On programming an interactive graphical application in logic

Manuel João Próspero

Paulo Jorge Pereira

Computer Science Department
New University of Lisbon (UNL)

ABSTRACT

A hierarchical graphical modelling system and a dialogue control mechanism were designed and implemented in first order predicate logic. Some main concepts are introduced in this paper by analysing a given application: a flat drawing design where the user is able to graphically specify the intended configuration and get important information about the existing restrictions.

Introduction

Programming an interactive graphical application is a task where one must pay attention to a lot of details which are really not at the level of the application problem itself. In that sense, the programmer may be disturbed in his/her activity and, being the output for a given input the main goal, the result is usually a poor user interface. This judgment includes both classes of users, the novice and the expert, respecting the application domain.

The improvement of the user interface is hard to do due to a lack of program structure at that high level. Either the application development stage or the subsequent maintenance are usually hard to carry out. Separating the dialogue from the functional part of a given application is a widely recognized necessity [ENC82]. Programs that are built in this way are more flexible than the remaining ones.

The main problem concerning the user is that he/she is usually considered as a simple input/output device such that an easy straight forward and uniform programming style dealing with the communication is implemented. This situation often happens when a general purpose programming language is used. A better solution is to introduce a higher programming level, where a dialogue is easily specified. For this purpose, there are many good systems available on the market today [ERO87]. However, in spite of being well supported by existing language formalisms, using those systems is not an easy work for the programmer in what concerns the application interface. This is normally done with traditional procedure calls, so that the interface must have knowledge about the application's data structures.

Where complex data structures are needed, the intentional description for entities usually adopted with general purpose procedural languages has an important drawback. In fact, the addition of new properties, when not foreseen at the very beginning, also implies the deletion of the old data. The number of fields in a Pascal record is a good example of that. This situation is a strong limitation, specially for design purposes.

On the contrary, our approach is to introduce an extensional description based on a logic interface by programming in Prolog a system to deal with modelling and user dialogues. An early and coarse version was first introduced in [PRÓ86].

In the following sections we will try to show some interesting features of the system evaluated on the basis of an actual application problem.

Presenting the Entities Model

The primary abstraction is the usual *entity* concept. For the sake of graphical representation of entities, a fully instantiated Prolog term is called a *family*. Whenever free variables occur in such a term we refer it as a *template*. Since a very large number of entities can be decomposed into others, the same thing applies to families. This means that a

picture on the screen may be derived from a hierarchy of families and the corresponding graphics.

A *scene* is a meta-object so that a specific set of images can be grouped together in the same viewport, that is, the root of a hierarchy of entities.

In the example from figure 1 four templates are shown which have free logical variables in the arguments. The meaning of that is the definition of the kind of relations expected by the application program. Each relation corresponds to a link in the graph and is to be stored in the logical data base as a fact with an associated transformation matrix.

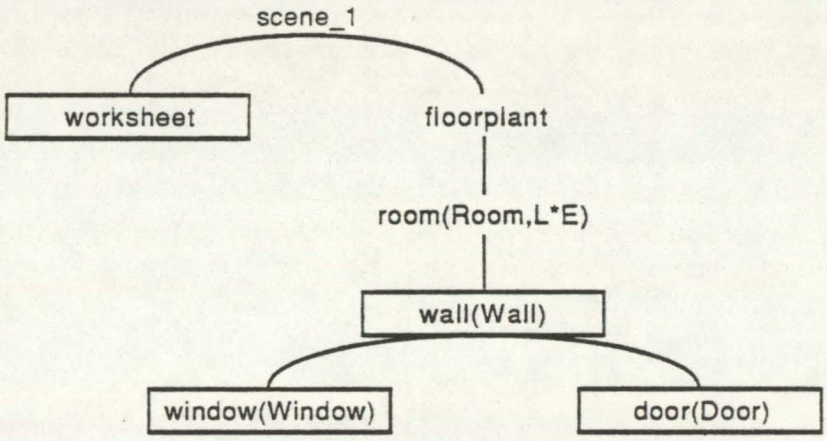

Fig. 1 — Hierarchy with direct graphics objects.

Instance transformation matrices play a very important role in our system, as they are used to construct path names in the hierarchy through the transitive closure given by a *part_of* predicate. This method is not usual in other systems, as it is only feasible within symbolic programming where a matrix can also be seen as a simple name.

Rectangular boxes stand for families with a proper definition of graphical output. In figure 1 only direct graphics objects are present, while symbolics graphics appear in figure 2 [PFA85]. As it can be seen, a different scene was designed for this purpose.

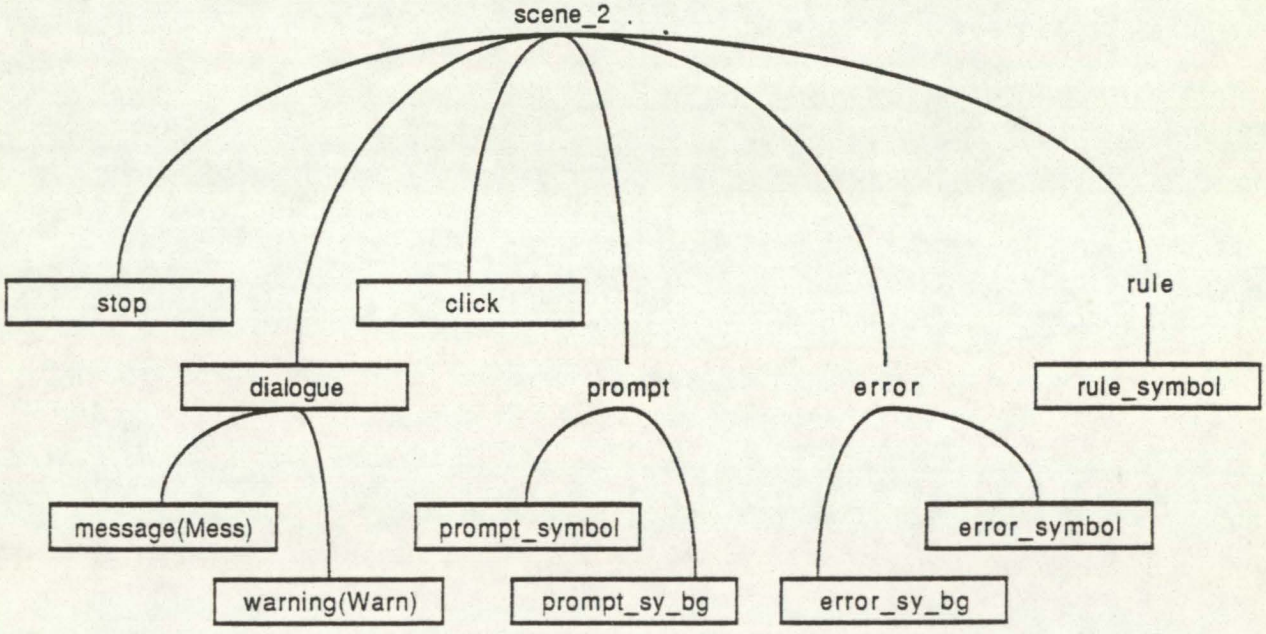

Fig. 2 — Hierarchy with symbolics graphics objects.

Dynamic manipulation of links is possible by using general high-level predicates. *Part_of*, *create_link*, *delete* and *display_hierarchy* are some of such predicates referred in

this paper. For instance, if the display of an error message is wanted, the necessary connections between preexistent descriptions are provided by the following goals:

```
create_link(error,scene_2,_,_),
create_link(dialogue,scene_2,_,_),
create_link(message(Error_Message),dialogue,_,_),
create_link(click,scene_2,_,_).
```

Anonymous variables are used here in place of the two last arguments of each subgoal since there are no attribute values to establish.

A waiting situation is then generated until the user clicks a light-button. After that, the error message must be erased from the screen. This can be programmed with another predicate, which removes links from the logical database and their side-effects on the screen:

```
delete(error::_,i(scene_2,_,_)),
delete(dialogue::_,i(scene_2,_,_)),
delete(message(_)::_,i(dialogue,_,_)),
delete(click::_,i(scene_2,_,_)).
```

The first parameter gives us an example of path names but where transformation matrices (on the right of the infix operator '::') were not instantiated.

A drawing design

Let us see now the main goals of an application program dealing with the drawing design of a flat:

- creation of a new plant or loading a saved one;
- dynamic control over the type and number of rooms in a flat;
- creation (or deletion) of specific rooms and/or of their associated doors and windows;
- storage of any plant configuration;
- validation of a configuration according to some set of rules and at any stage of the design;
- zooming on and off to make the user interaction easier.

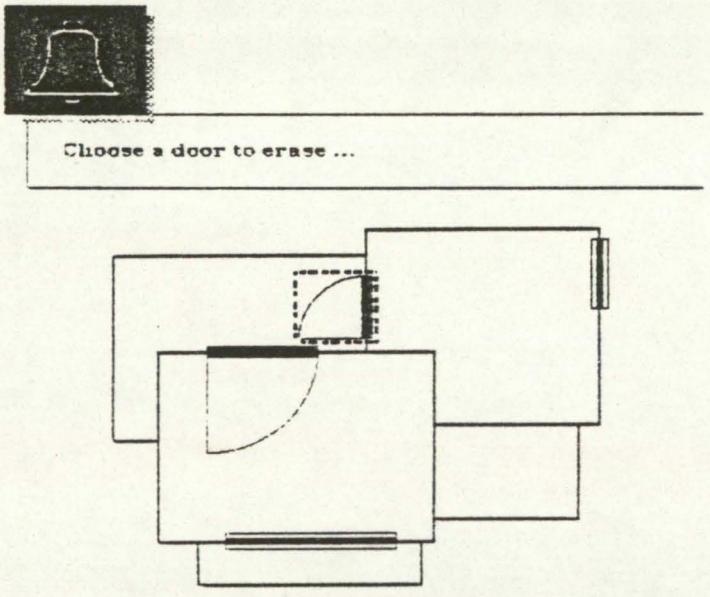

Fig. 3 — Screen layout example.

A common technique for the communication with the end-user, which is found in most interactive graphical applications, is through menus. We implemented a general menu generator where the well-known Choice logical input device is supposed to be used to

produce pop-up menus. Any item of a menu will correspond to a state of the dialogue. This dialogue is partially specified by giving the set of all allowable states in the next step:

```
begin ---> [<state_1>,<state_2>,...,<state_n>].
<state_1> ---> [<state_1_1>,<state_1_2>,...,<state_1_j>].
    ...
<state_n> ---> [<state_n_1>,<state_n_2>,...,<state_n_j>].
    ...
exit(<List_of_terminal_states>).
```

In the particular case of the flat design, that specification is given by the rules:

```
begin ---> [new_,restore_,save_].
create ---> [create_room,create_door,create_window].
create_room ---> [].
create_door ---> [file_,create,erase,structure,
                  zoom_on,zoom_off,validate,quit].
create_window ---> [file_,create,erase,structure,
                    zoom_on,zoom_off,validate,quit].
erase ---> [erase_room,erase_door,erase_window].
erase_room ---> [file_,create,erase,structure,
                 zoom_on,zoom_off,validate,quit].
erase_door ---> [file_,create,erase,structure,
                 zoom_on,zoom_off,validate,quit].
erase_window ---> [file_,create,erase,structure,
                   zoom_on,zoom_off,validate,quit].
validate ---> [file_,create,erase,structure,
               zoom_on,zoom_off,validate,quit].
structure ---> [add_str,del_str].
add_str ---> [file_,create,erase,structure,
              zoom_on,zoom_off,validate,quit].
del_str ---> [file_,create,erase,structure,
              zoom_on,zoom_off,validate,quit].
zoom_on ---> [file_,create,erase,structure,
              zoom_on,zoom_off,validate,quit].
zoom_off ---> [file_,create,erase,structure,
               zoom_on,zoom_off,validate,quit].
file_ ---> [new_,restore_,save_].
new_ ---> [file_,create,erase,structure,
           zoom_on,zoom_off,validate,quit].
save_ ---> [yes_save,no_save,create_door_save].
yes_save ---> [file_,create,erase,structure,
               zoom_on,zoom_off,validate,quit].
no_save ---> [file_,create,erase,structure,
              zoom_on,zoom_off,validate,quit].
create_door_save ---> [file_,create,erase,structure,
                       zoom_on,zoom_off,validate,quit].
restore_ ---> [file_,create,erase,structure,
               zoom_on,zoom_off,validate,quit].
quit ---> [yes,no].
no ---> [file_,create,erase,structure,
         zoom_on,zoom_off,validate,quit].
exit([yes]).
```

Items of a specific menu are obtained from the corresponding names of the states, according to existing translation rules. Designations like "zoom_on" or "file_", for instance, must be conveniently replaced by strings of characters like `"zoom ON"` or `"File ..."`, respectively:

```
zoom_on  becomes  "zoom ON".
file_    becomes "File ...".
```

States not considered by rules of this kind will be seen exactly as they appear in the dialogue specification. This is achieved by converting atoms (ie, state names) into strings due to the extra-logical facilities found in the implementation language.

Nevertheless, it might happen not all the states enclosed by one pair of square brackets (as it was stated above) be possible at one time. Instead of creating more states by subdivision, we restrict the set of admissible items at run-time by the interpretation of conditions having the form

```
<state> <=> <Condition>.
```

where `<Condition>` stands for a Prolog goal. For instance, if an option `DELETE OBJECT` is shown, then the user will know that at least one instance of that type must exist. Otherwise the program would not show that item. Therefore, the interpretation of the following facts

```
create_door   <=> defined_room(_).
create_window <=> defined_room(_).
```

is that menu options corresponding to *create_door* or *create_window* must be eliminated from selection if there are no rooms created in advance. By instantiating the variable *R*, the goal *defined_room(R)* returns the designation of some room already outlined by the user. This predicate fails if there is an empty configuration at that moment.

Another example is the existence of mutually exclusive alternatives, like *zoom_on* and *zoom_off*, which cannot be shown on the screen simultaneously.

Entered the state item, the user may proceed the dialogue by using other graphical input techniques. This means that basic interactions provided through Locator, String, Valuator and Pick input devices are also available. Only request mode is available at the moment.

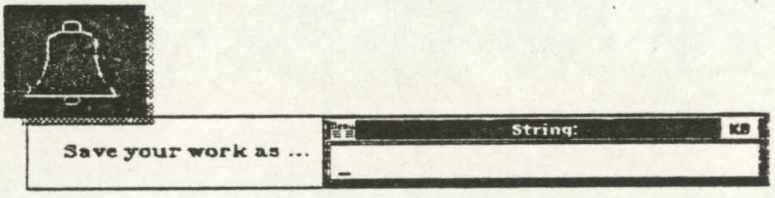

Fig. 4 — Saving descriptions into a file.

Dealing with interaction

The underlying model of interaction specification was based on the way graphical input is managed in current graphical standards [ISO85,86,87] and was designed from an extension of the same main ideas of the Dialogue Cells concept introduced by Borufka, *et al* [BOR82].

A Logical Dialogue Building Block (or LDBB for short) is thus a logic term having separate rules for dealing with different aspects of the dialogue decomposition. Figure 5 shows the main parts of an LDBB, where

- prompt rules are called at the time the corresponding LDBB starts to be solved;
- symbol rules give the syntax of sub-dialogues, which will be specified in terms of the basic logical input at the very low level;
- value rules are responsible for the translation of complex data structures;
- echo rules state the way graphical output is produced in response to a specific input at that LDBB level;
- help rules include messages to a different operating system process depending on the context of the execution and not disturbing the current screen layout.

The achieved description using this technique is very modular, being a good tool for incremental programming. On the other hand, although assuming the existence of an adequate meta-interpreter, this description was planned to have the clear syntax of logical terms.

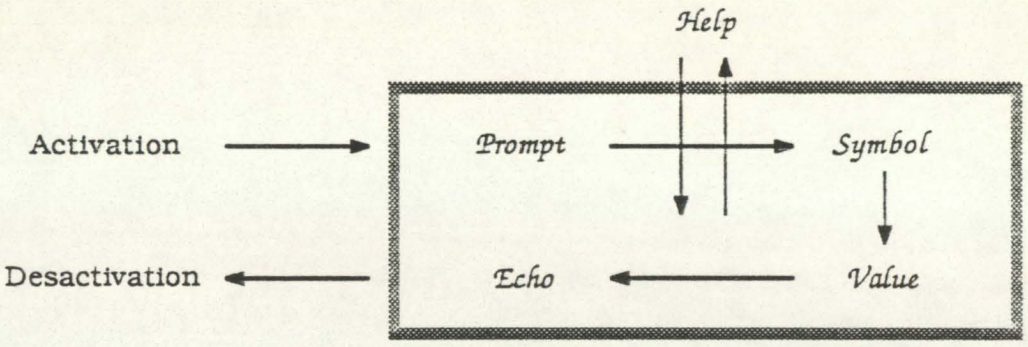

Fig. 5 — Main parts of an LDBB.

The dialogue control programming

Going on with the application used for illustration in this paper, the programmer might create several LDBB's levels depending on the current state. In what follows, *action(<State>)* is a term standing for an LDBB. Therefore the behaviour implied by each dialogue state is detailed in a general structured form, given by symbol rules

```
action(<State>) :- action_(<State(<Parameters>)>).
action_(<State(<Parameters>)>) :- <interaction_goals>.
```

For illustration purposes, here follows the way how the zoom activation capability was programmed:

```
action(zoom_on) :- action_(zoom_on(_,_)).

action_(zoom_on((P,Area),(P0,P1))) :-
            get_point(scene(scene_1),prompt(3),Area,P,P0),
            get_point(scene(scene_1),prompt(5),Area,P0,P1).

prompt_rl(action_(zoom_on((P,_),_)),_) :-
     scene(scene_1,wv(P,_,_,_)).
value_rl(action_(zoom_on(_,(X0:Y0,X1:Y1))),_) :-
     '#TEST'(not equal_points([[X0:Y0,X1:Y1]),
                  "Invalid zoom area specification !..."),
     min(X0,X1,Xw0),max(X0,X1,Xw1),
     min(Y0,Y1,Yw0),max(Y0,Y1,Yw1),
     retract(scene(scene_1,wv(PW0,PW1,PV0,PV1))),
     asserta(scene(scene_1,wv(Xw0:Yw0,Xw1:Yw1,PV0,PV1))),
     retract(application(zoom(disabled))),
     assert(application(zoom(enabled(wv(PW0,PW1,PV0,PV1))))),!.
echo_rl(action_(zoom_on(_,_)),_) :-
     display_hierarchy(scene_1).
```

The initialization of a reference point is done in the prompt rule. Changing the normalization transformation occurs when the value rule is evaluated. Predicate *'#TEST'(Goal,Warning)* will succeed if the validation goal *Goal* also succeeds. Otherwise it will fail and send a warning to the display surface, storing a new fact in the logical data base as well:

```
'#TEST'(Goal,_) :- call(Goal),!.
'#TEST'(_,Error_Message) :-
            assert(application(error(Error_Message))),!,fail.
```

Finally, the simple echo rule written above is the responsible for the screen up-to-date.

Two other LDBB's were also referred: *action* and *get_point*. The later is at a lower level and is indispensable in order to allow the end-user to define the picture area for enlargement. The former has a more general nature in this application example.

We have already seen a symbol rule for *action*. The remaining non-empty parts of this LDBB is described bellow

```
prompt_rl(action(State),_) :-
      prompt(State,Prompt_Message),
      create_link(prompt,scene_2,_,_),
      create_link(dialogue,scene_2,_,_),
      create_link(message(Prompt_Message),dialogue,_,_),!.
echo_rl(action(_),_) :-
      unlink_subgraph(dialogue),
      succeed(delete(prompt::_,i(scene_2,_,_)),!.
```

If there is a fact for *prompt* in the logical database and matching the current dialogue state, a prompt message (given by the second argument) will be displayed on the screen.

The purpose of predicate *unlink_subgraph(FName)*, which always succeeds, is to erase all links concerning family *FName* from the logical database. Predicate *succeed(Goal)* has the form

```
succeed(Goal) :- call(Goal),!.
succeed(_).
```

This means that the prompt object link, if any, will be conveniently deleted (see figure 1).

There is another symbol rule for *action* which, according to the execution strategy of Prolog, must be inserted after the other rules. That is the reason why the argument value does not care in this case:

```
action(_) :- display_error.

display_error :-
      just_once((repeat,locator(_,_))).

prompt_rl(display_error,_) :-
      retract(application(error(Error_Message))),
      succeed(delete(message(_)::_,i(dialogue,_,_))),
      succeed(delete(prompt::_,i(dialogue,_,_))),
      create_link(error,scene_2,_,_),
      create_link(message(Error_Message),dialogue,_,_),
      create_link(click,scene_2,_,_),!.
echo_rl(display_error,_) :-
      unlink_subgraph(dialogue),
      delete(click::_,i(scene_2,_,_)),
      delete(error::_,i(scene_2,_,_)),!.
```

One could argue the fact of *display_error* being also an LDBB. However, in spite of its name, *display_error* appeals to an end-user reaction as it can easily be seen above. This LDBB is always executed when an error occurs, so that the fact *application(error(Error_Message))* must be retracted.

Operations on direct graphics objects

The interactive design of a flat is somewhat simplified by making use of Manhattan polygons (whose sides are parallel to two orthogonal axes) for modelling the shape of rooms. Nevertheless this is the case of the great majority of flats in the real world.

The application program allows interactive drawing of a floorplant, based on subtraction operations over such polygons. The user defines a room by means of the manipulation of a rubberband rectangle. The final outline of the room will be obtained as the result of subtracting all existent configuration from the rectangle specified by the user.

The module dealing with polygons was fully implemented in Prolog having

```
full_subtraction(Fig_*Ext_ - List_of_Figures,Fig*Ext)
```

as the top goal for the subtraction and where *F*E* is a description of a polygon: *F* is a complete list of vertices given in a clockwise order, *E* is its rectangular extent given in the form *extent(XMin:YMin,XMax:YMax)*. *List_of_Figures* is a set of polygons represented in the same way except that vertices must be in a counter-clockwise order. The second

parameter will return the polygon obtained after subtraction and whose vertices are in a clockwise order.

From the user point of view, if the current state of the design is like figure 6, the user might define another room (the hall in this case) by manipulating a rubberband rectangle (shown in figure 7). The result is a new flat configuration, whose new room is given after an automatic subtraction of polygons (figure 8).

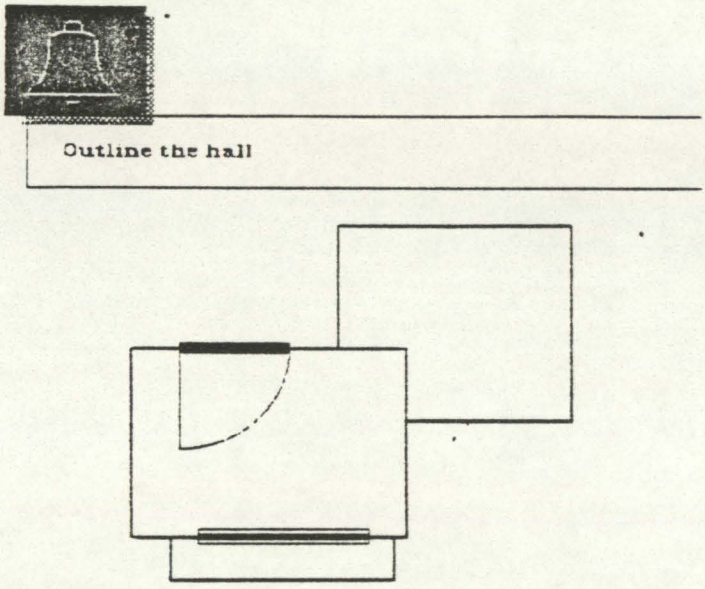

Fig. 6 — Room definition process.

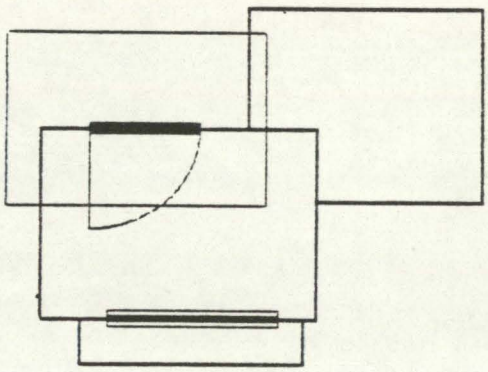

Fig. 7 — Rubberbanding.

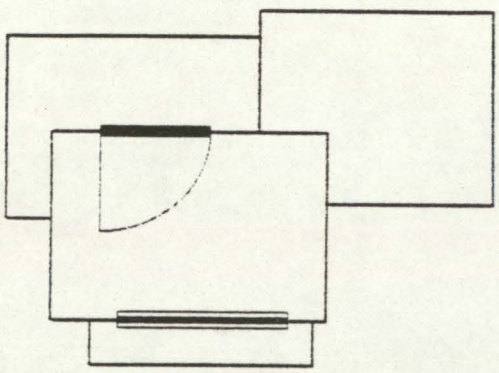

Fig. 8 — Resulting configuration.

Ambiguity caused by overlapping

As we have said before, pop-up menus are automatically generated by the dialogue specification. But this does not forbid the application programmer to use menus at a lower level LDBB. In our application program we can find two typical examples of that.

The first one occurs when one wants to define an object such as a door or a window. It will belong to the wall picked by the user. But whenever two walls of different rooms full or partially overlap the user might want to know which room was really selected by picking one of its walls and getting information from it. If he gets a unexpected answer, he is allowed to try again by selecting the appropriate item from the menu shown in figure 9. And another pick at the same location will not return the same object, unless it overlaps none. The reason of this is that after an object has been picked its relative priority for display is automatically set to the lowest level.

Another occurrence of menus concerns the validation mechanism applied to any configuration as it is described in the next section.

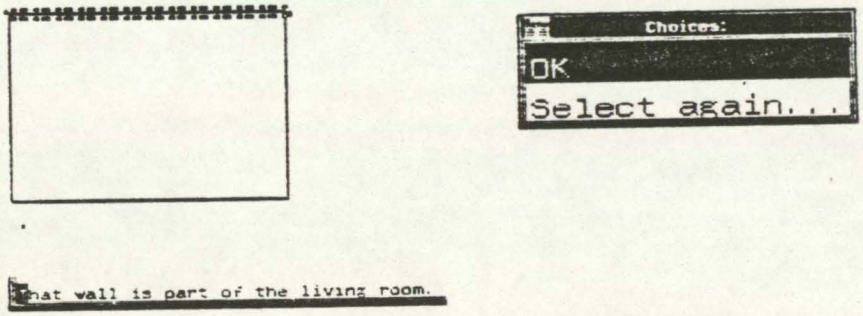

Fig. 9 — The user must confirm the selection.

Explanatory mechanisms

Neither all resulting configurations in the design process are acceptable. Validation techniques may vary from a simple checking of a set of rules to a powerful expert system. We describe here one solution that can be easily generalized.

Checking rules must be in the form

```
<Rule>:  if <Condition>
         then <Goal>.
```

Every rule has two explanatory facilities: one of them (the *rule_fail_indication* predicate) gives a short message about the most admissible reason of a rule failure, while the other one (the *rule_fail_explanation* predicate) is a more detailed description in case of failure.

For example, if we cannot imagine a bathroom without at least one door and one window, the corresponding rule could be expressed in the form

```
bath_door_window: if defined_room(bathroom) and
                     part_of(door(_)::_,room(bathroom,_)::_) and
                     part_of(window(_)::_,room(bathroom,_)::_)
                  then bathroom_ok.
```

The rule above can be used by the programmer in order to provide suitable error messages:

```
rule_fail_indication(bath_door_window,
                     "Window or Door missing in the bathroom") :-
            defined_room(bathroom),!.
```

```
rule_fail_explanation(bath_door_window) :-
            not defined_room(bathroom),
            write('Bathroom not yet defined !'),
            nl,fail.
rule_fail_explanation(bath_door_window) :-
            not part_of(door(_)::_,room(bathroom,_)::_),
            write('There are no doors in this bathroom !'),
            nl,fail.
rule_fail_explanation(bath_door_window) :-
            not part_of(window(_)::_,room(bathroom,_)::_),
            write('There are no windows in this bathroom !'),
            nl,fail.
rule_fail_explanation(bath_door_window) :- nl.
```

Validation is done by checking all the existent rules. When the condition of a rule is not verified, the short message from *rule_fail_indication* is displayed and a pop-up menu (figure 10) gives the user one of the following possibilities:

- to know more about what is wrong;
- to ignore the specific rule and resume the task of validation;
- to quit that task and return to the drawing work.

Some basic validations are done automatically while the floorplant is outlined: windows and doors cannot intersect each other, a door must be opened without intersect any wall, a room cannot be inside other room, etc..

In the real world there are regulations to be followed in designing a flat. We have inserted some of those legal rules in the logical data base according to portuguese governmental regulations. One of these rules, for instance, states that the maximum width admissible for a balcony is 1.80 meters (71st article in [RGE79]).

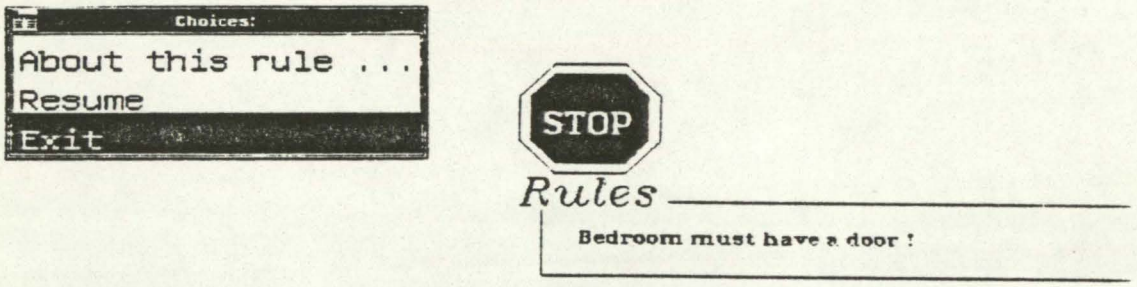

Fig. 10 — Invalid configuration.

Conclusions

There are recognized advantages in the approach just introduced. In fact, clausal form of logic programming preserves modularity, while a declarative style is useful to produce executable specifications. Unification is a very powerful mechanism, allowing pattern matching automatically and eliminating an explicit distinction between input and output parameters in a procedure call. Backtrack is general enough to be applied to any search process, restoring the previous environment on failure independently from the application control (depth-first strategy). Meta-interpretation is easy to implement and all features are inherited from the top level interpreter of Prolog.

As it can be seen, data structures and specification of rules are truly flexible enough to adapt to changes of almost any kind. In general, the presentation order of clauses does not care. Note that this is unusual even in what concerns data structures. For instance, the complex mechanism for structures in PHIGS [ISO86] strongly depends on the order of the programmed elements, which is not a necessity of the application itself. This is not natural for the programmer at all.

In our logic approach, Pick is not at the low level of the graphical system data structures. On designing and implementing the entities model, we tried to build a system such that a declarative style of programming might be applied, on the one hand, and where the application model would be easier to map, on the other. As it was stated before, graphical objects are instances of families and picking any available object returns its name, which

can be directly recognized by the application. A complete description of our modelling system and details on dialogue control can be found in [PRÓ88].

Our current implementation was built on top of GKS level 2b. This has the disadvantage of dealing with a one-level segmentation mechanism without editing capabilities. The obvious result is a lost of efficiency in some basic operations (such as *display_hierarchy*). However we think that an acceptable compromise has been achieved, since a quite good GKS implementation was available.

The research and development work at ULN is being supported by the ESPRIT project p973 (Advanced Logic Programming Environments [PRE87]).

References

[BOR82] Borufka, H.G.; Kuhlmann, H.W.; ten Hagen, P.J.
Dialogue Cells: a Method for defining Interactions, *IEEE Computer Graphics & Applications*, 25 pp 25-33 (July 1982)

[ERO87] Ero, J.; van Liere, R.
User Interface Management Systems, *Eurographics'87 Tutorial*, Amsterdam (1987)

[ENC82] Encarnação, J.; Hanusa, H.; Strasser, W.
Tools and Techniques for the Description, Implementation, and Monitoring of Interactive Man-Machine Dialogues, in *IEEE Proceedings of the International Zurich Seminar on Man-Machine Communications* (1982)

[ISO85] International Organization for Standardization
Information processing systems — Computer graphics — Graphical Kernel System (GKS) functional description, ISO IS 7942 (July 1985)

[ISO86] International Organization for Standardization
Information processing systems — Computer graphics — Programmer's Hierarchical Interactive System (PHIGS) Part 1 — functional description, ISO dp9592/1 (October 1986)

[ISO87] International Organization for Standardization
Information processing systems — Computer graphics — Graphical Kernel System for Three Dimensions (GKS-3D) Functional Description, ISO DIS 8805 (1987)

[PFA85] Pfaff, G.E. (editor)
User Interface Management Systems, Springer-Verlag (1985)

[PRE87] Preston, N.; Próspero, M.J.; Gandilhon, T.
Prolog and Graphics — Specification, Deliverable for WP3.1, ESPRIT project ALPES-p973 (September 1987)

[PRÓ86] Próspero, M.J.; Messina, L.A.
Towards the Construction of Graphical Interfaces on Basis of Geometric Models, in *Proceedings of the Eurographics'86*, pp 173-183, North-Holland (August 1986)

[PRÓ88] Próspero, M.J.
Estilo declarativo na Programação Gráfica Interactiva: análise e avaliação sobre sistemas em Prolog, Phd Dissertation — UNL (August 1988)

[RGE79] *Regulamento Geral das Edificações Urbanas*, Imprensa Nacional — Lisboa (1979)