INTERFACE DESIGN OF A VISUAL PROGRAMMING LANGUAGE:
EVALUATING RUNNABLE SPECIFICATIONS

Gabriele JANKE and Klaus KÖHNERT

Project ABSYNT
Department of Computer Science
University of Oldenburg

We give a description of how the development and psychological examination of an interface can be done by designing and analyzing state-transition-diagrams and equivalent Prolog-programs. The advantage of such an user-interface-specification is the possibility of examining properties before the implementation in the final implementation language. The results of an examination may be changes and further examination until the best version of the interface is found. Only this last version has to be implemented in the final implementation language. So the suggested procedure of specification is a time saving one.

This report is about a special aspect of the research done within the ABSYNT project. In our project we are doing basic research on the planning behavior of novice programmers. Our findings may prospectively by applied within the field of intelligent computer-aided instruction (ICAI). Our group developed ABSYNT (ABstract SYNTAX), a purely functional visual language, to the needs of our project. It is furnished with an user-friendly interface that provides maximum support.

1. ABSYNT

ABSYNT is a visual and purely functional programming language. It was developed in our group and emerged from ideas presented in (1) and in (2). It has been modified from illustration diagrams to a runnable programming language with directly manipulatable visual objects((3), (4), (5)). ABSYNT is especially tailored to the needs of our project. Its design is motivated by

- empirical and formal reasons based on the original version of Bauer, Goos and various alternatives.
- a Prolog specification of its semantics which determines all the necessary details visible on the screen ((6), (7), (8)).

+ This research is sponsored by the DFG in the SPP Wissenspsychologie under Grant No. Mo 292/3 – 2.
FIGURE 1
Example of the editor's user-interface.
Figure 1 gives an example of the editor's user-interface. The user has just created a syntactically correct function consisting of a head-tree and a body-tree in the upper frame. The lower frame is still empty. The trees are composed by the user with the manipulatable objects node and connection. They are created, moved, positioned, and deleted by a mouse-controlled cursor. Available nodes (provided by ABSYNT or self-defined ones) are visualized in NodeWindows at the margins of the screen, from where copies are created and moved into a frame window.

2. THE REPRESENTATIONAL FORMALISM

2.1. The state transition diagram

The editor user-interface was formalized as a state-transition-diagram. Our formalism is similar to ANTs (Augmented Transition Networks, see (9)) and GTNs (Generalized Transition Networks, see (10)); but contains some important differences. The state transition diagram describes how one state of the interface will be transformed into another state by user actions. Figure 2 gives further explanation: a transition from state-1 to state-2 is done, when the user performs action-1. State-2 is transformed to state-3, if the user executes action-2, and to state-4 with action-3.

![State Transition Diagram]

FIGURE 2
A state-transition-diagram.

Next we had to determine which of the characteristics of a state must be made explicit in the rectangle representing that state. The structure of a state description depends on the device to be formalized. The sample version of our editor to be presented here has four relevant features that characterize a state. Figure 3 shows how they are represented. The four fields are interpreted as
exis.object: existing object is one of "at least one node with position P1" (N(P1)), "at least one connection with endpoints P1 and P2" (C(P1,P2)) and "no node or connection or any number of nodes and connections" (any Obj).

cursorpos: current cursor position is one of 9 relevant possibilities like "at a node with position P1" (N(P1)) or "at any position of the screen" (anyP).

flash.object: node currently moved with the cursor (N) or connection with fixed endpoint P1 currently moved with the cursor (C(P1)).

menu: actually visible pop-up menu one of 8 possible menus like the menu appearing at nodes (N)

<table>
<thead>
<tr>
<th>ex.obj</th>
<th>cursorpos</th>
<th>flash.obj</th>
<th>menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>N(P1)</td>
<td>anyP</td>
<td>anyP</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 3**
Features of a state

**FIGURE 4**
A node

Figure 4 shows a state with at least one node visible on the screen at position P1. The cursor position is not determined. There is no object flashing at the cursor and there is no pop-up menu visible.

The next step was to formalize all sequences of states and actions possible within the editor by connecting all possible states with all transforming user actions. The result was a large and complex network. To reduce complexity, make the network easier to survey, and to represent the repeated run of state-action-chain we structured the network into subnets as described in (9). All state-action-chains represented within this formalism will occur only once, although they may be used frequently. Entering a subnet is indicated by an enter-arrow. The return from a subnet is indicated by a leave-arrow.

Figure 5 illustrates this mechanism: all actions and resulting states that are possible from state-1 are described in subnet-n (represented as an enter-arrow to the ellipse). The first action in subnet-n is action-1, followed by state-2, which in turn may be transformed to state-3 by action-2. Reaching state-3 successfully finishes the processing of subnet-n and the leave-arrow leads back to the calling network, where may
be continued with action-3. With other words, subnet-n in the topmost line may be replaced by the actions and states in the middle-line. This replacement is illustrated in the bottommost line.

FIGURE 5
Networks and subnets

Figure 6 gives an example. The top-level network of our diagram is called Edit (figure 6a). All possible actions and resulting states are described in four subnets. Two of them can only be reached if a condition marked at the top of enter-arrow is satisfied. For example, there must be a node (node!) to enable the user to do actions described in subnet Actions NMenu(P1). Figure 6b shows subnet Actions NMenu(P1) calling another subnet, CreateNMenu(P1), represented in figure 5c. The latter is the subnet with the deepest nesting in this example. It shows how a pop-up menu at a node is created. The cursor may be moved to the node. When now a mouse button is clicked, the node-specific pop-up menu appears at the node. But clicking a mouse button is not obliged, all other actions described in the Edit network are possible (indicated by the enter-arrow to Edit). If after a mouse click the node's pop-up menu appeared, the actions that are possible from now on are described at the place from where the subnet was called (indicated by the leave-arrow). In our example it is in the subnet represented in figure 6b. The possible actions are: select no item (no-sel), select item "move node" (sel(mN)), select item "delete node" (sel(dN)), and select item "make

Example Diagram:

```
[Diagram showing network and subnets: state-1 to subnet-n to state-4 via action-3, subnet-n to state-2 to state-3 via action-2, and state-1 to state-2 to state-3 via action-3.

Calling network:
- State-1 to subnet-n to state-4 via action-3.
- Subnet-n to state-2 to state-3 via action-2.
- State-1 to state-2 to state-3 via action-3.

Possible actions in subnet-n include:
- Select no item (no-sel).
- Select item "move node" (sel(mN)).
- Select item "delete node" (sel(dN)).
- Select item "make

FIGURE 5
Networks and subnets]
```
FIGURE 6
An example for a network.
connection" (sel(nc)). The possible actions after selecting the items "move node" and "make connection" are represented in the subnets Pos N and PosC(P1). The leave arrows at the end of this subnet indicate again, that the following actions are described in the network from where the subnet is entered. In this case the enter-arrows back to the Edit network illustrate the possibility of a repeated run through the network.

2.2. The translation to Prolog

To have a runnable version of the just described diagram we translated it into a Prolog program. Figure 7 illustrates how the descriptions of states are translated into Prolog structures. Each state gets a name and a corresponding structure. All structures consist of the four relevant aspects object, cursor, flashing and menu, that themselves may be structured, like the aspect object which is structured by node(p1).

Actions are build in a similar way: They also consist of a name and a describing structure, whose aspects are in turn substructured (figure 9).

<table>
<thead>
<tr>
<th>N(P1)</th>
<th>any</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

is translated to a Prolog term like this:

(statel(name(node p1),
    structure(object(node(p1)), cursor(any), flashing(no), menu(no)))

**FIGURE 7**

Prolog structures.

The two steps of figure 8 show how the translation of transitions to Prolog facts is done. The slots for the three arguments of the functor transition are preState, postState, and userAction. preState is the initial state, postState the final state, and userAction the user action that transforms preState to postState.

Figure 9 gives an example for the transition from a state with at least one node at position P1 and any cursor position to a state where the cursor is located at the node with position P1 by moving the cursor to this node.
is translated to a Prolog fact like this:

```
transition(preState, postState, userAction).
```

```
transition(state-1(name(...), structure(...)),
          state-2(name(...), structure(...)),
          action(name(...), structure(...))).
```

**FIGURE 8**
Prolog facts.

is translated to a Prolog fact like this:

```
transition(state1(name(node_p1),
                 structure(object(node(p1)), cursor(any),
                             flashing(no), menu(no))),
           state2(name(node_p1),
                 structure(object(node(p1)), cursor(node(p1)),
                             flashing(no), menu(no))),
           action(name(move_node_p1),
                  structure(move(node(p1))))).
```

**FIGURE 9**
An example for the transition.
The collection of all transitions represents the network. The Prolog rules 'path' and 'steps' provide two different ways to find state-action chains leading from one state to another:

1. The rule 'path' automatically gives all desired state-action-chains that are possible within the network. If the user's task is to transform state-1 into state-n, the query to Prolog will be

\[
\text{path(state-1, state-n, StateActionList)}, \text{ where state-1 and state-n are the state's names.}
\]

It returns StateActionLists for each possible solution, if any, like

\[
\text{StateActionList = } ( (\text{action-1, state-2), (action-2, state-3),} \\
\ldots, (\text{action-<n-1>, state-n}) ).
\]

The elements of the returned StateActionList are lists with two elements. The first element of each list is the name of the user action, the second the name of the state. The shortest list represents the way an ideal expert user will go from state-1 to state-n, if a way exists. For example to arrive at a StateActionList describing the actions an expert user performs to move a node from its old position(p1) to a new position (p2), we set state-1 to node_p1 and state 2 to node_p2, the names for the initial and the final state. The query and the answer are shown in figure 10. For a more readable arrangement of states and actions, they are printed out in columns. The left column gives the user actions, the right the resulting states. After moving the cursor to the node at position p1 the cursor is at that node, clicking any mouse button will cause the node's pop-up menu to appear at the node, after selecting the menu-item "move node" the node will flash at the cursor, still at position p1. When the cursor is moved to the free position p2 (no overlapping with another node or connection), the flashing node is at the new position, where a mouse click fixes it.

:path(node_p1, node_p2, StateActionList)
StateActionList = [ 
    (move_node_p1, cursor_node_p1), 
    (click, menu_node_p1), 
    (select_move_node, flash_node_free_p1), 
    (move_free_p2, flash_node_free_p2), 
    (click, node_p2) 
]

FIGURE 10
Prolog rule 'path'.
2. The rule 'steps' offers the possibility to get through the network stepwise. Given any state the program shows a dialog window, where the name of the actual state is indicated and a scroll menu with all possible actions is opened. Figure 11 shows a snapshot of a dialog that started from a state with at least one node which is located at position p1. The dialog has just arrived at a state with a pop-up menu at the node. After the selection of one of the items in the scroll menu another dialog window will appear indicating the following state and all actions now available. Selecting the button END will finish the dialog and return the produced State ActionList.

: steps(node_p1, StateActionList)

![Dialog window](image)

FIGURE 11
Prolog rule 'steps'.

3. EVALUATION

The final step to take is the psychological evaluation. Psychological aspects of the quality of an user interface are ease of learning and ease of use. These features must become apparent as features of the state-transition-diagram or the Prolog program. Measurable facts in the diagram are for example number of states and transitions, number of applications of the same subnet in different networks, similar sequence of user actions in different networks. The Prolog program represents measurable facts as the number of transitions, the number of different structures for states, the length of StateActionLists, similar sequences of user actions in this StateActionLists for different queries (corresponding to tasks).
FIGURE 12
Similarity of subnets

Roughly speaking one can say, the number of states and transitions is an indicator for the complexity of a system. A high number of applications of the same subnet in different networks indicates that the complexity is reduced. The similarity of subnets may be interpreted as an indicator for consistency. Figure 12 gives a simple example for similarity of subnets. The actions to create a pop-up menu at a node and at a connection are very similar: move the cursor to the object (node or connection) and click any mouse button.

For a correct assessment of the quality of a system one has to differentiate at least two user groups: expert users and beginners. For expert users a system is good, if it is easy to use, beginners need a system that is easy to learn. Another important factor for beginners is how the system reacts to errors.

Let us consider for example how consistency may be tested and achieved by Prolog. The expert user's actions to perform the following four tasks should be consistent:

1. Create Node
2. Move Node
3. Create Connection
4. Move (one endpoint of a) Connection

Figure 13 gives the actions an expert user would perform in the first version of our system. With exception of the task 'Create Node' the sequence of actions is very similar:
FIGURE 13
Performance in the first version of the system.

FIGURE 14
Performance in another possible version of the system.
- move the cursor to a task-specific place
- click any mouse-button
- select a task-specific item from a pop-up menu that just appeared
- move the cursor with the flashing node or connection to another task specific place
- click any mouse button.

When creating a new node in the original version one of these steps is skipped: after the first mouse click no pop-up menu appears and no item has to be selected, the node appears flashing at the cursor just after clicking. To make the system more consistent and thus easier to learn, this feature is changed in the next version. This version makes the user perform one more action to create a node (select the item from an appearing pop-up menu containing only this single item). Moreover, it is possible that the user generalizes the actions to perform all four tasks. This kind of consistency may be provided by a carefully designed program. Also the possible mental process may be modelled by Prolog. The Prolog program should be able to modify itself and result in generalized facts for transitions, using variables concerning the objects node and connection and the special places. This program must be supported by psychological criteria, that determine which tasks and which actions are psychologically similar.

To demonstrate the difficulties let us consider another possible version of our ABSYNT system:

To avoid pop-up menus, different buttons and combinations of buttons could determine the reactions of the system. The actions an expert user would do to perform the four tasks are illustrated in figure 14. The number of actions is reduced for each step. The system is consistent again, because the sequence of similar actions is still the same for all tasks. But there is an important disadvantage: There is no semantic relation between the action and the task. For example there is no mental connection between the task 'move node' and the action 'click right mouse button'. This connection would have to be learned. Moreover it is rather difficult to make consistent the relations between mouse button and task if the number of different actions is too large. In the example of figure 14 the right mouse button is used for nodes if it is clicked at a node or a NodeWindow, and it is used for connections if it is clicked at a connection.

An assessment of the systems reactions to errors (or error handling) is achieved by trying different ways to perform a task and observe how the system's reaction demonstrates errors and possible repairs. The Prolog rule 'steps' is tailored for this kind of examination.
4. CONCLUSIONS

We introduced a specification that allows the description and examination of several versions of an user interface before its implementation in the final programming language. It produces the expert user's actions for all tasks, the system's reactions, and gives information about erroneous actions and repair-possibilities.

More difficult but also possible are psychological evaluations by Prolog. For this demand features like ease of learning, consistency, similarity of states and actions, and mental connections between tasks and actions have to be expressed by Prolog rules. It is then possible that a Prolog program provides some changes to an interface.

When testing several versions the designer has to decide on the most important criteria (e.g. ease of learning or ease of use), perhaps change some version that best meets the special demands.

5. REFERENCES


Interface Design of a Visual Programming Language 581

