ODE: A Highly Customizable Graphical Object Design Editor

Torsten Leidig

AG Telematik
Department of Informatics
University of Kaiserslautern

12.02.90

Abstract

This article describes the basic concepts of an extensible customizable knowledge-based graphical editor and its adoption to the DOCASE methodology and tool environment. One aspect in this field is the mapping of conceptual models (expressed in a specific language) to their graphical representations. This also has impacts to the semantic of the user actions in a graphical editor tool. The ability to extend and customize the editor can be used to build specific graphical interfaces to various kinds of tools in the software development process. Major aspects of ODE are semantics-directed editing besides normal syntax-directed editing, support of abstraction mechanisms, multiple modeless views to attack complexity, semantic analization and animation. The result is an highly customizable graphical editor construction set that matches requirements of applications in many domains of system design.

1 Starting with DOCASE as target system

The DOCASE [9] model is object-oriented. It provides a design language called DODL [10], which is thought to be applicated to different areas like office automation, CIM or distributed multimedia authoring and learning environments. (distributed applications engineering). This implies to be customizible in case of the language and in case of the tools used in DOCASE environment too. More design principles of DOCASE are: explicit representation of semantics and the integration of configuration and computation aspects.

DODL comes with a generic class hierarchy that serves as semantic model convenient for distributed processing and is known to all DOCASE tools. The hierarchy is divided in: a structural part, a computational part and a system part. The following list gives a short overview of the approaches in DODL:

- the language is strongly typed to match requirements of early error detection
- it has a structural hierarchy; systems recursively consists of subsystems in an arbitrary depth
- there are typed relations between objects enabling explicitly description of communication patterns, cooperation, etc.
- the language allows description of both static parts (configured elements) and dynamic parts (generated elements)
- the languages allows description of both classes (types) and instances (objects)
- a graphical representation of language constructs is suggested

There is a strong need for graphical tools in DOCASE because of the complexity of system design especially when it is combined with problems of distributed programming. So we don’t want to stress this point. However, the list above implies much requirements or wants for an interactive graphical editor tool.

2 Requirements for a graphical editor

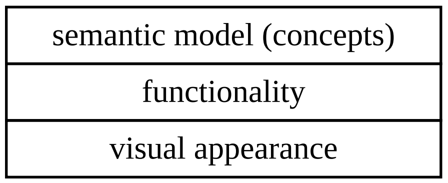
Using DOCASE as an application area has some major impacts on the functionality for the graphical object design editor. So the editor must be able to:

- display, construct and extend the class hierarchy of DODL
- display and modify relationships between objects of various types (also in a type hierarchy)
- handle structural hierarchies in object design (system to subsystem hierarchies)
- show and edit representations of DODL intrinsic data types
- check the semantics of the above constructs to prohibit ’some’ wrong specifications and maintain consistency and completeness
- analyze the specified systems using certain methods and algorithms
- allow animation of the system under development
- interchange data between DOCASE tools

There are many more detailed requirements, which we wont discuss here in this depth.

However, an important implication of nearly all mentioned points is the customizability or extensibility of the editor. Looking on existing software products we can recognize different levels of extensibility, that you can divide into three layers. Lowest layer is the layer of extensiblity through simple changes in visual appearance. This isn’t really an extension to the existing system, so the word customization matches a little bit more. Almost every graphical tool is able to allow this kind of customization. The usual way is: to supply some ’style’ files in which the appearance of the objects used in the editor can be changed. Attributes, which can be changed are for example: color, height and width, font, line width, and so on. A higher degree of extensibility is achieved by being able to add new functions in an otherwise elsewhere more static editor frame. On the third layer a much more flexible system is obtained if the editor is able to incorporate the concepts of the application domain.

Figure 2-1 Layers of extensibility



This point will become more clearly by the following example: Suppose you want create a Petri net editor from a universal graphical editor tool, which is able to display a directed graph and handle some user actions on it (like selecting nodes and edges by clicking on it). First you can create the visual styles of the different nodes (places and transitions) and maybe the style the graph is to be drawn. At this point we are able to draw some fine valid Petri nets. But we are also able to draw things that are never Petri nets (e.g. an edge between two places or two transitions). If your editor allows functional extensions you can add some functions to set marks into a place or to fire a specific transition. But this works only in one view. This topic raises also question of how to do the mapping from user actions (button press, pointer motion) to functions. If your editor is able to present multiple views of overlapping issues of the regarded domain he must have knowledge about the semantics of the used objects. For example, if the editor maintains (and shows) a list of places he must know what objects are places. If he maintains a list of currently fireable transitions he must know the difference between fireable and nonfireable transitions. This is what I mean with conceptual extensibility and customization.

At the last, we can distinguish between dynamic (at run time) and static (at compile time) extensibility. For the developer of a customized editor it can be worthfull to do this interactively, at runtime. This is to be kept in mind.

3 Current state of the art[1]

Let us now consider the technology of graphical (sometimes called ’visual’) editing. Early language environments have had only textual editing provided. The functions were static and they regarded the input only as plain text. Typical functions are cursor positioning, insertion, deletion, cut and paste of text parts. Then there came up editors that were extended by programming them; first there were only keyboard macro facilities; later on there came specialized highlevel languages. This opened the way for more syntax directed editing depending on the language used. A good example of such an editor is Emacs [21], which is programmable in a Lisp dialect.

With the increasing capabilities of graphic workstations came a lot of specialized, one domain graphical editors along. I will not discuss any of them here because they have not the things I want. They sometimes have interesting editing capabilities which are merged in modern graphical environments anyway. To look at the graphical environments is much more interesting.

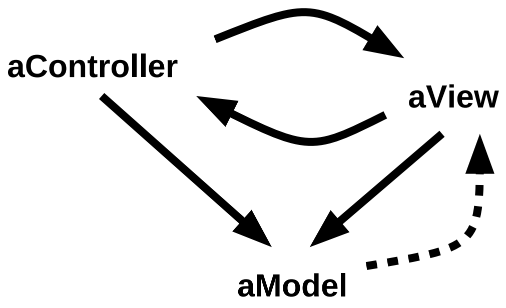
First environment to be mentioned is Smalltalk [17], of course. Smalltalk introduced an highly visual user interface with a new language extending (object-oriented). It has overlapping windows of an arbitrarily large virtual size. User can modeless switch between windows and their functionality or task. Kay’s [14] ’user interface paradigm’ should serve as basis for his ’integrated environment’, which makes the frontiers between the operating system and the application fade away. The Smalltalk system is the father of many current user interface systems, not only used for programming language environments, like the Apple Macintosh. With the multiple windows came up the problem of sharing a common internal data representation. All instances of the i.e. ’Class Browser’ should reflect an internal change. Smalltalks way to do handle this is the Model-View-Controller (MVC) [1, 2] approach. You will never find this term in any of the three bibles [17], whether the functions are implemented and described (dependency field). This leads to the assumption that the mechanisms were not very well understood at the release time. The first system that explicitly knows of ’views’ is the Pecan [19] System. Their idea was to present internal data in several ways and to leave it to the individual user to select that views that are most useful at a particular time. The next section gives a brief introductions how the MVC technique functions.

The *mode*l is the data to be represented in a window on the Smalltalk screen. It may be any data structure you can think of. Important is, that the model has no idea about how to display it self. That’s the task of the *view*, which is used as the mechanism that a model uses to display itself. The view knows how to display the components of the model. There usually are many kinds of views in a system. A *controller* is is the mechanism used by interaction with the model displayed by the view. The controller observes the mouse pointer and the

[1] I don’t want bother you with stories from adam and eve, so only the recent past is considered.

mouse buttons activity on the displayed view. Note that the model has no reference to the view or the controller of that view but the view and the controller know about each other.

Figure 3-1 Model-View-Controller relations



The view to controller, controller to view, view to model and controller to model relations are one to one relations. Otherwise a model can have multiple views (dotted line). It is not in sense of good programming to let the model know about all of it views. In other words a simple instance must not know the hole system.

Another approach to handle this is the ’daemon’ approach [4]. Daemons are similar things like controllers, they are ’fired’ at particular events (i.e. a call of a specific method) and will invoke methods of the ’whole’ (world) to maintain it.

One Method used in e.g. KEE is the active value method. Whenever the value of a field changes or is read or otherwise activated then an user supplied activity is started. This activity is evaluated in the context of the object and knows about the whole like the object itself. It is notable that a new kind of programming style is born out of this technique, that is called data driven style. Methods (= functions) are invoked by altering data fields. In KEE active values are used for the same reasons that Smalltalk uses the MVC method. This is to enable the multiple view approach but further to trigger rules of an expert system.

3.1 Visual programming

Constraint-based programming

A new quality of the multiple view approach is achieved with constraint-based Systems. Views are in a constraint relation with the model; basically constraints can hold on any object in the system. One of the most impressive systems that experiments with constraints is the ThingLab [6]. Giving a set of dependencies between values in form of special expressions, a solution is generated which satisfy all of this (if it exists). The number of solutions are small if the number of constraints is sufficient. ThingLab is a constraint-based simulation environment, that allows direct manipulation of graphic objects on the screen using the mouse pointer and the results of the constraint evaluation are drawn onto the screen in return. The evaluations takes a long time (too long for interactive applications) if applications become more complex. So there is a strong evidence for research in algorithms of fast constraint satisfaction. ThingLab is a such kind of visual language system and it incorporates elements of programming by demonstration too (see following sections). Newer visual language environments like ThinkPad [5] and Rehearsal World [7] were heavily influenced by ThingLab.

Programming-by-demonstration

This approach means that the user performs a graphical manipulations on the screen (demonstrates the program) and the system maps this to executable programs. In ThinkPad users demonstrations on data structures via graphical representations of it are mapped to Prolog code. The ’Rehearsal World Theatre’ [7] produces Smalltalk code (but the user does not have to know Smalltalk!).

Form-based programming

Form-based programming environments are similar to spread sheet programms. But they allow the user to arrange cells in any order, size and appearance in so called forms. Cells are constrained, like spreadsheet cells, by expressions or procedures. There exists an evaluation strategy that can be influenced by the user. This approach is essentially a specialized form of constraint programming where the constraint evaluation algorithm is quite simple. An example of a form-based programming environment is Forms [8].

4 What are the ODE concepts

4.1 How to build visual representations from the model?

One of the main tasks in the project is to explore how one can map the various facets of the model into proper visual representatives. There are a lot of things to learn from other graphical user interfaces and graphical tools. Apart from that we can profit from the knowledge of psychology, especially in the field of communication between people and cognitive psychology. Issues of this domain where important and complex enough to present them an own paper. Main goal is, I think, to map parts of the conceptual model in his cognitive form to visual representations in a one to one or one to many relation, without inserting to much stages of transformation. To the field of visual representation belongs also the translation of user actions into functions. Not only data objects of the application domain have to be mapped to graphical representations. Furthermore the activities (operations) on this objects have to be converted into user actions on the graphical representations (e.g. moving and resizing of objects, dragging a slider a.s.o.). In most editors this kind of mapping is solved by programming it out for each special case. The method looked after in the project of ODE is to give a translation scheme for translating a sequence of atomic user activities (like mouse clicking, pointer motion,...) and complex activities (that are constructed of atomic ones) into a specific function.

4.2 Multiple independent views

For ODE the concept of multiple independent views were adopted. One view shows an particular aspect of a part of the system. It shows the objects in particular colors depending on the significance of data in the subject of the view. Other views may display the same object in a different manner according to the viewpoints they offer. Regardless of that, all views show the actual state of the system under development and are to be maintained if anything changes. First approach to achieve that is the use of active value expressions which is similar to the dependency field technique in Smalltalk or the active valua technique in KEE. Further research is required to supply a more abstract (high-level) and therefore more user friendly method of specification of dependencys (constraints). I think of a rule-based and/or functional system that is combined with a constraint evaluator. Maybe there are restrictic-tions in the kind of dependencies to be made to achieve a fast evaluation.

Views also work as a user interface to input and change the system. So in some views might be controllers like editable text fields, toggle buttons or sliders to alter the internal state of that object. ODE provides views in forms of directed graphs, lists and text areas. It is currently expanded to combine any of the known user interface objects of todays graphical user interfaces to views. In order to enable program animation graphical attributes of the view contents like color, borderwidth, background pixmaps and position can be altered.

Currently we have three kinds of views in ODE. First and most important is the GraphView. It is based on an extensible graph editor that were developed at the University of Karlsruhe. Thist view consist of a viewport wich shows an directed graph layouted automatical with several choosable layout techniques or layouted manualy as users desires. Several parameters and style attributes can be altered. The view is so much important because the object-oriented programming paradigm used in DOCASE is heavily based on relations between objects. For instance one of the approaches in DOCASE is to make communications between a number of objects explicit via a special communications relation which stands in a hierarchy of relation types and has a particular semantic (e.g. a 1:n broadcast communication or a n:m communication following some protocols).

GraphViews are subtyped to various GraphViews which allow editing of special subsets of

Figure 4-1 Sample of Views in DOCASE-ODE

relations between special subsets of objects or object types. There is of course a ClassView that shows subclass/superclass relations between the classes of DODL. Other views show and edit communications patterns combined with a has-component relations which expresses the subsystem structure of the system. A sample screen of views are shown in Figure 4-1. Note that the nodes in this example are placed automatical. Each class of nodes is linked to a popup menu were user actions on this node (that represents an object) can take place. The whole view itself can have a popup menu through which general view actions can be started. User can move nodes by ’dragging’ them to a new location if the automatic layout don’t taste to him. GraphViews allow much more manipulations like editing, layouting and so on, but I don’t want to draw out all. The GraphView is implemented with an large graph editor tool called EDGE [22, 23] that offers a lot of functionality in the fields of graph layout.

Second view is the ListView which holds a set of elements of same or different type. Operations on it were selection, insertion or deletion of elements. Third view is an ordinary text view that allows text editing. The three view kinds are to be unterstood as generalized head-classes to be specialized as needed in the application environment. This is done by the DOCASE-ODE customization. The views can be placed into a window frame that consists of titlebar, pulldown menus and command menus that are arranged in a command panel.

The mechanism used to maintain consistency between the multiple views it very similar to the ’active value’ described above. Upon this there exist special monitoring object classes. Instances of them can simple be attached to objects that should be observed. This technique allows simple and fast evaluation but has sometimes lacks in comfort of the specification of constraints, because its more procedural as declarative..

4.3 Extensibility by multi-paradigm language

The second approach of ODE is the extensibility achieved by a simple but mighty built-in language interpreter. The language I choose for ODE is a lisp-like functional language augmented by an object-oriented data structure and method call mechanism. The object-oriented extension is not to be compared with that in Flavors [13], CommonLOOPS [11] a.s.o., however it were influenced by the them. The concept of the objects in ODE is on the one hand made appropriate to the requirements of the editor and on the other hand as simple as possible. Further it is adaptable to other class concepts like the metaclass concept of Smalltalk or the support of multiple inheritance. Another important point is that relations are taken into the language as intrinsics and are extensively used to build intern data representations. The inheritance lattice too is constructed by relation. Relations are typed (stand in a class hierarchy too) and use inheritance. The advantages are enormous:

- relations can be used to semantic modeling of the domain
- user can ask system for special relations via coherent orthogonal functions defined over relations (i.e. can compute sets of transitives)
- operations can be applied uniformly to a set of relatives (relation) as a whole
- relations can be drawn by a uniform directed graph editor (semantic is defined by the relation)
- relations can be the foundation of rule based inference
- user externalizes knowledge about the domain (makes it explicit)
- relations gather important information rather then distribute it to many objects by using simple references
- relations can be constrained with properties like symmetry or transitivity
- constraints between different relations can be specified (i.e. some objects can’t be in two partially adjacent relations)

Indeed you can for example write your own is-subclass relation to implement your own inheritance mechanism.

The reasons for using Lisp as original for our task were the following:

- Lisp is a simple orthogonal language (therefore easy to learn)
- it is comparatively easy to write an interpreter for Lisp
- object-oriented appraoch is easily integrated
- Lisp is compact
- equivalence between data and program empowers meta programming

In addition, the subjective preferences of the author might be have played a role.

Why the multiple paradigm approach?

A multiparadigm approach is better then a pure single paradigm like OOPS because of the different adequateness of used methods in different fields of the design. A good method for specifying systems is the functional one. Most engineers are familiar with a functional way of thinking. Functions of their arguments are specified by giving a valid expression which expresses the function as a term of other functions (lastly atomic ones). Pure functional languages give no hint about how functions will be evaluated, especially there is no order of evaluation of subexpressions, so the programming style is called declarative. A second widely used style is the operational one. This is well known in the domain of programming languages as procedural style. The developer describes the sequence of actions to be taken to yield a proper result. Third there is the logical paradigm used e.g. in the Prolog language. Logical paradigms requires the specification of a number of facts (usually very large) and a set of rules over the domain. Rules are implications of a combination of preconditions. Again the programming style is more declarative then procedural, nevertheless some logical programming environments (like Prolog) require knowledge about the order of evaluation. A given abstract engine is able to analyze the workspace forward and backward to produce new facts or prove any assumptions. Other more specialized methods, for example 'access-oriented' [20] are available. One paradigm of special interest is the object-oriented paradigm. This paradigm combines the operational approach with an inheritance mechanism in a class hierarchy. Procedure call is replaced by method call.

4.4 Other confinements

There are some more pragmatical (technical) requirements for our editor tool like operation speed, usage in a standard environment or the integration into a software development environment. For the reason of speed, we used C++ as implementation language and not Smalltalk or something like this. For reasons of standardization and integration we decided to use X windows and the X toolkit [24, 25] as user interface platform, which is regarded as the standard window system in the near future and it offers all what we need. Systems like Smalltalk or KEE are very useful, they offer a lot of highlevel functionality but they are closed shops, which means that the integration with other tools writen in other languages are difficult.

5 Results

6 References

[1] G. Krasner, *Smalltalk-80: Bits of History, Words of Advice,* Addison-Wesley, Reading, Mass. 1983.

[2] Ralf L. London and Robert A. Duisberg, "Animating Programs Using Smalltalk", *IEEE Computer*?, August 1985.

[3] S.P. Reiss, "Pecan: Program Development Systems that Support Multiple Views", *IEEE Trans. Software Engineering*, Vol. SE-11, No. 3. Mar. 1985, pp. 276-285

[4] Makoto Murata and Koji Kusumoto, "Daemon: Another Way of Invoking Methods", *JOOP*, Jul/Aug 1989.

[5] R.V. Rubin, E.J. Golin, and S.P. Reiss, "ThinkPad: A Graphical System for Programming by Demonstration", *IEEE Software*, Vol. 2, No. 2, Mar. 1985, pp. 73-79.

[6] A. Borning, "Defining Constraints Graphically", *Proc. CHI 86*, Conf. Human Factors in Computing Systems, Apr. 86, ACM, pp. 137-143.

[7] W. Finzer and L. Gould, "Programming by Rehearsal", *Byte*, Vol. 9, No. 6, June 84, pp. 187-210.

[8] A.L. Ambler, "Forms: Expanding the Visualness of Sheet Languages", *Proc. 1987 Workshop on Visual Languages*, Tryck-Center, Linkoping, Sweden, Aug. 87, pp. 105-117.

[9] A. Schill, L. Heuser, M. Mühlhäuser, "Using the Object Paradigm for Distributed Application Development", In: P.J. Kühn (Hrsg.), "Kommunikation in verteilten Systemen", Proceedings : ITG/GTI-Fachtagung, Grundlagen, Anwendungen, Betrieb, Stuttgart, Feb 1989, Springer-Verlag

[10]W. Gerteis, A. Schill, L. Heuser, M. Mühlhäuser, *"DODL: A Design Language for Distributed Object-Oriented Applications"*, unpublished, University of Karlsruhe, Institute of Telematics

[11]D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik F. Zdybel, "CommonLoops: Merging Lisp and Object-Oriented Programming", In: *Object-Oriented Programming Systems, Lanugages and Applications 1986*, Conference Proceedings, Association for Computing Machinery, Sep. 1986

[12]D. Ingalls, s. Wallace, Y-Y Chow, F. Ludolph, K. Doyle, "Fabrik - A Visual Programming Environment", *OOPSLA'88 Proceedings*, ACM, pp. 176-190

[13]D.A. Moon, "Object-Oriented Programming with Flavors", *OPSLA'86 Proceedings*, ACM, pp.. 1-8

[14]Alan Kay, *"The reactive Engine"*, Ph.D. Thesis, University of Utah, Salt Lake City, Sep. 1969

[15]A. Kay, A. Goldberg, "Personal Dynamic Media", *IEEE Computer*, Vol. 10, No. 3, Mar. 1977, pp. 31-41

[16]J. Rumbaugh, "Relations as Semantic Constructs in an Object-Oriented Language", *OOPSLA'87 Proceedings*, ACM, pp. 466-481

[17]A. Goldberg, *"Smalltalk-80: The Interactive Language Environment"*, Addison-Wesley, 1984

[18]A. Goldberg, D. Robson, *"Smalltalk-80: The Language and its Implementation"*, Addison-Wesley, 1983

[19]S.P. Reiss, "PECAN: Program developement systems that support multiple views", *IEEE Trans. Soft. Eng.* , SE-11, Mar. 1985

[20]M.J. Stefik, D.G. Bobrow & K.M. Kahn, Xerox Palo Alto Research Center, "Integrating Access-Oriented Programming into a Multiparadigm Environment", *IEEE Software*, Jan. 1986

[21]R. Stallman, *"Emacs, the Extensible, Customizable Self-Documenting Display Editor"*, 545 Tech Square, Cambridge, MA 02139, USA

[22]Walter F. Tichy, Frances J. Newbery, "Knowledge-based Editors for Directed Graphs", In Howard K. Nichols and Dan Simpson, editors, *1st European Software Engineering Conference*, pp. 101-109, Springer, 1987

[23]Frances J. Newbery, "An Interface Description Language for Graph Editors", *Proceedings of the IEEE Workshop on Visual Languages*, Pittsburg, PA, October 10-12, 1988

[24]R.W. Scheifler, J. Gettys, "The X Window System", *ACM Transactions on Graphics*, Vol. 5, No. 2, Apr. 1987, pp. 79-109

[25]J. McCormack, P Asente, and R.R. Swick, *"X Toolkit Intrinsics - C Language Interface"*, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1988