A Prolog-based Architecture for an Object-Oriented CASE Shell *

Esteban Zimányi † Manuel Kolp ‡

Abstract

The goal of this paper is to show how Prolog can be used as the sole programming language for the development of a highly modular and customizable CASE shell. This CASE shell, called *Eroos Case*, supports the process of object-oriented method engineering and object-oriented development and includes a sophisticated Graphical User Interface. We explain the essential role played by Prolog in developing such a CASE shell, illustrating the advantages and facilities of using Prolog with respect to other imperative programming languages.

Keywords: Prolog, object-oriented development, CASE.

1 Introduction

Since the early days of software engineering, analysts and developers have realized the importance of a systematic approach to the software development process. The first system development methods used a "structured" approach. While these methods were suitable for describing systems driven by data transformations, they were inadequate for coping with more sophisticated technologies like networks, distributed computing, real-time systems or sophisticated man-machine interfaces. A key problem is that structured methods are only of limited use in describing the run-time behavior of hardware and software applications.

Object-oriented (OO) methods (e.g., [21, 20, 4, 5, 22, 8, 17, 1, 6]) aim at remedying the situation. This led to a new way of thinking about problems with models organised around real-world concepts. The fundamental construct is the object, which combines both data structure and behavior in a single entity. Object-oriented methods are useful for understanding problems, communicating with application experts, modeling entreprises, preparing documentation and designing programs and information systems.

*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.

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

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

OO methods generally agree on the decomposition of the development life cycle into several steps: requirements specification, analysis, system design, object design, and implementation. *Requirements specification* consists of eliciting requirements from the users. *Analysis* starts from these requirements and builds a model of the real-world situation showing its important properties. *System design* makes high-level decisions about the overall architecture of the system and its components. *Object design* builds a design model based on the analysis model but containing implementation details. *Implementation* translates the object classes and relationships developed during object design into a particular programming language, database or hardware implementation.

OO methods use several kinds of models to describe a system. Each model captures a particular type of information or viewpoint about the system at a particular step of the development life cycle. These models are orthogonal parts of the description of a complete system and are cross-linked. A few examples of models proposed by the different methods are as follows.

An *object model* describes the static structure of the objects in a system; usually such a model is a variant of the classical entity-relationship (ER) model. A *dynamic model* describes the aspects of a system that change over time and implements the control aspects of a system; such a model is usually based on state machines, statecharts, or Petri nets. A *functional model* describes the data value transformations within a system; usually such a model is a variant of the data flow diagrams. A *requirements model*, usually based on use cases, specifies the system operations, i.e., the set of transactions between the system and the agents in its environment. *Object interaction graphs* define an implementation of system operations stating the objects involved and how they communicate to realize the specification.

Each OO method has its own characteristics both with respect to the kinds of models they use to represent a system and to the steps prescribed to construct such models (the process). Some models appear in several methods, like the object model which is common to all methods. However the variations of such models in each method may be significant. For example, there is still a debate on whether an object model should contain relationships of the kind found in ER models.

Significant research in Computer-Aided Software Engineering (CASE) tools has been spurred on by rapid advances in computing power and by the ever-increasing complexity of software development (e.g. [16]). These tools support the multiple tasks of the software development life cycle and aim at improving both the productivity and the quality of the software by taking care of tedious tasks like consistency checking, diagram storage, project management, etc., and by allowing code generation.

However, new objectives for OO development continually arise. In particular, technological aspects such as client/server architectures, distributed computing or parallel information systems, raise new requirements for altering the methodological basis of software development. These new needs lead to the concept of *method engineering* which concerns 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 [16]. To support this activity new computerised systems called Computer-Aided Method Engineering (CAME) tools are developed for allowing the user to correctly manage, create or customize design methods to fit a given development situation.

In this modeling and development context, we are currently implementing in Prolog a flexible CASE shell [27] supporting both tasks of method engineering and OO development. Several development methods such as [20, 6, 12] have already been implemented in the shell.

A high degree of modularity and flexibility characterizes our CASE and allows the software developer to configure the implemented methods and the ways these methods are supported with respect to the user's requirement specifications. Thus, it supports incremental method engineering: methods can be gradually customized, expanded, refined, reduced, and formalized based on users' needs and the changing demands of the development life cycle. One important characteristic of our system is the concept of *representational independence*: while the conceptual and representational constructs can combined in a flexible way in the modeling environment, they are implemented separately.

The use of Prolog (e.g. [13, 3, 2, 23]) for completely building our CASE environment is essential to our approach.

First, the declarative nature of Prolog offers, of course, several well-known advantages over conventional programming languages such as clarity, modularity, conciseness, and legibility. As a result, programs are much closer to the developer's perception of the domain and, for this reason, much easier to maintain. This is especially important in the context of large-scale programming.

Our system has a highly modular architecture to be able to manipulate different design methods, each one composed of different models. The fact that Prolog is a dynamic language with untyped variables allowed us to implement different levels of abstraction, realized by the separation of abstract and concrete primitives. *Abstract primitives* are low-level rules and primitives common to any modules, while *concrete primitives* are rules specific to a particular modules. As explained in Section 2, Prolog allows us to define an integrated three-layer architecture in which the abstract primitives are grouped in a kind of Operating System called Eroos engine while the concrete primitives, grouped in concrete model or method subsystems, are "plugged in" over the Eroos engine.

Prolog offers a powerful and expressive development environment allowing to separate concepts from algorithmic control. As will be explained in Section 6, to achieve representational independence, the fact base of our system consists of two repositories: one stores concepts and the other stores their graphical representation. These repositories were exploited by the Prolog inference engine, using the mechanisms of pattern-matching and backtracking.

Second, and more important, we use Prolog as the only language for constructing both the Graphical User Interface (GUI) and the core of the system. The aim of this paper is to concretely show that, contrary to what is most frequently believed, developing the entire application in Prolog is better and more consistent than the alternative approach of implementing the GUI in an imperative graphical development language such as VisualBasic or Visual C++ and implementing the inference engine of the system in Prolog. This alternative approach is more complex, not uniform and sometimes not elegant: for instance, in such a dual approach in Windows, the Prolog inference engine must be called through Dynamic Link Libraries (DLLs) in order to be able to use it with the running GUI.

Again, as will be shown in Sections 6 and 7, the memory management features of Prolog offer facilities for graphics and GUI development. In constrast with other imperative languages, Prolog does memory management automatically, freeing the programmer from this error-prone task. Thus, the developer always works with expressive names instead of low-levels pointers. This was a major benefit when working with the fairly complex dynamic structures of graphical applications and sophisticated user interfaces.

Third, Prolog is better suited to CASE prototype design than conventional languages: its declarative nature encourages *incremental prototyping* where code is gradually fleshed out in a top-down manner over a period of time. Rather than throwing away the prototype and re-writing the final program from scratch, incremental prototyping saves development time and costs, and allows continuous user feedback. Imperative languages are not really suited for incremental prototyping. In this context, Prolog allows us to easily implement new models and methods but also to improve and update models and methods already implemented in the tool.

The rest of the paper is structured as follows. Section 2 presents the general architecture of our CASE shell. The following sections explain and illustrate the main components of our system. Sections 3 and 4 present, respectively, model management and method management. Section 5 deals with fact base management. Section 6 is devoted to graphics management while Section 7 explains the GUI management. Section 8 deals with application management, and Sections 9 and 10 describe example sessions of already implemented models and methods. Related works on other CASE tools for OO development are discussed in Section 11. Finally, Section 12 gives conclusions and points to further work.

2 A Three Layer Structure

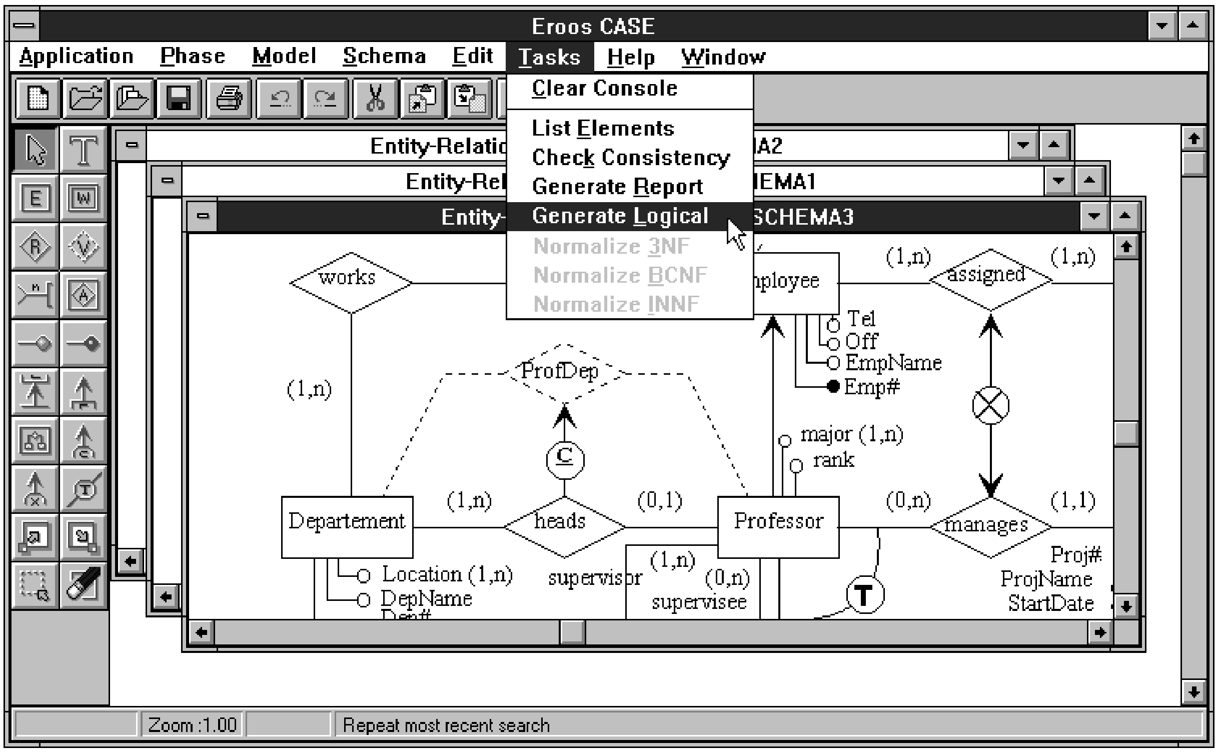


Figure 1: General layout of the Eroos Case.

As shown in Figure 1, the Eroos Case has a sophisticated GUI including graphical schema editors, Multiple Document Interface (MDI) windows, palettes, tool bars, status bars, menus, dialogue boxes etc. This leads to a user-friendly and ergonomic development environment only requiring basic familiarity with Microsoft Windows or X-Windows interface systems.

Figure 2 shows the architecture of our shell which consists of three integrated layers.

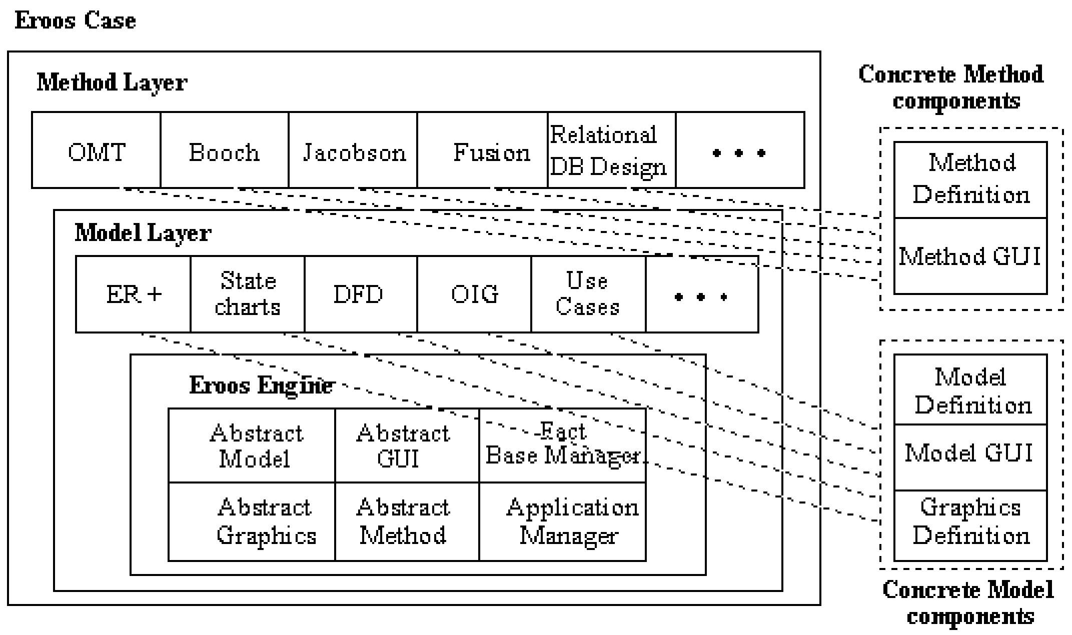


Figure 2: System architecture of the Eroos Case.

The bottom first layer is the *Eroos engine*, which is the core of the system. It implements what we call *abstract primitives*, that is, low-level primitives and rules common to all modules (models and methods) of higher layers. This first layer can be viewed as a kind of operating system providing services to the upper modules. The Eroos engine is composed of six subsystems which are explained next.

The *abstract model* subsystem defines primitives for defining concrete models (e.g., enhanced entity-relationship model) which are implemented in the second layer. Similarly, the *abstract method* subsystem contains primitives for defining concrete methods (e.g., OMT method) which are implemented in the third layer. The *abstract graphics* subsystem provides low-level primitives for defining the graphical representation of models while the *abstract GUI* subsystem provides basic rules for manipulating GUI components such as parent/child windows, menus, dialogue boxes, list boxes, buttons, and palettes. The *fact base manager* is in charge of managing and especially checking consistency of the fact base. Finally, the *application manager* takes in charge low-level application tasks such as management of projects, schemas, and reports.

The second layer (*model layer*) implements specific concrete models, such as enhanced entity-relationship (ER+), statechart, data flow diagram (DFD), object interaction graphs (OIG), and use cases. Each *concrete model* subsystem is composed of three components. It defines, in a specific *model definition* component, the conceptual language used to describe each particular aspect of the system. For example, our ER+ model defines rules and primitives for the basic ER constructs (e.g. entity, relationtionship, attributes), in addition to abstraction mechanisms such as generalization, aggregation, aggregated and derived relationships, as well as several interrelationship constraints. Two other components are associated to each concrete model subsystem: a *Model GUI* component implements GUI management primitives specific to the model while a *Graphics Definition* component contains information about the graphical representation of the model abstraction mechanisms. Each of these concrete model subsystems use the primitives provided by the Eroos engine.

The third layer (*method layer*) implements concrete methods such as OMT, Booch, Jacobson, Fusion, or relational DB design. Each method is implemented by a subsystem and uses models defined in the model layer. For instance, OMT integrates the ER+, statechart and DFD models, respectively, as static, dynamic, and functional model. Each *concrete method* subsystem is composed of two components. A *method definition* component defines primitives specific to one method such as customization rules for the method or consistency constraints between its models, while a *method GUI* contains GUI management rules associated to a particular method. A concrete method subsystem can also directly call primitives of the Eroos engine.

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

3 Model Subsystems

To each concrete model is associated a set of concepts, each defined by a predicate `concept`. This predicate has three arguments: the first one describes the concept with its arguments, the second one defines the design phase, and the third one represents the model in which the concept is used. For example, some of the concepts defined in our ER+ model are the following:

```
concept(entity('Entity Name'),'Conceptual','ER+').
concept(relationship('Relationship Name'),'Conceptual','ER+').
concept(participates('Relationship Name','Participant Name',
    'Minimal Cardinality','Maximal Cardinality','Role'),'Conceptual','ER+').
concept(attribute('Owner Name','Attribute Name','Minimal Cardinality',
    'Maximal Cardinality'),'Conceptual','ER+').
concept(identifier('Entity Name','List of Properties'),'Conceptual','ER+').
concept(weak_entity('Entity Name','Ident. Relationship'),'Conceptual','ER+').
concept(er_fd('Owner Name','Left-Hand Side','Right-Hand Side'),'Conceptual','ER+').
concept(subset_of('Left Relationship','Right Relationship'),'Conceptual','ER+').
concept(generalization('Superclass Name','total/partial','excl/overl',
    'Criteria','Defining Attribute'),'Conceptual','ER+').
concept(isa('Subclass Name','Superclass Name','Criteria','Defining Expr.'),'Conceptual','ER+').
concept(derived_relationship('Derived Relationship','Defining Relationships'),'Conceptual','ER+').
```

Different predicates define the arguments of concepts. For example, the following facts define some properties of the concept `participates`, linking an entity to a relationship:

```
domain(participates,'Participant Name',string).
domain(participates,'Minimal Cardinality',integer).
default_value(participates,'Minimal Cardinality',0).
allow_nulls(participates,'Role').
```

Predicate `domain` defines the domain of the argument, `default_value` states the default value of the argument, and `allow_nulls` specifies optional arguments.

To each concept is associated a set of integrity constraints. Integrity checking is realized at three different levels:

(1) arguments of a concept (`ic_arg`)
(2) whole concept (`ic_concept`)
(3) consistency of a model (`ic_model`).

As will be shown in Section 4, a fourth level defines constraints associated to the consistency between models, i.e., a method (`ic_method`). We next give examples of constraints associated to each of these three levels.

```
1) ic_arg(entity,‘Entity Name‘,Name) :-
     check_neg_refer_int(‘Entity Name‘,Name,relationship(Name)).
2) ic_arg(participates,‘Participant Name‘,Name) :-
     check_refer_int(‘Owner Name‘,Name,[entity(Name),relationship(Name)]).
3) ic_concept(participates(_,_,MinCard,MaxCard,_)) :-  MinCard > MaxCard,
     error_msg(‘Error‘,[‘The Maximal Cardinality must be greater than ‘,
       ‘or equal to the Minimal Cardinality.‘]),!.
4) ic_concept(participates(RN,PN,_,_,R1)) :- participates(RN,PN,_,_,R2),
     ( R1=R2,
       error_msg(‘Error‘,[‘Entity ‘,PN,‘ already participates in relationship ‘,
         RN,‘ with the same role‘])
     ; R1==‘‘,
       error_msg(‘Error‘,[‘Since Entity ‘,PN,‘ already participates in relationship ‘,
         RN,‘ then a value for role is required‘])
     ; R2==‘‘,
       error_msg(‘Error‘,[‘Entity ‘,PN,‘ already participates in relationship ‘,
         RN,‘ with no role‘])
     ),!.
5) ic_model(relationship(Rel)) :-
     findall(Part,participates(Rel,Part,_,_,_),PartList),length(PartList,N), N<2,
     error_msg([‘Relationship ‘,Rel,‘ has less than 2 participating objects.‘]).
6) ic_model(entity(Name)) :- \+(identifier(Name,List)),
     error_msg([‘Entity ‘,Name,‘ has no defined identifier.‘]).
```

The first two constraints are attached to arguments of concepts. The first one checks that the name of an entity is not the same as the name of a relationship. The second constraint checks that the participant of a participates predicate is an existing entity or relationship.

The third constraint is attached to predicate `participates`: it states that the maximal cardinality must be greater than or equal to the minimal cardinality. The fourth rule defines a more complex constraint: when an object (or a relationship) participates more than once in a relationship, each participation must have a different role.

It is important to notice that `ic_arg` and `ic_concept` rules are fired when the user wants to add or modify a fact from the concept repository using dialogue boxes as those given in Figure 5. When the user press the OK button, each argument is validated by firing the `ic_arg` rules; the third argument passed to such rules is the value typed in the edit control. When all arguments are validated individually, the `ic_concept` rules are then fired; the argument passed to such rules is the fact the user wants to add to the concept repository.

The fifth constraint checks model consistency: it verifies that each relationship has at least two participating objects. The sixth constraint, also for model consistency, checks that every entity has at least one identifier.

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

In addition, each model must define a set of cascade rules that must be enforced when the fact base is modified. These rules are defined by add_rule, modify_rule, and delete_rule. For example:

```
1) modify_rule(entity(OldName),entity(NewName)) :-
     modify_cascade(attribute(OldName,Attr,MinC,MaxC),attribute(NewName,Attr,MinC,MaxC)).
2) modify_rule(relationship(OldName),relationship(NewName)) :-
     modify_cascade(participates(OldName,PN,MinC,MaxC,R),participates(NewName,PN,MinC,MaxC,R)).
3) delete_rule(entity(Name)) :- delete_cascade(attribute(Name,_,_,_)).
4) delete_rule(entity(Name)) :- delete_cascade(identifier(Name,_)).
```

The first rule is fired when an entity name is changed: it replaces the new entity name in all associated attribute predicates. As shown by the second rule, similar modify_rule predicates exist for all interelated model concepts. The third rule is fired when an entity name is deleted: it deletes in cascade all corresponding attribute predicates while the fourth rule deletes all identifiers of an entity when this entity is deleted.

The ER+ model explained above is an example of a concrete model defined using the primitives of the abstract model subsystem of the Eroos engine. As already said, such primitives are applicable to different models. Examples of such primitives are check_refer_int, modify_cascade, and delete_cascade used by the rules of the concrete ER+ subsystem, as explained above. They are shown next.

```
check_refer_int(Field,Value,Term) :-
    not(lst(Term)),!,check_refer_int(Field,Value,[Term]).
check_refer_int(Field,Value,List) :- lst(List),not(exists_or_list(List)),
    ( List = [Term] -> error_msg([‘The ‘,Field,‘ must be an existing ‘,Term,‘.‘])
    ; List = [Term1,Term2] -> error_msg([‘The ‘,Field,‘ must be an existing ‘,Term1,
      ‘ or ‘,Term2,‘.‘])
    ; ext_pred(List,List1),error_msg([‘The ‘,Field,
        ‘ must be one of the following ‘,List1,‘.‘])
    ),!.
```

Predicate check_refer_int is used in concrete models to check – through predicate exists_or_list – referential integrity between arguments of two concepts. Another similar predicate check_neg_refer_int is used to avoid referential integrity between arguments of two concepts. As can be seen, messages in dialogue boxes are displayed when the rule is violated in order to help the user correct the fact base.

```
modify_cascade(OldTerm,NewTerm) :- OldTerm,OldTerm =.. [Pred|_],atom_string(Pred,PredStr),
    write_msg1([‘Modifying ‘,PredStr,‘ predicates ...‘]),fail.
modify_cascade(OldTerm,NewTerm) :- retract(OldTerm),assert(NewTerm), fail.
modify_cascade(OldTerm,NewTerm).

delete_cascade(Term) :- Term,functor(Term,Pred),atom_string(Pred,PredStr),
    write_msg1([‘Deleting associated ‘,PredStr,‘ predicates ...‘]),fail.
delete_cascade(Term) :- retract(Term),fail.
delete_cascade(Term).
```

Predicates modify_cascade and delete_cascade are typically abstract primitives since they manipulate predicates given as arguments. They are totally independent of concrete models and concepts.

Other examples of primitives are domain, default_value, and allow_nulls, which were used above for defining the arguments of concepts. Also, several ic_arg, ic_concept, and specific cascade rules are attached to the abstract model subsystem. For example, one of them tests that the same fact is not entered twice in the repository.

4 Method Subsystems

A concrete method associates a set of models to each phase of the development life cycle. 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 consistency between models. For example, the following 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, while the following constraint

```
ic_method :-  transition(Statech,TransEvt,Cond,Action,OrState,DesState),
  attached(Statech,Class),\+(defined_operation(Action,Class)),
  error_msg([‘Action ‘,Action,‘ is not defined as an operation of class ‘,Class ‘.‘]).
```

checks that every action defined in a statechart attached to a class is defined as an operation in the interface of that class.

Each concrete method customizes its set of models according to the method (and users') specificities. For instance, the OMT method does not use derived and subset 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 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 in Inclusion Normal Form. In the same way, in the OMT method, an operation allows to generate, from a statechart specification, the C++ code that realizes the corresponding functionality. These operations are briefly described in Sections 9 and 10.

Finally, 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 subsystem defines primitives available to all methods. Examples of such primitives are `load_method` 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 while `generate_report` is used to generate a report.

Figure 3 shows an example of interaction between abstract and concrete method subsystems for generating a report. Predicate `generate_report` is selected by the user during a development phase for a particular model (e.g., conceptual DB design and ER+). Such an abstract primitive needs to know how the report for the phase and model must be realized. As said, this is defined in concrete method subsystems.

5 Fact Base Manager

This subsystem is in charge of managing the fact base. In fact, there are two different fact 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

```
                 -------------*****************************************---------------
                                   Report for Application COMPANY
                                   Phase : Conceptual, Model : ER+
                 -------------*****************************************---------------

entity : employee {             | entity : project {               | entity : department {
  identified by :               |   identified by :                |   identified by :
    enum                        |     pnum                         |     dnum
    ename, efname               |   attributes :                   |   attributes :
  attributes :                  |     pname : card = [1,1]         |     dname : card = [1,1]
    address : card = [1,1]      | }                                | }
    zip: card = [1,1]           |                                  |
    city : card = [1,1]         |                                  |
}                               |                                  |
entity : manager {              | relationship : reports_to {      | relationship : reports_to {
  specializes :                 |   participating objects :        |    participating objects :
    employee                    |     manager : card = [1,1]       |      project : card = [1,m]
  identified by :               |     department : card = [1,m]    |      department : card = [1,m]
    mnum                        |                                  | }
  attributes :                  | }                                |
    mname : card = [1,1]        |                                  |
}                               |                                  |
                                |                                  |
relationship : participates {   | relationship : sponsored_by {    | derived relationship : m_d {
  participating objects :       |    participating objects :       |    derived from :
    employee : card = [1,1]     |      project : card = [1,1]      |      is_a, participates,sponsored_by
    project : card = [1,m]      |      department : card = [1,m]   |    participating objects :
}                               | }                                |      manager : card = [1,1]
                                |                                  |      department : card = [1,m]
                                |                                  | }
```

Figure 3: Generating a report for a development phase and a model.

```
grafix(er_window_1,123,[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)`.

As already said, this separation of both repositories allows us to achieve representational independence. However, it is necessary to have a predicate `gfx_conc` that makes the correspondence between each concept and its graphical representation. This predicate is used, e.g., when the user clicks the right button on an graphical object, to determine the corresponding concept that is to be modified.

Each graphical object has a unique Object Identifier (OID), kept in the second argument of the `grafix` predicate. OIDs are necessary in particular to maintain connections between graphical objects. For example the fact `participates(works,employee,1,1,'')`, linking `employee` with relationship `works` is represented graphically with a line connecting the corresponding square and diamond. Since all the caracteristics of the square and the diamond can change (i.e., color, size, text annotation), OIDs allow to keep track of the connection between them.

Predicates `add_element`, `modify_element`, and `delete_element` are available to modify the contents of the concept repository. 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 fact base manager subsystem contains primitives for firing integrity checks with predicates such as `check_ic_arg`, `check_ic_concept`, `check_ic_model`, and `check_ic_method`. It also contains 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 an entity or a relationship in a schema).

Predicates `assert_grafix` and `retract_grafix` are used to modify the contents of the graphics repository. As a side-effect, these predicates draw and delete the graphical objects in windows.

An important functionality concerning both repositories is to allow the undo/redo capability when drawing schemas. For this purpose, to each window are associated two stacks (undo and redo) implemented using the retract and assert predicates.

```
modify_repositories(Window,GfxObjs,NewGfxObjs,Concepts,NewConcepts) :-
  retractall(redo(Window,_,_,_,_)),
  asserta(undo(Window,GfxObjs,NewGfxObjs,Concepts,NewConcepts)),
  retract_grafix(GfxObjs,Window),assert_grafix(NewGfxObjs,Window),
  fire_del_rules(Concepts),fire_add_rules(NewConcepts).
```

The above predicate is used for every modification of the repositories. The first argument states which window is concerned. `GfxObjs` and `Concepts` are the lists of facts that has to be removed from, respectively, the graphics and the concept repositories. Similarly, `NewGfxObjs` and `NewConcepts` are the list of facts that has to be added to both repositories.

Thus, after the redo stack for the window has been cleared, an element is pushed into the undo stack, the old graphical objects are deleted, the new ones are added (using `retract_grafix` and `assert_grafix`), and the same is done for the concepts (using `fire_del_rules` and `fire_add_rules`). If the user wants to undo the operation that modified the repositories, the inverse operation is realized, the new facts are removed from the repository, and the old facts are reasserted.

Notice that the list of graphical objects/concepts to be deleted can be empty, e.g., when adding a new entity or relationship. The same is true for the list graphical objects/concepts that has to be added, e.g., when deleting an entity.

6 Graphics Subsystems

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 newly exposed areas to 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.

Prolog offers important advantages over conventional programming languages for managing graphical objects, which are often complex and large data structures. The Prolog built-in knowledge base as well as its pattern matching and backtracking mechanisms allow to realize graphical manipulations at a more abstract level. Since Prolog is a dynamically typed language, complex terms and structures representing graphical objects can be created and manipulated without knowing their type. The memory required to operate on these structures is dynamically allocated and deallocated by the Prolog run-time system. So, graphical objects can be manipulated in lists which can grow to an arbitrary length.

For realizing equivalent manipulations in other imperative languages it is necessary to use dynamic memory allocation as well as list traversals and memory pointers. Thus, programmers in C++-like languages must know everything about memory allocation at compile and run time. This low-level programming task is often hard and difficult and it requires experimented programmers. In particular, a program can be correctly compiled and nevertheless contain errors because some pointers do not point to data structures at run time.

Further, dynamic memory allocation in these conventional languages is not really manageable because C++ does not provide any form of garbage collection to reclaim the memory released by defunct objects. In the context of graphical object manipulation, this can lead to memory overloading. On the contrary, Prolog dynamically manages the system memory. Of course, such a mechanism can be programmed in imperative languages but this increases the complexity of development. Notice however that some dynamic OOPLs, for instance CLOS [9], also provide garbage collection.

It is in the development of the graphics subsystems, explained next, that many of the advantages of Prolog turned out to be extremely important.

The abstract graphics subsystem defines primitives for manipulating graphics objects independently of their shape. The particular visual appeareance and the behaviour of objects is defined by concrete graphics subsystems. For example, the ER+ graphics subsystem defines the size, color, shape, etc., of the entities, relationships, as well as the lines connecting them. Furthermore, this ER+ concrete subsystem also determines how these objects react when they are manipulated, i.e., resized, selected, dragged.

The abstract graphics subsystem was developed with an object-flavored programming style. As already said, each graphical object has a unique OID. In addition, each predicate first checks the type of the object passed as argument: if the predicate was defined for this type, the process goes on; otherwise, the inference engine backtracking mechanism tries to match another predicate with the same name and the same signature. This object-flavored programming style combined with our abstract/concrete separation allows encapsulation, an important advantage of Object-Oriented programming.

The abstract graphics subsystem contains typical low-level graphics primitives not requiring information about objects. Examples are geometric primitives for manipulating points, segments, rectangles, and so on. Some abstract graphics primitives are the following.

A predicate `eq_segm([X1,Y1,X2,Y2],[DY,DX,C])` computes the equation of the segment determined by the points (`X1,Y1`) and (`X2,Y2`). Predicate `inters_segm([A1,B1,C1],[A2,B2,C2],[X,Y])` determines the intersection point (`X,Y`) ot the two segments given as arguments. A predicate `distance([X1,Y1],[X2,Y2],Dist)` computes the distance between two points. Predicate `sq_overlap([L,T,R,B],[L1,T1,R1,B1])` tests whether two squares overlap. Predicate `sq_inters([L1,T1,R1,B1],[L2,T2,R2,B2],[L,T,R,B])` computes the intersection of two squares.

Predicate `add_connection(Id1,[Id2,X,Y])` allows to add to object `Id1` a connection with object `Id2` at point `(X,Y)`. It checks that the point (`X,Y`) belongs to the object `Id`. Other similar rules are `del_connection`, `modify_connection`, and `retrieve_connection`.

Predicate `extent` returns the physical extent of a graphics sequence, while predicate `shift_args` shifts the graphics elements by a specified offset. There are also predicates for manipulating the "select" corners such as `draw_corners`, `delete_corners`. A predicate `test_objects` is used to select the objects enclosed by a rectangle drawn on the screen; these objects can then be moved, deleted, or resized. Other predicates show and store status information, scroll the image, move (drag) or resize a set of selected objects, magnify and reduce the scale (zoom-in and zoom-out) of a window.

To facilitate the manipulation of lines a predicate `incertitude` determines the sensibility area of the mouse cursor around a point. Indeed, a problem that arises when manipulating lines is that it is difficult to place the cursor exactly over the line (i.e., over the exact pixel). Therefore, a sensibility zone around this line is declared with predicate `incertitude`.

Concrete graphics subsystems are defined for each model defining how objects are drawn, moved, selected, resized, shifted, etc. Some of the primitives for the concept `entity` are explained next.

```
conc_color(entity,‘Conceptual‘,‘Object‘,0,0,0).
gfx_extent(Gfx,L,T,R,B) :- Gfx=[_Brsh,_Pen,rect(L,T,R,B),_Font,_Trns,_Fore,_Text],!.
hit(Gfx,X,Y) :- Gfx=[_Brsh,_Pen,rect(L,T,R,B),_Font,_Trns,_Fore,_Text],
  L =< X, T =< Y, R >= X, B >= Y,!.
gfx_conc(_,_,Gfx,entity(Ent)) :- Gfx=[_Brsh,_Pen,rect(_,_,_,_),_Font,_Trns,_Fore,text(_,_,Ent)],!.
shift_object(Gfx,Dx,Dy,NewGfx) :- Gfx = [Brsh,Pen,rect(|Corners),Font,Trns,Fore,Text(X,Y,Name)],
  shift_args(Corners,Dx,Dy,NewCorners), shift_args([X,Y],Dx,Dy,[X1,Y1]),
  NewGfx=[Brsh,Pen,rect(|NewCorners),Font,Trns,Fore,Text(X1,Y1,Name)],!.
outline_object(Gfx,OutGfx) :- Gfx = [_Brsh,_Pen,rect(|Corners),_Font,_Trns,_Fore,_Text],
  OutGfx = [pen(0,0,255,-2),rect(|Corners)]),!.
```

Predicate `conc_color` states the color (a combination of R,G,B) used for drawing concepts on the screen: e.g., entities are drawn in black. Predicate `gfx_extent` states the smaller square enclosing the graphical representation of concepts: for entities this is determined by the corners of the rectangle. Predicate `hit` determines whether a point (`X,Y`) is situated over a graphical object. Predicate `gfx_conc` allows to establish the correspondance between a concept and its graphical representation: for entities it suffices to see the text annotation inside the square. Predicate `shift_object` determines how to shift objects: for entities both the corners of the rectangle and the point (`X,Y`) of the text annotation must be shifted (with `shift_args`). Finally, predicate `outline_object` states how objects are outlined when they are being dragged: for entities, a dotted blue square is drawn on the screen.

Figure 4 shows an example of how the abstract and the concrete graphics subsystems work together. In Figure 4 (a), the user selects an area in the screen with the mouse cursor. All objects enclosed in the area are selected. For this, the primitive for selecting an area (defined in the abstract subsystem) determines the graphical objects to be selected.

Then, the dragging operation can be applied as shown in Figure 4 (b). During the dragging operation, at each mouse move, the outlines of the objects to be moved must be drawn. As explained above, the concrete subsystems define how the objects are outlined. 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 4 (c). Here also the abstract shifting primitive uses the definition of how the different objects are shifted, as defined in the concrete subsystems.

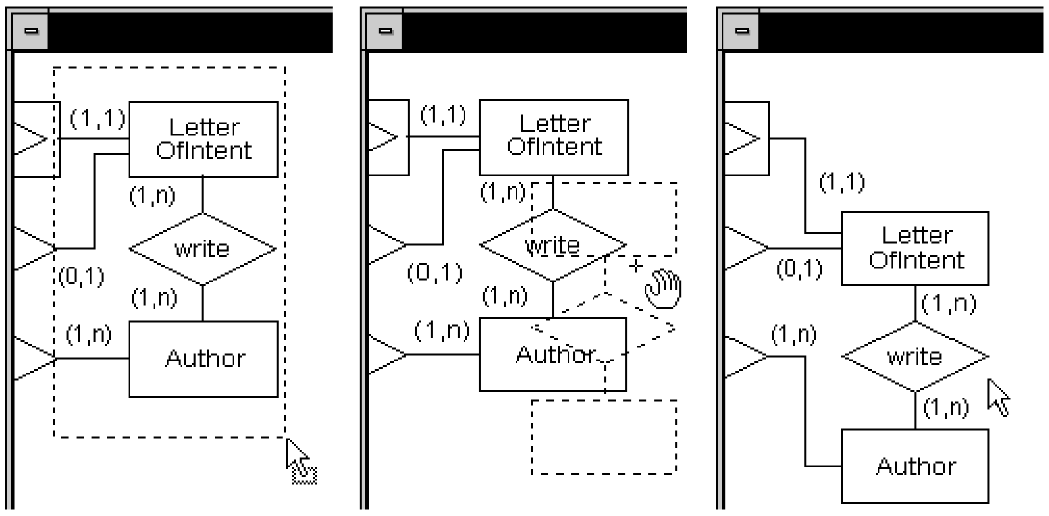


Figure 4: Steps of a dragging operation.

7 GUI Subsystems

The abstract GUI subsystem comprises primitives for manipulating GUI elements, such as menus, dialogue boxes, palettes, status bars and toolbars. This includes, e.g., adding, removing, selecting and retrieving elements from listboxes, editing and retrieving edit controls, selecting radio buttons, and manipulating palettes.

A concrete GUI component is associated to each model and to each method. Such components use the primitives provided by the abstract component. For instance, the general menus of the Eroos Case as well as the `Tasks` menu associated to the ER+ schema editor (shown in Figure 1) are defined using the predicate `menu_item` as follows:

```
menu_item(main_menu,‘&Application‘,appl_menu,enable,true).
menu_item(main_menu,‘&Phase‘,phase_menu,enable,true).
menu_item(main_menu,‘&Model‘,model_menu,enable,true).
menu_item(main_menu,‘&Schema‘,schema_menu,enable,true).
menu_item(main_menu,‘&Edit‘,edit_menu,enable,true).
menu_item(main_menu,‘&Tasks‘,tasks_menu,enable,true).
menu_item(main_menu,‘&Help‘,help_menu,enable,true).
menu_item(main_menu,‘&Window‘,window_menu,enable,true).

menu_item(tasks_menu,‘&Clear Console‘,1000,enable,clr_console).
menu_item(tasks_menu,‘‘,0,separator,true).
menu_item(tasks_menu,‘List &Elements‘,1001,enable,lst_elements).
menu_item(tasks_menu,‘Chec&k Consistency‘,1002,enable,chk_consist).
menu_item(tasks_menu,‘Generate &Report‘,1003,enable,gen_report).
menu_item(tasks_menu,‘Generate &Logical‘,1004,enable,gen_logical).
menu_item(tasks_menu,‘Normalize &3NF‘,1005,disable,norm_3NF).
menu_item(tasks_menu,‘Normalize &BCNF‘,1006,disable,norm_BCNF).
menu_item(tasks_menu,‘Normalize &INNF‘,1007,disable,norm_INNF).
```

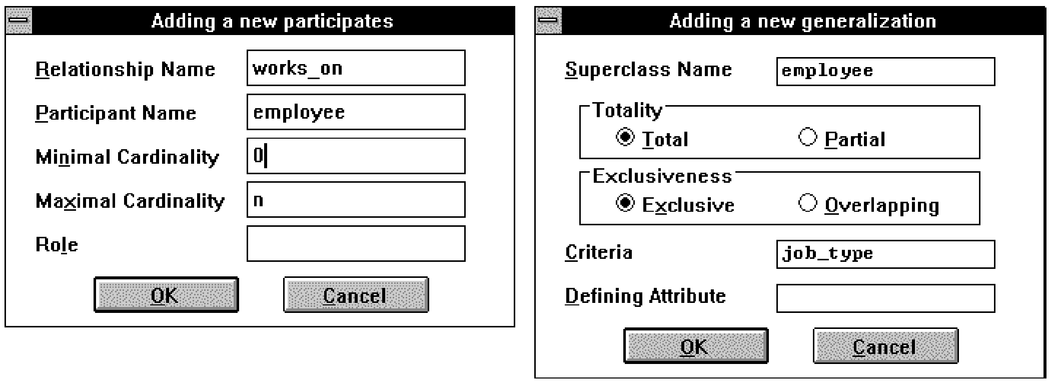
For each menu item, the following are specified: parent menu, text to display, either the identifier of the message generated by Windows or the submenu, whether the menu is enabled, and the predicate to execute when the menu is selected.

Other examples of concrete rules are palette definitions for a specific schema editor. The ER+ palette given in Figure 1 is defined as follows:

```
palette(er_palette,10,2,30,30).
default_btn(er_palette,arrow_btn).
palette_btn(er_palette,1,1,arrow_btn,two_state,select_object,‘Ready‘,0).
palette_btn(er_palette,1,2,text_btn,two_state,add_text,‘Add Text Annotation‘,text_cursor).
palette_btn(er_palette,2,1,entity_btn,two_state,add_entity,‘Add Entity‘,entity_cursor).
palette_btn(er_palette,2,2,weak_ent_btn,two_state,add_weak_entity,‘Add Weak Entity‘,weak_entity_cursor).
palette_btn(er_palette,3,1,relation_btn,two_state,add_relation,‘Add Relationship‘,relation_cursor).
palette_btn(er_palette,3,2,der_rel_btn,two_state,add_der_rel,‘Add Derived Relationship‘,der_rel_cursor).
palette_btn(er_palette,4,1,partic_btn,two_state,add_partic,‘Add Participation‘,part_rel_cursor).
palette_btn(er_palette,4,2,aggr_rel_btn,two_state,add_aggr,‘Add Aggregated Relationship‘,aggr_rel_cursor).
[...]
```

The first line defines the ER+ palette as having 10 × 2 buttons, each of 30 × 30 pixels. Then each of the buttons is defined with predicate `palette_btn`. It defines the bitmap to paint over the button, whether the button is one- or two-state, the predicate to execute when the button has been pressed, the help message painted in the status bar, and the cursor that has to be displayed.

As can be imagined, 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 subsystem, they are the same for all the palettes. In addition, the abstract GUI subsystem also defines generic dialogue boxes for n-ary predicates. For example, the dialogue box associated to predicate `participates` is shown in Figure 5 (a).


Adding a new participates

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

Adding a new generalization

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

Figure 5: Two dialogue boxes for the ER+ editor.

These generic dialogue boxes provide basic functionality for modifying the fact base. This is extremely useful in the first stages of 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 several concepts, one of which is shown in Figure 5 (b).

8 Application Manager

This last subsystem is a classical module of any computerized tool: it is in charge of the management of applications and files, and requires no further explanations. For applications, it has predicates such as `new_application`, `open_application`, `save_application`, and `close_application`.

Each application is composed of multiple files. For example, a large application will define an object model represented by a set of separate ER+ schemas, a dynamic model represented by a set of statechart schemas associated to the different classes and so on.

Therefore, the Application Manager subsystem also has predicates such as `new_schema`, `open_schema`, `save_schema`, and `close_schema`.

9 Model Example Sessions

This section briefly presents example sessions of two models (ER+ and statechart) implemented in the Eroos Case. We refer to [10, 12, 26] for a complete description of these implementations.

In addition to the basic ER concepts (entity, relationship, attribute, identifier and cardinality), our implementation of the ER+ model [12] takes into consideration some important abstraction mechanisms such as generalization, aggregation, derived and aggregated relationships, and several interrelationship constraints. It subsumes any object model of the well-known development methods.

ER+ schemas, such as the University schema shown in Figure 6, are drawn on the screen using the GUI. As explained in Section 3, they are validated with the integrity constraints defining the syntax and semantics of the ER+ abstractions.

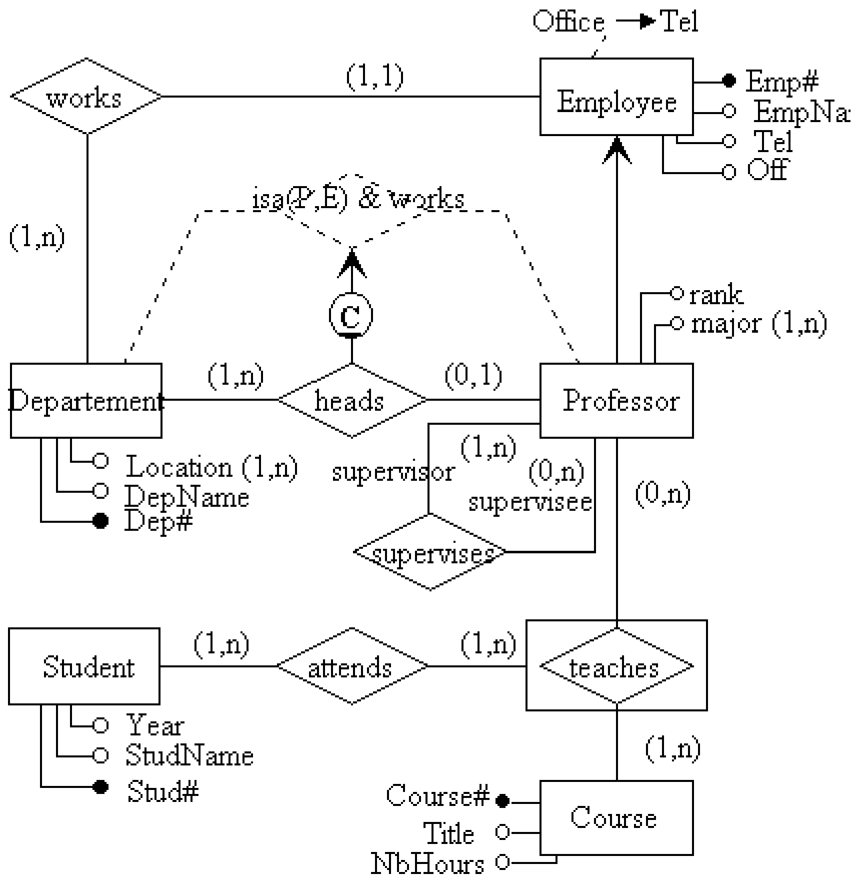


Figure 6: An ER+ schema for an hypothetical university

Prolog predicates encoding the graphical schemas are automatically generated. Figure 7 shows the fact base corresponding to the University ER+ schema specifications. These predicates 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 in Section 10.

```
entity(depart).                     entity(course).
attribute(depart,depNo,1,1).        attribute(course,courseNo,1,1).
attribute(depart,depName,1,1).      attribute(course,title,1,1).
attribute(depart,location,1,n).     attribute(course,nbHours,1,1).
identifier(depart,[depNo]).         identifier(course,[courseNo]).
entity(employee).                   entity(professor).
attribute(employee,empNo,1,1).      attribute(professor,status,1,1).
attribute(employee,tel,1,1).        attribute(professor,major,1,n).
attribute(employee,empName,1,1).    entity(student).
attribute(employee,office,1,1).     attribute(student,studNo,1,1).
identifier(employee,[empNo]).       attribute(student,studName,1,1).
er_fd(employee,[office],[tel]).     attribute(student,year,1,1).
identifier(student,[studNo]).
relationship(teaches).                 relationship(attends).
participates(teaches,professor,0,n,''). participates(attends,student,1,n,'').
participates(teaches,course,1,n,'').    participates(attends,teaches,1,n,'').
relationship(works).                    relationship(isaWorks).
participates(works,employee,1,1,''). participates(isaWorks,professor,1,1,'').
participates(works,depart,1,n,'').      participates(isaWorks,depart,1,n,'').
relationship(heads).                    participates(heads,professor,0,1,'').
participates(heads,depart,1,n,'').      relationship(supervise).
participates(supervise,professor,0,1,supervisor).
participates(supervise,professor,0,n,supervisee).
generalization(employee,partial,exclusive,'','').isa(professor,employee,'','').
derived_relationship(isaWorks,[professor,works]). subset_of(heads,isaWorks).
```

Figure 7: Prolog facts representing the University ER+ schema.

Statecharts [7] are an extension of finite state machines and diagrams, with formal syntax and semantics used for describing system dynamics. Although statecharts were originally proposed for describing hardware devices, the dynamic models of OO methods, such as OMT and Booch, use the statechart formalism for modeling the temporal behaviour of a system.

Our implementation of the statechart model [26] takes into account most of the statechart concepts such as events, states, super- and substates, transitions (simple, merging, concurrent, and splitting), conditions, actions, and history enforcement. The module has the following functionalities. The dynamic behaviour of a system is specified as a collection of interacting statecharts, each one attached to a class. As with the ER+ model, these statecharts can be drawn on the screen via the GUI and are validated with the 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.

Figure 8 shows the layout of the statechart schema editor. The tasks menu is used for dynamic simulation. The first option allows to start an object of a class by starting its associated statechart. Several objects of the same class can be created, each one executing its own statechart. The second option of the menu allows to stop an object. The third option shows the status of an object, i.e., the active states of its associated statechart. The fourth option allows to send an event to an object. The fifth option enables or disables the predicate printing the status of an object after transitions.

Our C++ code generation for statechart classes is based on an inheritance hierarchy comprising classes which can be partitioned in two groups: the *statechart engine classes* and the *application classes*. The first group implements the semantics of the statecharts

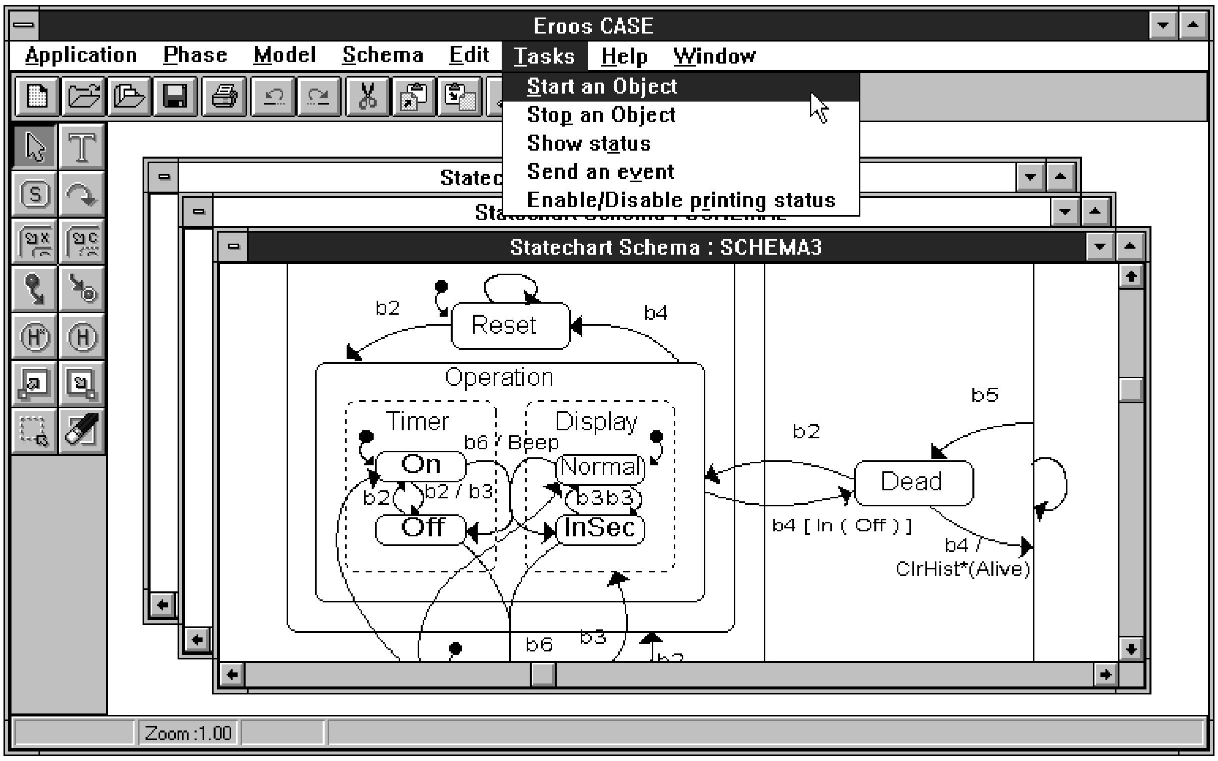


Figure 8: Menu for the dynamic simulation.

independently of any particular application while the second group is specific to the application. For additional details of the statechart implementation in C++, we refer to [25].

To illustrate this C++ code generation, we explain the two top-level predicates generating the C++ header file and the C++ code file.

```
gen_header_cpp(Class,StateCh,HFile) :- tell(HFile),
    write('#ifndef _H_'),writeU(StateCh),nl,write('#define _H_'),writeU(StateCh),nl,
    write('#include <stddef.h>'),nl,write('#include "statech.h"'),nl,nl,
    write('class '),writeU(StateCh),write(';'),nl,nl,
    gen_g_state(Class,StateCh),get_states(StateCh,StateList),
    gen_state_headers(StateList,Class,StateCh),nl,
    gen_statech_header(StateList,StateCh),nl,write('#endif'),nl,
    told,tell(user).
```

Predicate `gen_header_cpp` generates the C++ header file. After writing the preamble in the header file (`HFile`), it generates the header of the generic state class (`gen_g_state`), constructs the list of states of the statechart (`get_states`), generates the header for each state class (`gen_state_headers`), and generates the header of the statechart class (`gen_statech_header`).

For instance, the header of the statechart class for a digital watch, generated by `gen_statech_header`, is shown next.

```
class WatchSC: public StChart {
public:
  MainState Main;           AliveState Alive;   DeadState Dead;
  StopwatchState Stopwatch; HourState Hour;     ResetState Reset;
  OperationState Operation; TimerState Timer;   DisplayState Display;
  NormalState Normal;       InsecState Insec;   OnState On;
  OffState Off;
  void reactEvent(int Event);
  WatchSC(); // constructor
};
```

Predicate gen_code_cpp generates the C++ code file.

```
gen_code_cpp(Class,StateCh,HFile,CFile) :- tell(CFile),
    write('#include <stdio.h>'),nl,write('#include <iostream.h>'),nl,
    write('#include "statech.h"'),nl,write('#include "'),write(HFile),write('"'),nl,nl,
    get_states(StateCh,StateList),
    gen_state_impl(StateList,Class,StateCh),
    gen_sc_constr(StateList,StateCh),told,tell(user).
```

After writing the preamble in the code file (CFile), it calls gen_state_impl for generating the implementation of all the states of the statechart, and calls predicate gen_sc_constr to generate the constructor of the statechart class.

10 A Method Example Session

This section deals with an example session of the relational database design (RDBD) method implemented in the Eroos Case. We refer to [10, 11, 12] for a complete description of this implementation.

Nowadays, relational database design typically goes through, first, conceptual (ER) schema design and, second, translation into a relational schema. One of the originalities of our RDBD method consists in generating database schemas normalized in Inclusion Normal Form (IN-NF) [15]. Unlike classical normal forms that only characterize individual relation schemas, this design criterion removes redundancies (superfluous attributes) in the database schema considered as a whole.

In practice, the RDBD method works in the Eroos Case as follows. First, an application is specified by a set of ER+ schemas customizing the ER+ module exposed above. Then, the relational translation process takes as input the concept repository encoding the ER+ specifications, such as the one shown in Figure 7, and generates a relational database in IN-NF. The method uses an integrated three-step algorithm:

- ER+-to-relational mapping,
- Third normal form normalization of individual relations and key generation,
- database normalization in Inclusion Normal Form.

At each step, new concepts are generated and introduced into the concept repository. Thus, reports can be provided to the user at any design phase. Figure 9 gives the final fact base resulting from our RDBD method. These facts correspond to the IN-NF relational database schema given in Figure 10.

This schema can be provided to the user in a clear and readeable text similar to the one shown in Figure 3. Notice IN-NF normalization consequences: relation teaches was removed because it is redundant with respect to relation attends. Also, attribute depNo is detected as superfluous in heads, and thus rel_attrs(heads,[empNo,depNo]) is replaced

```
rel_attrs(depart,[depNo,depName,location]).      rel_attrs(works,[empNo,depNo]).
rel_attrs(employee_a,[empNo,tel,empName]).   rel_attrs(employee_b,[tel,office]).
rel_attrs(student,[studNo,studName,year]).    rel_attrs(isaWorks,[empNo,depNo]).
rel_attrs(attends,[empNo,courseNo,studNo]).  rel_attrs(professor,[empNo,major]).
rel_attrs(course,[courseNo,title,nbHours]).           rel_attrs(heads,[empNo]).
rel_attrs(supervisor,[supervisor_empNo,supervisee_empNo]).

key(depart,[depNo]).   key(employee_a,[empNo]).  key(employee_b,[office]).
key(student,[studNo]). key(course,[courseNo]).     key(professor,[empNo]).
key(works,[empNo]).    key(heads,[empNo]).          key(isaWorks,[empNo]).
key(depart,[depNo]).   key(employee_a,[empNo]).  key(employee_b,[office]).
key(student,[studNo]). key(course,[courseNo]).     key(professor,[empNo]).
key(works,[empNo]).    key(heads,[empNo]).          key(isaWorks,[empNo]).
key(attends,[studNo,empNo,courseNo]).

fd(employee_a,[empNo],[empName]).            fd(depart,[depNo],[depName]).
fd(employee_a,[empNo],[tel]).                fd(depart,[depNo],[location]).
fd(employee_a,[empNo],[office]).             fd(student,[studNo],[studName]).
fd(employee_b,[office],[tel]).               fd(student,[studNo],[year]).
fd(course,[courseNo],[title]).               fd(course,[courseNo],[nbHours]).
fd(professor,[empNo],[major]).               fd(works,[empNo],[depNo]).
fd(isaWorks,[empNo],[depNo]).

view(isaWorks,[professor,works]).

ind(attends,[studNo],student,[studNo]).  ind(student,[studNo],attends,[studNo]).
ind(attends,[courseNo],course,[courseNo]).    ind(works,[depNo],depart,[depNo]).
ind(course,[courseNo],attends,[courseNo]).    ind(depart,[depNo],works,[depNo]).
ind(works,[empNo],employee_a,[empNo]).       ind(isaWorks,[depNo],depart,[depNo]).
ind(employee_a,[empNo],works,[empNo]).    ind(isaWorks,[empNo],professor,[empNo]).
ind(attends,[empNo],professor,[empNo]). ind(professor,[empNo],employee_a,[empNo]).
ind(heads,[empNo],professor,[empNo]).
ind(heads,[empNo,depNo],isaWorks,[empNo,depNo]).
ind(employee_a,[tel],employee_b,[tel]).    ind(employee_b,[tel],employee_a,[tel]).
```

Figure 9: Prolog facts representing the University relational database schema in IN-NF.

by `rel_attrs(heads,[empNo])` in the final concept repository. These redundancies are not detected by classical normalization (i.e., third normal form) but are a frequent source of update anomalies and data incoherences.

In particular, the latter redundancy is associated to the ER cycle implying relationships `heads` and `works` and derived relationship $\texttt{isa}(\texttt{P},\texttt{E}) \wedge \texttt{works}$ in Figure 6. In this kind of cycle comprising an inclusion constraint (represented by the arrow labeled with a $\subseteq$), some information can be deduced in more than one way. While these cycles are semantically correct, they may be sources of relational redundancies. Such redundancies must be deleted in the corresponding relational database schemas.

11 Related Works

Research on automating software development with computerized tools has been growing since the late 80's. This section describes some representative CASE shells, comparing them with the Eroos Case. However, none of them is implemented in Prolog.

OOTher CASE [24] is a documentation development package for the following methods/notations: Coad/Yourdon's OOA method, state machine diagrams, and Jacobson's use case diagrams and object interaction diagrams. The system is developed in C++. It checks consistency between diagrams, and verify consistent naming of objects and methods.

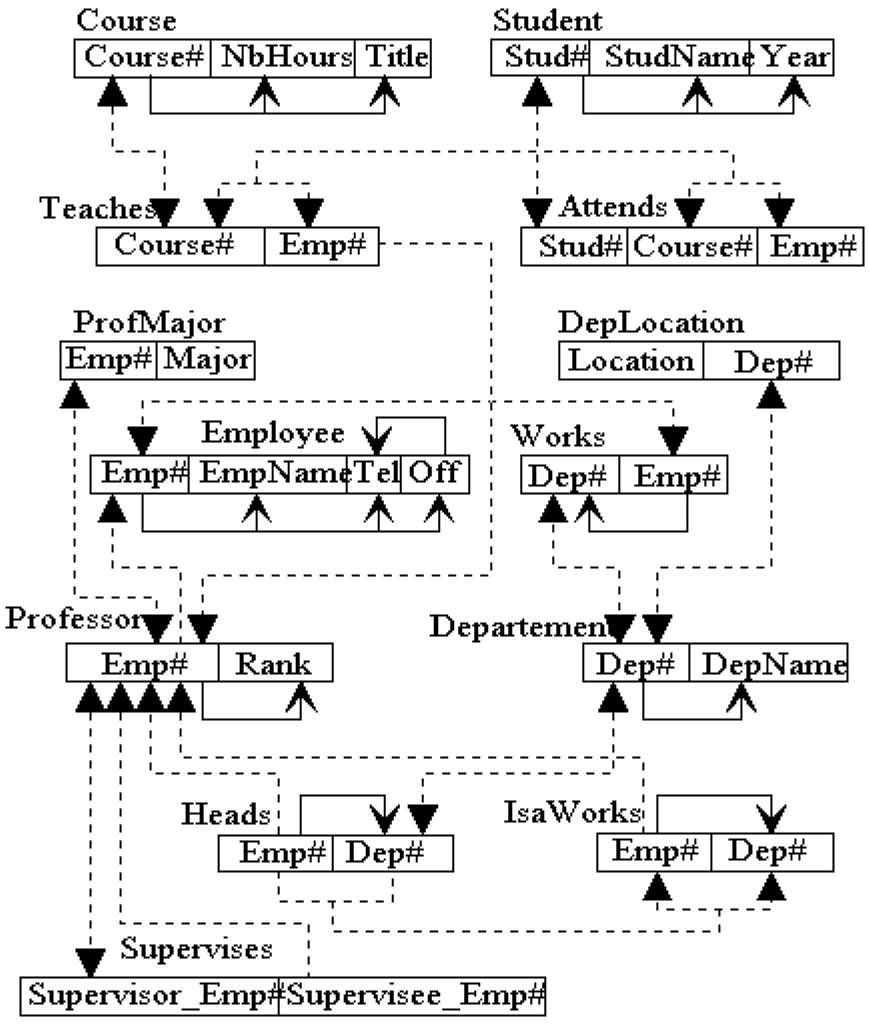


Figure 10: The University relational database schema in IN-NF

Like the Eroos Case, the tool is built around graphical editors and can provide documentation for all objects, attributes, methods and associations between objects. C++ headers may be automatically generated from the data schemas – this ensures consistent naming of member functions and attributes. However, the functionality provided by the OOTher CASE is rather limited with respect to that of the Eroos Case.

ObjectMaker [14] addresses both project and CASE requirements and supports several well-known object-oriented and structured methods: ER, statecharts, DFD, uses cases, Petri nets, Booch, Coad/Yourdon, Fusion, Martin/Odell, Shlaer/Mellor and others. The system architecture lets the developer use methods and integrate them with other tools, databases and frameworks. It also allows the users to develop their own code generators and reverse tools for the language of their choice.

Like the Eroos Case, ObjectMaker allows the designer to select and customize design methods to match the specific development environment. Language modules provide both code generation and reverse engineering (automatic diagram generation from source code). ObjectMaker's reverse engineering capability can facilitate reuse of existing source code, providing automatic documentation and supporting the maintenance and re-engineering of legacy systems. The Eroos Case provides most of the functionality of ObjectMaker, excepted the reverse engineering capability. Due to the modularity of our system it should be relatively easy to introduce such capability in the Eroos Case.

MetaEdit+ [18] is a metaCASE, developed in Smalltalk, supporting multiple methods in a client-server environment. CASE tool support for developer's own or customized methods can be built with the method workbench. The system is based on the concept of *representation independence* which is similar to ours and uses the concept of what they call the graph-object-property-role-relationship (GOPRR) metamodel. The GOPRR metamodel provides modeling tools for most of the widely used system development models and methods. It can support recursive structures for objects, OO hierarchies, interconnected models and methods, polymorphic modeling concepts, multiple representation of the same underlying conceptual description and rules for checking model integrity. This metamodel forms the basis on which various representations of data, including graphical diagrams but also hypertext and matrix, are built. Some of the OO methods supported by the tool are OMT, Fusion, Booch, Coad/Yourdon, Shlaer/Mellor, Moses. Again, the Eroos Case provides most of the functionality of MetaEdit+, excepted client-server support. We are currently making a comparison of our metamodel and the GOPRR metamodel to assess their corresponding expression power.

12 Conclusion

Since the recognition of the software crisis in the mid-70's, researchers have started to design methods that help analysts and developers construct computerized applications of increasing complexitiy. Object-oriented methods aim to cope with the inadequacy of structured methods. This led to a new way of thinking about problems using models organised around a fundamental construct: the object. However, due to the ever-increasing complexity of software development, researchers have also focused on providing CASE and CAME environments helping designers build applications in a more flexible way.

In this context, our goal was to show how Prolog can be used for completely constructing a CASE shell supporting both method engineering and 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 shell was conceived with a highly modular architecture. The adaptability of the Eroos Case helps the developer select and tailor analysis and design methods to match the specific development environment. Thus, different abstraction mechanisms and models can be customized and combined by the user, respectively, into a model and a method. Again, as the developer updates and improves his/her processes and methods, our system provides an incremental and extensible software engineering environment to fit the application. In this context, an important feature of our CASE shell is the representational independence between the conceptual constructs and their (graphical) representation. These features enable developers to be productive throughout the development life cycle, from analysis and design to post-delivery maintenance.

Second, we showed that the use of Prolog as the only development language for building an entire GUI is viable. We firmly believe that the use of only Prolog for developing the core of the system and the GUI is more coherent than using Prolog for the inference engine and a conventional imperative language for the GUI.

More important, we showed the ease of using Prolog to develop a graphical environment. For manipulating complex data structures like graphical objects, Prolog frees the developer from low-level tasks such as (dynamic) memory allocation and pointer management. We adopted an object-flavoured programming style using Prolog built-in backtracking and pattern-matching mechanisms, to which we added a unique Object Identifier mechanism. All these benefits lead to rapid prototyping also favoured by the incremental development environment of Prolog which is not allowed by imperative programming languages.

Several issues need to be further addressed. Our CASE shell is currently developed on a PC platform using a Windows LPA Prolog compiler. We are planning to port the system to Unix and/or Linux 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. Thus, only the OS function calls should be modified.

Our CASE shell provides a powerful testing environment to develop and implement new modeling abstractions and conceptual languages describing an application throughout the development life cycle. For instance, we already formalized a mechanism for the description of object models, called *materialization* [19]. We are also formalizing 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 statechart formalism.

References

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

[2] I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1986.

[3] W. Clocksin and C. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.

[4] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, second edition, 1991.

[5] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice Hall, 1991.

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

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

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

[9] S. Keene. *Object-Oriented Programming in COMMON LISP: A programmer's Guide to CLOS.* Addison-Wesley, 1989.

[10] M. Kolp. Elaboration d'un système expert pour la conception de bases de données relationnelles à partir du modèle entité-association. Mémoire de Licence Spéciale en Sciences

de l'Information et de la Documentation, INFODOC, Université Libre de Bruxelles, Sept. 1994.

[11] M. Kolp and E. Zimányi. Enhanced ER to relational database design and its implementation in Prolog. Technical Report TR-95/01, INFODOC, Université Libre de Bruxelles, Belgium, Apr. 1995. Submitted for publication.

[12] M. Kolp and E. Zimányi. Relational database design using an ER approach and Prolog. In S. Bhalla, editor, *Proc. of the 6th Int. Conf. on Information Systems and Management of Data, CISMOD'95*, LNCS 1006, pages 214–231, Bombay, India, Nov. 1995. Springer-Verlag.

[13] R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[14] Leverage Technologists Inc. Objectmaker. At http://stout.levtech.com/levtech-marketing, 1995.

[15] 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.

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

[17] J. Martin and J. Odell. *Principles of Object-Oriented Analysis and Design.* Prentice Hall, 1992.

[18] MetaCase Consulting. Metaedit+. At http://www.jsp.fi/metacase, 1995.

[19] 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, Santiago, Chile, 1994. Morgan Kaufmann.

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

[21] S. Shlaer and S. Mellor. *Object-Oriented Systems Analysis: Modelling the World in Data.* Prentice Hall, 1988.

[22] S. Shlaer and S. Mellor. *Object Lifecycles: Modelling the World in States.* Prentice Hall, 1992.

[23] L. Sterling and E. Shapiro. *The Art of Prolog - Advanced Programming Techniques.* MIT, 1986.

[24] R. Zielinski. Oother case tool 1.06f. At http://cuiwww.unige.ch/OSG/OOinfo/FAQ/oo-faq-S-11.19.0.1.html, 1994.

[25] E. Zimányi. Statecharts and their implementation in C++. Technical Report TR-94/01, INFODOC, Université Libre de Bruxelles, Belgium, Apr. 1994. Submitted for publication in *Journal of Object-Oriented Systems.*

[26] E. Zimányi. Statecharts and object-oriented development: a CASE perspective. In *Proc. of the 3rd Int. Conf. on Practical Application of Prolog, PAP'95*, pages 697–718, Paris, France, Apr. 1995.

[27] E. Zimányi and M. Kolp. Using Prolog to implement a CASE shell for Object-Oriented development. In *Proc. of the 8th Symposium and Exhibition on Industrial Application of Prolog, INAP'95*, pages 41–48, Tokyo, Japon, Oct. 1995.