Using Prolog to implement a CASE shell for Object-Oriented Development [*]

Esteban Zimányi
University of Brussels,
INFODOC,50 Av. F.D. Roosevelt,
CP 175-2, 1050 Brussels,Belgium,
e-mail: ezimanyi@ulb.ac.be.

Manuel Kolp
University of Louvain,
IAG-QANT, 1 Place des Doyens,
1348 Louvain-La-Neuve, Belgium,
e-mail: kolp@qant.ucl.ac.be.

Abstract

This paper shows how we use Prolog as the only programming language for constructing a flexible CASE shell environment supporting the tasks of object oriented development. We explain the important role played by Prolog in the entire design and the architecture of such a CASE shell, which has a sophisticated Graphic User Interface. The advantages and facilities of using Prolog as developing environment are illustrated with respect to our system called Eroos Case.

Keywords: Prolog, object-oriented development, CASE.

1 Introduction

Since the recognition of the software crisis in the mid-70's, a huge amount of work has been done to devise methods that help analysts to develop systems of ever-increasing complexity. Most of the "structured" methods developed in the 70's used a functional decomposition paradigm by which the system functionality is specified using a "black-box" approach. While this technique is appropriate for *transformational* systems, which are mainly driven by data transformations, it is not appropriate to cope with more sophisticated hardware and software technologies like networks, distributed computing, real-time systems, and sophisticated man-machine interfaces.

The object-oriented paradigm was developed to alleviate some of the above problems. Although it was initially used in the context of programming languages, it became rapidly clear that the initial phases of the software development lifecycle would also benefit from this approach. Thus, different methods for object-oriented development appeared (e.g., [9, 4, 1, 2])

Object-oriented methods represent a system with several complementary views or models, each one capturing a particular type of information (or viewpoint) about the system. For example, Object models describe the structure of objects in a system. Dynamic models use an event- and state-driven point of view to describe those aspects of a system concerned with time and control, i.e. the sequencing of operations. Functional models describe those aspects of a system concerned with transformations. Use-Case models describe the interaction of the system with its environment.

Due to the complexity of software development, a significant research activity was the development of Computer-Aided Software Engineering (CASE) tools (e.g. [7]) supporting the multiple tasks of the software development lifecycle. These CASE tools aim to increase both

[*]This work is part of the EROOS (Evaluation and Research on Object-Oriented Strategies) project, principally based at the University of Louvain and the University of Brussels.

the productivity and the quality of the software by taking care of tedious tasks like consistency checking, diagram storage, project management, etc., as well as allowing code generation.

However, new objectives for OO development and its methods continually arise. In particular, technological aspects such as client/server architectures, distributed computing, raise new requirements for altering the methodological basis of software development This leads to the area of Computer-Aided Method Engineering (CAME), i.e. the creation and adaptation of design methods to fit a given development situation. The area of method engineering covers the specification of the syntax and semantics of modeling notations, specification tasks and their dependencies, and modeling the organization of tasks among the various actors.

In this context, we developed in Prolog a flexible CASE shell supporting both the task of method engineering as well as the task of OO development. Several well-known OO development methods have been already implemented in the tool.

Our tool is characterized by a high degree of flexibility which permits the user to configure the methods supported and the ways these methods are supported. Thus, it supports the approach of incremental method engineering, i.e. methods can be gradually expanded, refined and formalized based on users' experiences and learning, and the changing demands of the development situation. An important feature to achieve flexibility is the representational independence, i.e. the separation of the conceptual and representational constructs and their flexible combination in the modeling environment.

The use of Prolog for completely building our CASE environment is essential to our approach. First, as it is well-knwon, the declarative nature of Prolog gives it several advantages (clarity, modularity, conciness and legibility) over conventional programming languages and make it more suitable for CASE prototype design. Second, and more important, we use Prolog as the only language for the development of both the GUI and the core of the system. We concretely show that, contrary to what is currently believed, using Prolog for the development of the entire application is better and more consistent than the alternative approach of implementing the GUI using an imperative language and implementing the Inference Engine in Prolog.

The rest of the paper is structured as follows. Section 2 presents the general architecture of our CASE environment. The subsequent sections explain and illustrate the main components of our system. So, Sections 3 and 4 deals, respectively, with Model and Method Management. Section 5 presents the Knowledge Base (KB) Management, Section 6 is devoted to Graphics Management and Section 7 to GUI management. Finally, Section 8 briefly describe examples of implemented modules while Section 9 gives conclusions and points to further work.

2 System Architecture

Figure 1 shows the general layout of the Eroos Case. It has a sophisticated GUI (Graphic User Interface) including Graphical Schema Editors, Dialog Boxes, MDI (Multiple Document Interface) windows, Button Palettes and so on. As shown in Figure 2, our shell can be seen as a compound system consisting of three integrated layers.

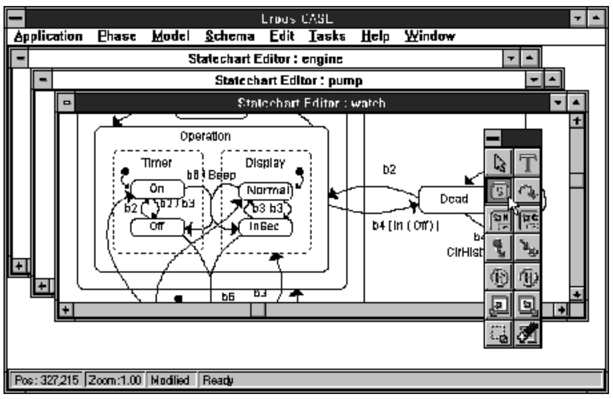

Eroos CASE
Application Phase Model Schema Edit Tasks Help Window
Statechart Editor : engine
Statechart Editor : pump
Statechart Editor : watch
Operation
Timer
Display
Beep
On
Off
Normal
InSec
Dead
b2
b4 [In (Off)]
ClrHist
b6
Pos: 327,215 Zoom:1.00 Modified Ready

Figure 1: General layout of the Eroos Case.

The first layer is the *Eroos Engine*, which is the core of the system. It implements abstract primitives common to all models and methods. It is composed of 6 subsystems explained next.

The *Abstract Model* defines primitives used for defining concrete models, such as Entity-Relationship or Statechart models. Similarly, the *Abstract Method* defines primitives used for defining concrete methods, such as OMT or Fusion. The *Abstract Graphics* provide primitives used to define the graphical representation of models.

The *Abstract GUI* provide basic primitives for the manipulation of GUI components such dialogue boxes, list boxes and so on. The *Knowledge Base Manager* is in charge of the repository. Finally, the *Application Manager* takes in charge the management of projects, schemas, reports, and so on.

The second layer (*Models Layer*) implements specific concrete models, such as ER+, statecharts, DFD, Object Interaction Graphs, and so on. Each concrete model defines the conceptual language used to describe a particular aspect of the system. For example our ER+ model defines abstractions such as generalization, aggregated relationship, subset relationship, and derived relationship, in addition to the basic ER constructs.

The third layer (*Methods Layer*) implements concrete methods such as Fusion, OMT, Booch, Jacobson, or even Relational DB Design. Each of these methods uses models defined in the Models Layer. For instance, OMT integrates the ER+, Statecharts and DFD models respectively as static, dynamic and functional model.

As shown by the dashed lines in Figure 2, each concrete method or model is defined by providing the concrete components corresponding to the abstract components. Examples of a concrete model and a concrete method are given respectively in Sections 3 and 4.

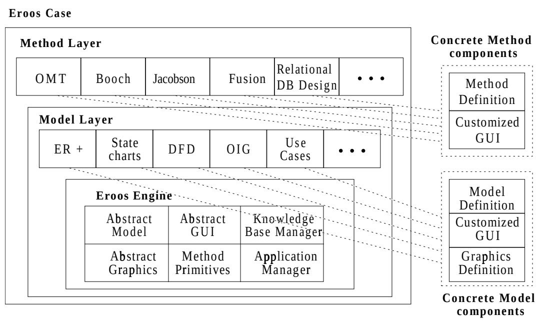


Figure 2: System architecture of the Eroos Case.

The main components of the Eroos Case are explained next. We refer to [11] for a more detailed description of our system.

3 Models Management

To each concrete model is associated a set of concepts, each defined by a predicate. For example, an Entity-Relationship model defines concepts such as

```
concept(entity(‘Entity Name‘),‘ER+‘).
concept(relationship(‘Relationship Name‘),‘ER+‘).
concept(participates(‘Relationship Name‘,
    ‘Participant Name‘,‘Minimal Cardinality‘,
    ‘Maximal Cardinality‘,‘Role‘),‘ER+‘).
```

Different predicates can be used to define attributes of predicates. For example, such predicates can be as follows:

```
domain(participates,‘Min Cardinality‘,integer).
dflt_value(participates,‘Minimal Cardinality‘,0).
dflt_value(participates,‘Maximal Cardinality‘,n).
allow_nulls(participates,‘Role‘).
```

To each concept is associated a set of integrity constraints. The integrity checking of models is realized at three different levels: (1) constraints associated to the arguments of a predicate (`ic_attrib`); (2) constraints associated to the whole predicate (`ic_concept`); and (3) constraints associated to the consistency of a model (`ic_model`). As will be shown in Section 4, an additional fourth level defines constraints associated to the consistency between different models, i.e. a method (`ic_method`). We give next examples of constraints associated to each of these three levels.

```
ic_attrib(entity,‘Entity Name‘,Name) :-
    check_neg_refer_int(‘Entity Name‘,Name,
    relationship(Name)).
ic_concept(participates
   (RN,PN,MinCard,MaxCard,R1)) :-
   MinCard > MaxCard,error_msg([‘The Maximal
   Cardinality must be greater or equal than ‘,
   ‘the Minimal Cardinality.‘]),!.
consist_rule(relationship(RName)) :-
    findall(Part,participates(Rel,Part,MinCard,
    MaxCard,Role),L),length(L,N), N<2,
    error_msg([‘Relationship ‘,Rel,‘ has less
    than 2 participating objects.‘]).
```

The first constraint is attached to an attribute: It checks that the name of an entity can not be the same of the name of a relationship.

The second constraint is attached to a concept (predicate `participates`): it states that the maximal cardinality must be greater than or equal to the minimal cardinality.

The third constraint is for model consistency: it verifies that each relationship has at least two participating objects.

Notice that when a constraint is violated, appropriate messages are displayed in dialogue boxes using predicate error_msg.

In addition, each model must define a set of cascade rules that must be enforced when the Knowledge Base is modified. These rules are defined by add_rule, modify_rule, and delete_rule. Consider the following two rules

```
modify_rule(entity(OldName),entity(NewName)) :-
 modify_cascade(attribute(OldName,Attr,MinC,MaxC),
 attribute(NewName,Attr,MinC,MaxC)).
delete_rule(entity(Name)) :-
 delete_cascade(attribute(Name,Attr,MinC,MaxC)).
```

The first rule is fired when an entity name is changed: it replaces in cascade the new entity name in all corresponding attribute predicates. The second rule is fired when an entity name is deleted: it deletes in cascade all corresponding attribute predicates.

The ER+ model explained above is an example of a concrete model defined using the primitives of the Abstract Model component of the Eroos Engine. As already said, such primitives are applicable to different models. Examples of such primitives are check_refer_int and check_neg_refer_int (already used above) for checking or avoiding referential integrity between two predicates. Other examples of primitive are domain, default_value, and allow_nulls, which were used above for defining attributes associated to concepts.

Also, several ic_attrib, ic_concept and cascade rules are attached to the Abstract Model Component. For example, one of them tests that the same fact is not entered twice in the repository.

4 Method Management

A concrete method associates to each phase of the development lifecycle a set of models. For example, the Fusion method associates to the analysis phase an object model, a system object model, and an operation model. Further, the concrete methods also determine the fourth level of integrity checking , i.e. the constraints checking the consistency between models. For example, the next constraint

```
ic_method :-  statechart(Statech),
  \+(attached(Statech,Class)),
  error_msg([‘Statechart ‘,Statech,
  ‘ is not attached to a class in
  the object model.‘]).
```

checks that every statechart defined in the dynamic model is attached to a class defined in the object model.

Each concrete method customize its set of models according to the method (and users') specificities. For instance, the Fusion method do not use derived relationships (defined in our ER+ model) while the Relational Database Design method integrates them as an essential mechanism to detect relational redundancies. Therefore, each concrete method must determine the concepts which are supported by its models.

Also, each method must determine which are the operations that can be performed in each phase and model. For example, for the Relational Database Design method, one operation allows to automatically generate from an ER+ conceptual schema a relational database schema which is in Inclusion Normal Form. In the same way, in Rumbaugh's OMT method, another operation allows to generate from a statechart specification the C++ code that realizes the corresponding functionality. These operations are briefly described in Section 8.

Finally, the concrete methods define how to realize a report containing all knowledge of the repository for a particular phase (e.g. analysis) and model (e.g. object model). Such reports constitute the textual representation of the graphical models.

The Abstract Method component defines primitives that are general to all methods. Examples of such primitives are load_method, generate_report, and consistency_check for firing the consistency check between models. Primitives choose_phase and choose_model are used to change the current phase and model.

5 Knowledge Base Management

This component is in charge of the management of the Knowledge Base. As a matter of fact, there are two different Knowledge Bases. The *Concept Repository* contains facts such as entity(employee) and relationship(works_on). The *Graphics Repository* contains the graphical representation of the repository concepts: it contains, e.g. the following predicate

```
grafix(er_window_1,[brsh(255,255,255,0),
  pen(0,0,0,1),rect(37,29,107,69),font(gredit),
  trns(1),fore(0,0,0),text(40,29,'employee')]
```

keeping the list of graphical commands that draw a rectangle and the text `employee` on the graphical window `er_window_1`. This is the graphical representation of `entity(employee)`.

Thus, it is necessary to have a predicate `gfx_conc` that makes the correspondance between each concept and its graphical representation. Such predicate is used, e.g., when the user clicks the right button on an object, to determine the corresponding repository concept that is to be modified.

To modify the contents of the Concept Repository, predicates `add_element`, `modify_element`, and `delete_element` are used. However, when the Concept Repository is modified, it must be ensured that it remains consistent. As explained above, four levels of integrity checking are defined. The Knowledge Base Manager component contains primitives for firing such integrity checking with predicates such as `check_ic_attrib`, `check_ic_concept`, `check_ic_model`, and `check_ic_method`. It also contains the primitives for firing the cascade rules with predicates `fire_add_rules`, `fire_modify_rules`, and `fire_delete_rules`.

Every modification of the Concept Repository (such as creating a new entity, connecting an entity with a relationship) implies a graphical primitive to be executed on the corresponding window. However, some graphical operations do not imply a modification of the repository (e.g. moving a relationship in a schema).

Predicate `assert_grafix` and predicate `retract_grafix` are used to modify the contents of the Graphics Repository. As a side-effect, these predicates draw and delete the graphical objects from the windows.

6 Graphics Management

Any graphics application needs to maintain a logical representation of the image in any window, so that it can be repainted when necessary. Operations such as moving, scrolling, and resizing of windows can cause portions of graphics regions to be uncovered and thus, the the newly exposed areas must be redrawn. This requires that the applications store a copy of every graphics operation so that it can be repeated on demand. Such information is kept in the Graphics Repository as explained in the previous section.

It is in this context that the use Prolog offers important advantages over conventional programming languages. First, the use of Prolog's Knowledge Base coupled with the unification and backtracking mechanisms allows to realize graphical manipulation at a more conceptual level. For realizing equivalent manipulations in other imperative languages it is necessary to use dynamic memory allocation as well as lists traversals and pointers manipulation.

The Abstract Graphics Component define primitives for manipulating graphics objects independently of their shape. Concrete Graphics Components are defined for each model defining how objects are draught, moved, selected, and so on.

Some of the abstract graphics primitives are explained next. Predicate `extent` returns the physical extent of a graphics sequence; predicate `shift_args` shifts the graphics elements by a specified offsets. There are also predicates for manipulating the "select" corners such as `draw_corners`, `delete_corners`. A predicate `test_objects` determines the set of graphics object enclosed by a rectangle in the screen. Other predicates show and store status information, include the position of the cursor when it is moved in a graphics window, scroll the image, move (dragg) or resize a set of selected objects, magnify and reduce the scale (zoom-in and zoom-out) of a window and so on.

Another primitive is used to select the objects enclosed in an area that is drawn in the screen; these selected objects can then be moved, deleted, and so on. Figure 3 illustrates how the Abstract and the Concrete Graphics components work together. In Figure 3 (a), the user selects an area in the screen using the mouse cursor. Then all objects that are enclosed in the selected area are selected. For this, the primitive for selecting an area (defined in the abstract component) must determine the graphical objects to be selected. This is defined in the concrete components. For example, an entity or relationship is selected if its four corners lie inside the select rectangle. On the contrary, only the participations (i.e. the lines) connecting selected entities and relationships must be selected, the other lines are redraught when the dragging operation is completed.

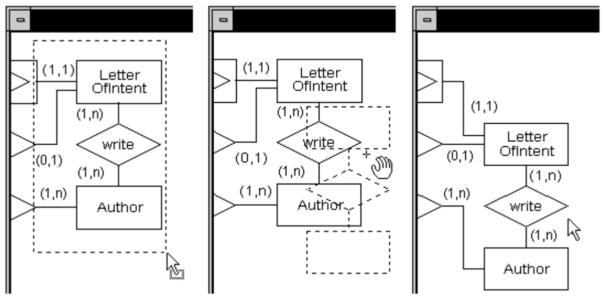


Figure 3: Steps of a dragging operation.

Then, the dragging operation can be applied as shown in Figure 3 (b). During the dragging operation, at each mouse move the outlines of the objects to be moved must be draught. The dragging operation (defined in the abstract component) must determine how the objects are outlined. This is defined in the concrete components. For example, the name of the entities and relationships, as well as the cardinalities of the participations are omitted in the outlines.

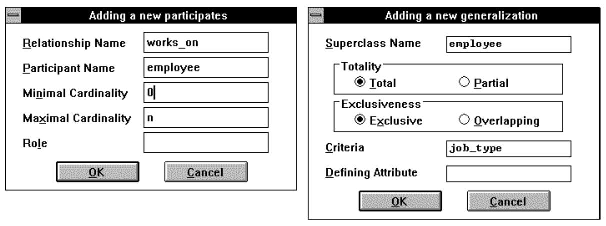
Finally, when the dragging operation is completed the objects must be shifted as shown in Figure 3 (c). Here also the abstract shifting primitive uses the definition of how the different objects are shifted, as defined in the concrete components

7 GUI Management

The Abstract GUI component comprise the primitives for manipulating GUI elements, such as menus, dialogue boxes, palettes and so on. This includes, e.g., adding, removing, selecting and retrieving elements from listboxes, editing and retrieving edit controls, selecting radio buttons, creating and manipulating palettes, and so on.

In addition, the abstract GUI component also defines generic dialogue boxes for n-ary predicates. For example, the dialogue box associated to predicate `participates` is given in Figure 4(a).

These generic dialogue boxes provide basic functionalities for modifying the Knowledge Base. This is extremely useful in the first stages of the method definition. The method designer can define customized dialogue boxes for specific concepts, thus enhancing the basic functionality. For example, the ER+ model defines customized dialogue boxes for generalization, virtual relationships, and so on, one of which is given in Figure 4 (b).



Relationship Name	works_on
Participant Name	employee
Minimal Cardinality	0
Maximal Cardinality	n
Role	

Superclass Name	employee
Totality	[✓ Total] Partial
Exclusiveness	[✓ Exclusive] Overlapping
Criteria	job_type
Defining Attribute	

Figure 4: Two dialogue boxes for the ER+ editor.

A Concrete GUI component is associated to each model. For example the statechart palette given in Figure 1 is defined as follows:

```
palette(sc_palette,7,2,30,30).
palette_button(sc_palette,1,1,arrow_button,
   two_state,select_object).
palette_button(sc_palette,1,2,text_button,
   two_state,add_text).
[...]
```

The first line defines the `sc_palette` as having 7×2 buttons, each of 30×30 pixels. Then each of the buttons is defined with predicate `palette_button`. It defines the bitmap to paint over the button, whether the button is one- or two-state, and the predicate to execute when the button has been pressed.

As can be shown, all the actions and graphical manipulations that must be executed when a mouse button is clicked over a palette button are defined in the abstract GUI component.

8 Examples of implemented modules

This section briefly presents examples of two models (ER+ and Statecharts models) and a method (Relational Database Design) implemented in *Eroos Case*. We refer to [5, 10] for a complete description of these implementations.

Our implementation of the ER+ model [5] takes into consideration, further the basic ER concepts (entity, relationship, attribute, identifier and cardinality), some important abstraction mechanisms as subset relationship, generalisation, aggregated relationship and derived relationship. In this way, it subsumes any object model of the well-known development method.

The ER+ schemas drawn on the screen using the GUI are validated using the integrity constraints defining the syntax and semantics of the ER+ abstractions. Prolog predicates encoding the graphical schemas are automatically generated and can then be processed by the chosen method, e.g. for translating a conceptual model comprising several ER+ schemas into a normalized relational database, as described in the Relational Database Design method below.

Statecharts [3] are an extension of finite state machines and diagrams, with formal syntax and semantic used for describing system dynamics. Although statecharts were originally proposed for desribing hardware devices, the dynamic models of OO methods, such as OMT and Booch, use the statecharts formalism for modeling the temporal behaviour of a system.

Our implementation of the Statecharts model [10] takes into account most of the Statecharts concepts: events, states, super- and substates, transitions (simple, merging, concurrent, and splitting), conditions, actions, hystory enforcement and so on. The module has the following functionalities. The dynamic behaviour of a system is specified as a collection of interacting statecharts, each one attached to one class. As with the ER+ model, these statecharts can be drawn on the screen via the GUI and are validated using integrity constraints. In addition, our module allows to simulate the dynamic behaviour of the application to test whether it behaves as expected. Finally, the system generates the C++ code implementing the overall behaviour of the application.

We explain now the Relational Database Design (RDBD) method [5]. Nowadays, Relational Database Design typically goes through, first, conceptual (ER) schema design and second, on translation into a relational schema. One of the originalities of our RDBD method consists in generating relational database schemas normalized in Inclusion Normal Form (IN-NF) [6]. Unlike classical normal forms that only characterize individual relation schemas, this lesser known relational design criteria remove redundancies in the relational database schema considered as a whole.

In practice, the RDBD method works in *Eroos Case* as follows. First, an application is specified by a set of ER+ schemas using and customizing the ER+ module briefly exposed above.

Then, the relational translation process takes as input the Prolog predicates encoding the ER+ specifications and generates a relational database in IN-NF. The method uses an integrated three steps algorithm: (1) ER+-to-relational mapping, (2) individual relations normalization in third normal form, and (3) database normalization in inclusion normal form.

9 Conclusion

Object-Oriented methods were developed as a way to cope with the inadequacy of structured methods for developing software. However, due to the ever-increasing complexity of software development, researchers have also focused on providing CASE and CAME environments helping designers to build an application in a high flexible way.

In this context, our goal was to show how Prolog can be used for completely constructing a CASE shell supporting object oriented development and including a sophisticated GUI. We explained how the declarative nature of Prolog programming turned out to be extremely powerful in our implementation.

First, we showed that the CASE was conceived with a high modular architecture. Different abstraction mechanisms and models can be customized and combined by the user respectively into a model and a method. Our system then provides an incremental and expansible software engineering environment to describe an application throughout the entire development lifecycle. In this context, an important feature of our CASE is the representational independance between the conceptual constructs and their (graphical) representation.

Second, we showed that the use of Prolog as the only development language for building an entire GUI is viable. In our CASE declarative implementation approach, the use of only Prolog for developing the core of the system and the GUI appears more coherent than using Prolog for the inference engine and a conventional language for the GUI.

Several issues need to be further addressed. Our CASE is currently developed in a PC platform using a Windows 3.1 LPA Prolog compiler. We are planning to port the system first on Windows 95 and later on Sun-Unix workstations. Our Prolog code, especially the code concerning the Eroos Engine primitives, is in fact as far as possible independent of the hardware/software configuration. In this way, only the OS function calls should be modified.

Our CASE provides a powerful testing environment to develop and implement new modeling abstractions and conceptual languages describing an application throughout the development lifecycle. For instance, we already formalized a mechanism for the description of object models, called *materialization* [8]. We are also formally studying the properties of the aggregation abstraction (part relationship). We plan to implement these primitives in order to combine them with the abstraction mechanisms already implemented in the ER+ model. Another work concerns the use of aggregation as a dynamic abstraction, and its interaction with the statecharts formalism.

References

[1] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, second edition, 1994.

[2] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice Hall, 1994.

[3] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[4] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering : a Use Case Driven Approach.* Addison-Wesley, 1992.

[5] M. Kolp and E. Zimányi. Relational database design using an ER approach and Prolog. To appear in *Proc. of the 6th Conf. on Information Systems and Management of Data, CISMOD'95*, 1995.

[6] T. Ling and C. Goh. Logical database design with inclusion dependencies. In *Proc. of the 8th IEEE Int. Conf. on Data Engineering, Tempe, Arizona*, Feb. 1992.

[7] K. Lyytinen and V.-P. Tahvanainen, editors. *Next Generation CASE Tools.* IOS Press, 1992.

[8] A. Pirotte, E. Zimányi, D. Massart, and T. Yakusheva. Materialization: a powerful and ubiquitous abstraction pattern. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. of the 20th Int. Conf. on Very Large Databases*, pages 630–641, 1994. ACM Press.

[9] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[10] E. Zimányi. Statecharts and object-oriented development: a CASE perspective. In *Proc. of the 3rd Int. Conf. on Practical Application of Prolog*, pages 697–718, Apr. 1995.

[11] E. Zimányi and M. Kolp. A Prolog-based architecture for building an Object-Oriented CASE shell. Technical Report RR 95-03, INFODOC, Université Libre de Bruxelles, Belgium, Sept. 1995. Forthcomming.