities like: sort algorithms, cross-referencing, ...
[12] For instance: the static diagrams in the Booch, OMT, UML and OML methodologies.
[13] Graphical editors, home-made tools, ...

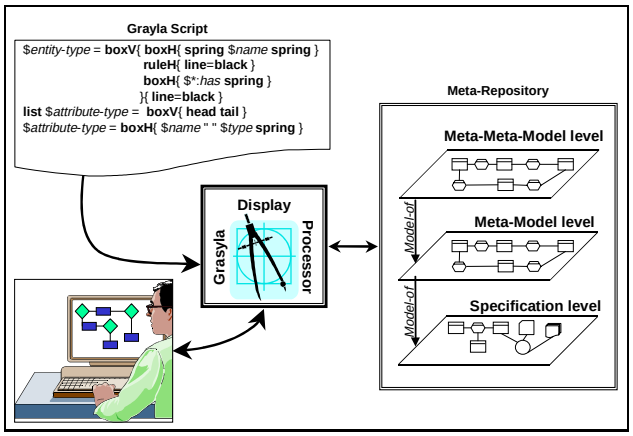

Figure 1.2 This schema shows how the Grasyla Display Processor interacts with the software engineer and the meta-repository. Its behaviour is controlled by the script

6) The graphical requirements should not affect the meta-model definition. Meta-repositories often offer richer abstractions than usual DBMS in order to meet the peculiar needs of software meta-models. However, this quality would be useless if we had to change the meta-model definition each time we modify or add a new visualization.

This list discusses some of the main challenges the engineers should investigate when defining the GUI requirements of a meta-CASE.

4. THE GRASYLA LANGUAGE

Grasyla is a graphical symbolic language that allows the method engineer to define the appearance of a specification through a display script. A Grasyla script defines which concepts the display processor will have to show as well as their graphical representation. Display processors manage all the interactions with the software engineer (contextual menus, click, drag&drop, selection, . . .). The environment of the graphical processor is shown in fig. 1.2.

4.1 THE GRASYLA PRINCIPLES

Each meta-model can have several named Grasyla scripts; each one will correspond to a definite visualization of the specifications. Scripts are made up of three sections (directive, main and connection sections). The ***directive section*** selects the meta-classes to display. The ***main section*** associates with each meta-class an expression that explains how to display its instances. This section is a textual description of a function $\$: \text{MetaClass} \times \{\texttt{single}, \texttt{list}\} \times \text{String} \rightarrow \text{G-Expr}$. When the graphical processor will have to display one or several meta-class instances, it will use the $ function to retrieve the best expression to build the shape of this class. The first argument of $ denotes the type of the class. The second argument will be `single` (resp. `list`) for a single

(resp. several) instance to display. The last argument denotes an identifier that will act as a switch and will be explained later. The section will be composed of constructs that will reflect the different kinds of argument of the $ function. The method engineer will have to edit one of these 4 patterns for each possible entry of the $ function:

1)	`$`*name* `=` *Grasyla expression*
2)	`list $`*name* `=` *Grasyla expression*
3)	*Func*`($`*name*`)` `=` *Grasyla expression*
4)	*Func*`(list $`*name*`)` `=` *Grasyla expression*

Grasyla expressions are symbolic descriptions of shapes built from the "semantics" of a concept/meta-class, i.e., its attributes, its roles, and its supertypes. The different kinds of expressions are

`$`***att*** **:** where *att* is the name of a meta-property; graphical representation of a meta-property value.

`$1:`***rel*** **:** where *rel* is the name of a meta-relation; graphical representation of the meta-relation's owner.

`$*:`***rel*** **:** where *rel* is the name of a meta-relation; graphical representation of the meta-relation's members.

***Func*`(`*exp*`)` :** *Func* is an identifier; graphical representation of the *exp* expression in using the *Func* name.

`boxH{`$\boldsymbol{exp}_1 \ldots \boldsymbol{exp}_n$`}{`***args***`}` **:** arranges n graphical objects (described by the exp_i) horizontally in a box. Similar expressions exist for circles, round rectangles, diamonds, ... `args` is a list of parameters that specify the colour, the line width and other aspects.

`boxV{`$\boldsymbol{exp}_1 \ldots \boldsymbol{exp}_n$`}{`***args***`}` **:** same as both but for vertical order.

`if` ***cond*** `then` $\boldsymbol{exp}_1$ `else` $\boldsymbol{exp}_2$ **:** uses graphical expression exp_1 or exp_2 depending on the value of the *cond* expression.

`head`**,** `tail` **:** used when the argument denotes a list (patterns of lines 2 and 4); denote respectively the first element and the tail of the argument; makes it possible to define recursive graphical expressions.

`spring` **:** introduces in a shape an invisible compressed spiral spring that will stretch out between the neighbouring faces.

These rules give the general principles of the Grasyla expressions syntax. Other functions exist such as handles to hitch arrows, line separators, multimedia data, fonts, aggregate forms, graphical alignment and so on. The Grasyla principle is inspired by the TEX's boxes [10].

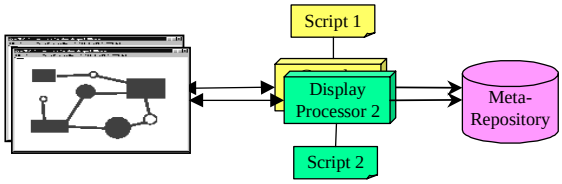

Figure 1.3 They are as many Display Processors as active displays. The events relative to a display are trapped and managed by its DP. DPs access the meta-repository only to read the classes description and to update their positions once the display is closed. Several DPs can process a same specification whatever their scripts. This makes possible several displays with distinct views of a same specification

Finally, the ***connection section*** will associate edges to meta-relations through such expressions as:

$*name* = connect{ *or-param* }{ *line-param* }{ *targ-param* }

The parameters are lists of values that specify respectively the form of the origin handle, the body of the line and the target handle. For instance, the following sentence

```
$has= connect{ outside_arrow{ size=5, fill=black, line=black }}
             { line=black }
             { bullet{ size=4,fill=white, line=black }}
```

will display the instances of the meta-relation `has` with this symbol: ○→ (the circle on the attribute side and the arrow on the entity-type side).

4.2 THE IMPLEMENTATION

The Grasyla architecture is depicted in fig. 1.3. Each “display” (window, printer, clipboard) is controlled by a *Display Processor* (DP). This machine has to display a specification with respect to a Grasyla script and must manage all the possible interactions with the software engineer. Every DP has its own memory, can access the meta-repository and can communicate via a common black-board with other DPs. The software engineer can ask for several displays of a specification whatever the script. He can thus visualize several parts of a huge specification in distinct windows simultaneously.

The meta-repository stores the (x, y) positions of the main objects only. Their shape description is automatically computed by the Grasyla machine. This independence makes it possible to keep the layout of a diagram even if its definition changes.

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 cannot be

modelled directly in Grasyla: Matrices, Tables, Browsers, Sequence Diagrams, Screen Layouts, ... For this reason, the meta-CASE architect[14] 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. Hence, one could think of writing a DP to display graphical matrices, for instance a matrix with the pictures of the software engineer as x-axis and entity-types as y-axis. Cells of such a matrix could be a textual form of the respective rights granted (read/write/...).

4.3 TEXTUAL REPRESENTATION

Grasyla can also be used to define textual views in exactly the same way as discussed in the previous sections. Moreover, these views are active graphical displays where bitmaps, boxes, arrows, ... have disappeared. This kind of textual representation is closer to hypertext than a passive ASCII text. This brings much more benefits. The modification of the text remains synchronous with the meta-repository and the software engineer can navigate through it via hyper-links. Nevertheless, the Voyager 2 language is much more suited to generate ASCII reports or code (RTF/IDL/HTML/...).

4.4 EXAMPLE

This example will illustrate the basic concepts of Grasyla on the "toy" ERA meta-model presented in Section 2. Let us imagine that entity-types are displayed as boxes with two compartments, their name being diplayed centered in the first one and attribute names left-justified in the second one. The compartments are separated by a line. The Grasyla script that defines this graphical layout is as follows:

```
1. $entity-type = boxV{ boxH{ spring $name spring }
2.                      ruleH{ line=black }
3.                      boxH{ $*:has spring }
4.                    }{ line=black }
5. list $attribute-type = boxV{ head tail }
6. $attribute-type = boxH{ $name " " $type spring }
```

CUSTOMER
Name String **ID_code Integer**

– Output Sample –

When a DP will have to display an entity-type, it will use the first rule (lines 1–4), substitutes all the variables (`$name` and `$*:has`) by their respective values and proceeds with other rules depending on the type of the variables. The `has` meta-relation will be replaced by a list of attributes, and another rule (line 5)

[14] The engineer who define and implement the meta-CASE program.

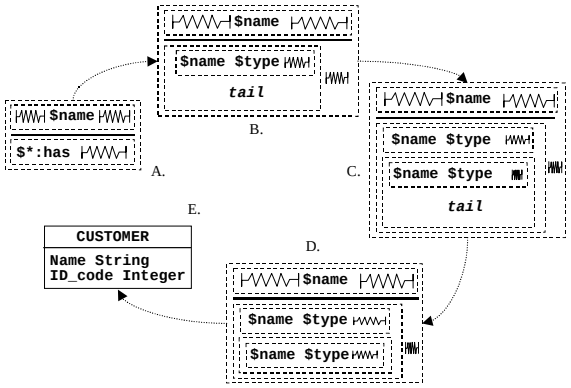

Figure 1.4 The A box denotes the Grasyla expression defined in lines 1–4. The $A \rightarrow B$ transition replaces the `$*:has` variable through the rule of line 5. The head has been substituted through the rule at line 6. The tail becomes empty at step C and variables are replaced with their respective values in transition $D \rightarrow E$

will be used, and so on. The picture in fig. 1.4 shows the successive steps the DP will follow to display an entity-type with two attributes.

5. CASE STUDY: A SECURITY META-MODEL

Let us examine the DSD meta-model (see fig. 1.1). 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 1.5 illustrates the graphical representation of a DSD specification excerpt. Items were positioned by the software engineer. 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* the *address* attribute and *d)* members of group *staff* are allowed to *read* entity type *customer* and to *delete* entity type *order*.

The Grasyla script that tells the display processor how to display each concept (*users*, *user_groups*, *data_objects*, *privileges*, ...) is given in fig. 1.6.

6. CONCLUSION

This article has presented the main challenges that should be taken into consideration when defining a meta-model of a CASE tool graphical user interface. The Grasyla graphical language was defined to meet these requirements in order to reconciliate the software engineers with CASE tools that are sometimes too static and too formal. We showed on several examples that the language was as simple as possible. The graphical language is powerful enough to bootstrap two editors of the meta-case to edit the meta-models definitions and to define the Grasyla scripts, i.e., to propose a graphical counterpart of its textual representation. Figure 1.7 illustrates a screen-shot of the meta-CASE with several advanced views.

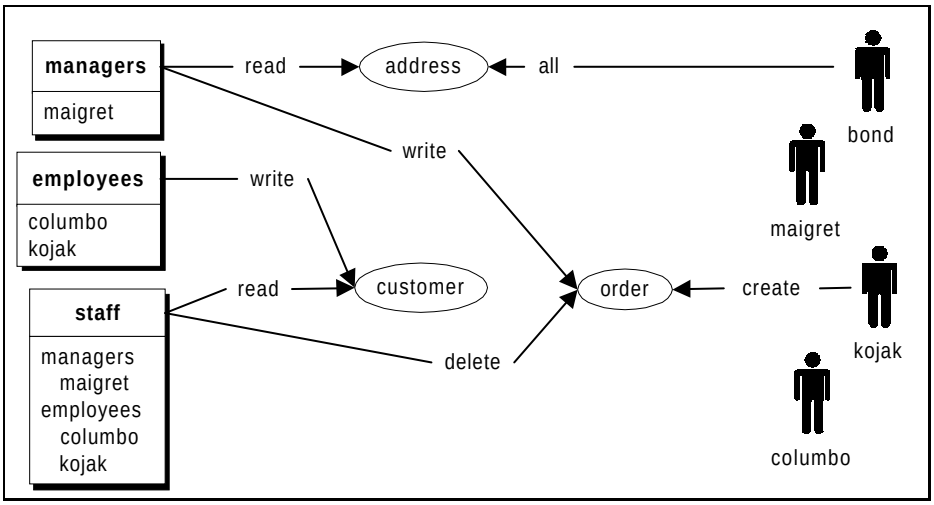

Figure 1.5 Graphical view of a DSD specification

```
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                   }{frame=shadow colour=black}
06 $individual= boxV{boxH{handle spring bitmap("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 colour=black}
13 $operation= boxH{handle $name handle}
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{$name spring}
18                              boxH{H(10pt) in_group($compound_of) spring}
19                             }
20 $granted = connect{ }{line=black width=2pt}{ }
21 $on = connect{outside_arrow{size=5}}{line=black width=2pt}{ }
```

Figure 1.6 **[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 to display instances of the corresponding meta-classes in terms of geometrical forms (box, oval, line, . . .), letters and their characteristics (properties and roles). Statements at lines 20 and 21 define the arcs denoting relations between classes. The use of the *in_group* name (line 4) allows the method engineer to use non standard Grasyla expressions for the *user* (and its subtypes) classes when they figure in *user_group* boxes

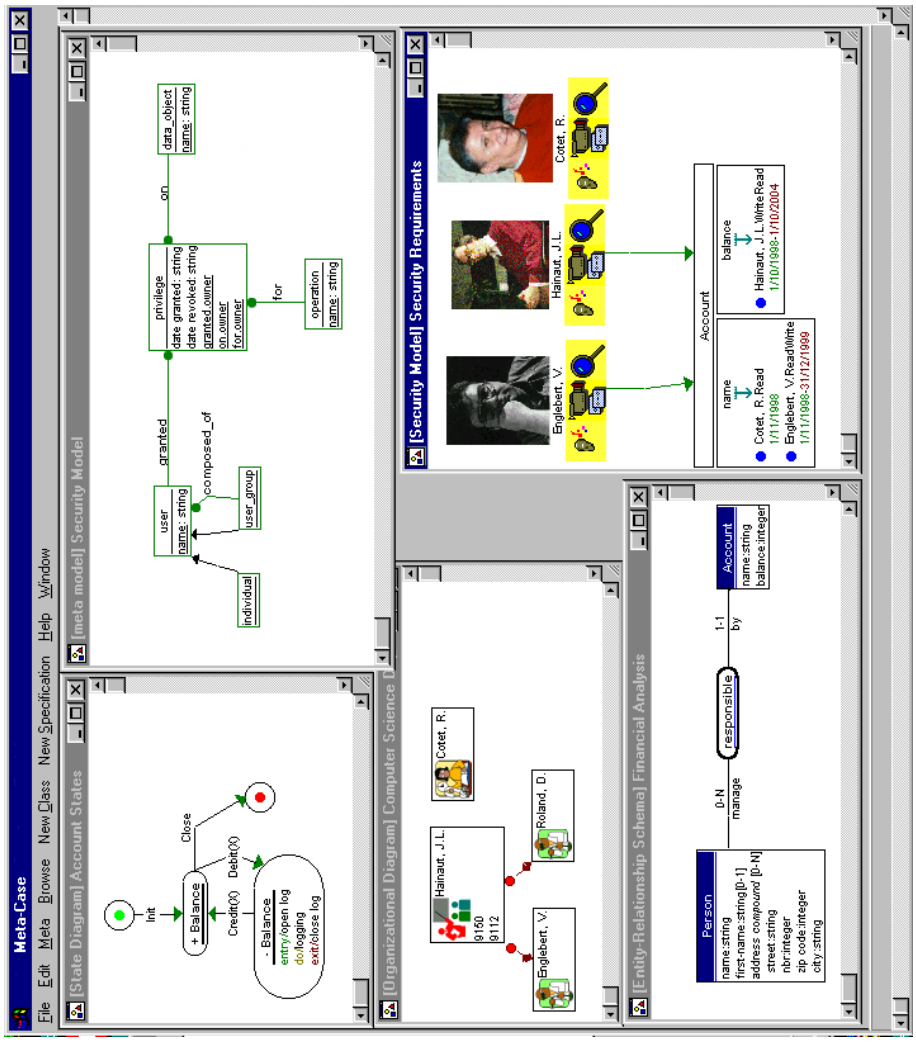

Figure 1.7 **[Screen Shot]** Four specifications and one meta-model definition are visualized in this screen shot. Windows denote respectively *1)* a state diagram *2)* an organisation diagram *3)* an entity-relationship schema *4)* a visualization of a DSD specification wrt. a richer Grasyla script 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 as a table, 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/magnifying glass) are active and a double click on them shows a multimedia document

References

[1] 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, pages 203–216, Barcelona, Catalonia, Spain, June 1997.

[2] H. Elshazly and V. Gover. A study on the evaluation of CASE technology. *Journal of Information Technology Management*, 4(1), 1993.

[3] Vincent Englebert. *Voyager 2. Version 3 Release 0*. University of Namur - DB-MAIN, FUNDP, Rue grandgagnage 21. 5000 Namur. Belgium, 1998.

[4] Vincent Englebert and Jean-Luc Hainaut. DB-MAIN: A next generation meta-CASE. *Information Systems*, 24(2):99–112, 1999. Special Issue on MetaCASEs.

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

[6] 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.

[7] 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.

[8] Stan Jarzabek and Riri Huang. The case for user-centered CASE tools. *Communications of the ACM*, 41(8):93–99, August 1998.

[9] 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* 8^{th} *International Conference CAiSE'96 on Advanced Information Systems Engineering*, volume 1080 of *LNCS*, pages 1–21, Heraklion, Crete, Greece, May 1996. Springer-Verlag.

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

[11] Diane Lending and Norman L. Chevarny. The use of CASE tools. In *SIGCPR'98. Proceedings of the 1998 conference on Computer personnel research*, pages 49–58, 1998.

[12] U. Leonhardt, J. Kramer, B. Nuseibeh, and A. Finkelstein. Decentralised Process Enactment in a Multi-Perspective Development Environment. In *Proceedings of the 17th International Conference on Software Engineering*, pages 255–264, April 1995.

[13] Fred Long and Ed Morris. An overview of PCTE: A basis for a Portable Common Tool Environment. Technical Report CMU/SEI-93-TR-1, Soft-

ware Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, March 1993.

[14] M.P. Martin. The case against CASE. *Journal of Systems management*, 46:54–57, Jan/Feb 1995.

[15] MetaCase Consulting, Ylistönmäentie 31. FIN-40500 Jyväskylä. Finland. *MetaEdit+ 3 Method Workbench User's Guide. Version 3.0*, 1999.

[16] C.R. Necco, N.W. Tsai, and K.W. Holgeson. Current usage of CASE software. *Journal of Systems Management*, pages 6–11, May 1989.

[17] Ronald J. Norman, Wayne Stevens, Elliot J. Chikofsky, John Jenkins, Burt L. Rubenstein, and Gene Forte. CASE at the start of the 1990's. In *ICSE '91. Proceedings of the* 13^{th} *international conference on Software engineering*, pages 128–139, Burlington, MA, USA, 1991. International Workshop on CASE.

[18] L. Beth Protsko, Paul G. Sorenson, Tremblay J. Paul, and Douglas A. Schaefer. Towards the automatic generation of software diagrams. *IEEE Transactions on Software Engineering*, 17(1):10–21, January 1991.

[19] Rank Xerox France, Direction Informatique Avancée et Génie Logiciel. 7 rue Touzet Gaillard. 93586 Saint-Ouen CEDEX. *GraphTalk Environnement objets de développement et d'utilisation d'atelier de génie logiciel*, 1991.

[20] Matti Rossi, Mats Gustafsson, Kari Smolander, Lars-Ake Johansson, and Kalle Lyytinen. Metamodeling editor as a front end tool for a CASE. volume 593 of *LNCS*, Manchester, May 1992. 4^{th} International Conference CAISE'92, Springer-Verlag.

[21] J.A. Senn and J.L. Wynekoop. The other side of CASE implementation: Best practices for success. *Information Systems Management*, 12:7–14, 1995.

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

[23] M. Sumner. Making the transition to computer-assisted software engineering. In A.L. Lederer, editor, *1992 ACM SIGCPR Conference*, pages 81–92, New York, 1992. ACM Press.

[24] Unisys. *Universal Repository. Technical Overview. Release 1.2*. Unisys Corp., August 1996.