An Object-Oriented Graphical Editor for Distributed Application Development

T. Leidig, M. Mühlhäuser

University of Kaiserslautern
Department of Informatics, Working Group 'Telematics'
Erwin-Schrödinger-Str., D-6750 Kaiserslautern
[+49]-631-205-2802, leidig@informatik.uni-kl.de

Abstract:

The increasing use of distributed computer systems leads to an increasing need for distributed applications. Their development in various domains like office automation or computer integrated manufacturing is not sufficiently supported by current techniques. New software engineering concepts are needed in the three areas 'languages', 'tools', and 'environments'. We believe that object-oriented techniques and graphics support are key approaches to major achievements in all three areas. As a consequence, we developed a universal object-oriented graphical editor ODE as one of our basic tools (tool building tool). ODE is based on the object-oriented paradigm, with some important extensions like built-in object relations. It has an extensible functional language which allows for customization of the editor. ODE was developed as part of DOCASE, a software production environment for distributed applications. The basic ideas of DOCASE will be presented and the requirements for ODE will be pointed out. Then ODE will be described in detail, followed by a sample customization of ODE: the one for the DOCASE design language.

1 Introduction

Looking at recent achievements in communication technology, it can be observed that high-speed networks, integrated services digital networks, and multimedia communication are presently among the driving forces of communication technology and of its merge with new ways of distributed computing. But a look at attempts to *use* distributed systems for more than simple peer-to-peer distributed applications such as ‘remote login’, ‘mail’, or ‘file transfer’ shows a different picture: computer aided manufacturing, office automation, enterprise-wide software integration remain goals hard to achieve despite a well-established infrastructure for ‘transporting the data’: there is a substantial lack of programming support and software engineering technology in these fields that would allow to produce sophisticated and highly complex *distributed applications.* Since the continuing progress in communication technology can be expected to even enhance the underlying infrastructure, an even greater demand for distributed applications can be foreseen for the future, aggravating the need for appropriate distributed application development support.

The DOCASE project, which we will refer to in this paper, is aimed at providing such software engineering and programming techniques for distributed applications. The Universities of Kaiserslautern and Karlsruhe and the Digital Equipment CEC research centre in Karlsruhe are co-operating in this project.

Object-oriented techniques and *graphics support* are key concepts of the DOCASE prototype software engineering environment for design, implementation, testing, and execution of distributed applications. The first version of DOCASE - including the mainstream development tools and proof-of-concept versions of the major workbenches (see below) - will be ready in early 1991. The marriage of object-oriented and graphical concepts is achieved in part by a ‘tool-building tool’ called ODE. ODE is an extensible object-oriented graphical editor with a customization language which also incorporates the functional programming approach. This paper will specifically describe the ODE tool within the DOCASE toolset.

Chapter 2 will present the main characteristics of the software production environment DOCASE. The principal requirements for the object-oriented graphics editor ODE are derived from there. Chapter 3 will concentrate on ODE itself. The customization facilities of ODE will be described in chapter 4 along with an overview about one of the customizations that were derived from ODE, namely the one for the object-oriented design language DODL (DOCASE Design Oriented Language) [1,2].

2 DOCASE coarse architecture

Figure 2-1 shows the main functional blocks of the DOCASE software production environment.

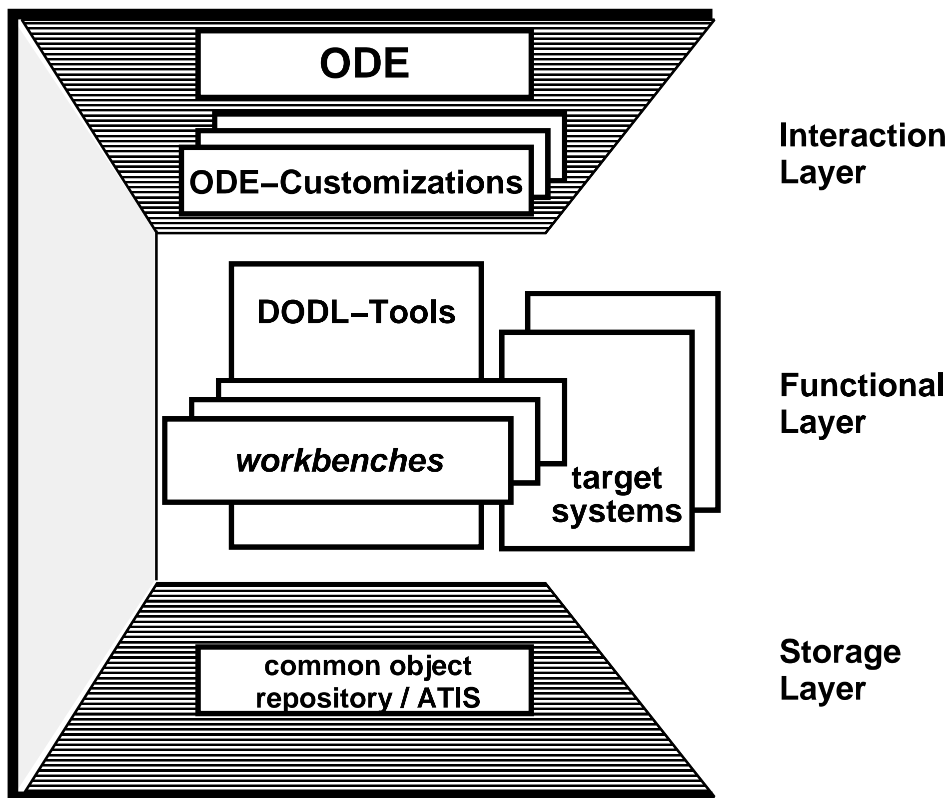

Figure 2-1 DOCASE main functional blocks

DODL tools: DODL (DOCASE Design Oriented Language) can be considered the "semantic kernel" of DOCASE. DODL is an object-oriented language with the following main features:

- Specific support for the special characteristics of distributed application (complexity, parallelism, asynchrony, dynamic creation of autonomous entities at runtime, etc.).
- Object orientation, which has proven to be an appropriate concept for distributed programming (for example, distributed object-oriented programming languages have shown to offer a high degree of network transparency).
- Compensation of major drawbacks of object-oriented techniques: introduction of an additional hierarchical structuring concept called "subsystems" (an extension of the aggregation concept which allows to group "related" objects, instead of "similar" objects as with the traditional inheritance hierarchy of object-oriented systems); introduction of a pre-defined initial inheritance tree which is reflected in all the software engineering tools of DOCASE (containing "subsystems", "configured objects", "dynamic objects", "semantic relations", etc.).

- Limited wide spectrum characteristics: DODL is both a design and an implementation language. In terms of a design language it supports incomplete specifications, graphical representations, abstractions, etc. (for the graphical representation see chapter 4). As an implementation language it supports one-to-one transformation into a distributed object-oriented programming language.

While we believe that distributed object-oriented programming languages will dominate the area of distributed applications in the future, we do not believe that a single such language will soon become a widely accepted standard. Therefore both DODL and DOCASE are designed to support different target environments with different distributed programming languages. This means that DODL is not directly translated into machine code, but into target language code first. The first target language offered in DOCASE is a distributed version of Trellis [3].

The major DODL tools are

- In the interaction layer: a customization of ODE for DODL (see chapter 4)
- In the functional layer: compiler, interpreter and animation tool (the latter is closely coupled to the ODE customization).

Workbenches: A major contribution of DOCASE is the identification and the separate but integral handling of development aspects. Distributed application development, due to the high degree of complexity of the applications, usually represents a programming-in-the-large project. This means that a significant number of development aspects like "performance optimization by simulation", "optimal object placement on the basis of object migration", "fault tolerant design" and many others have to be regarded. In the past, this was extremely difficult because of two reasons:

- The parts of the application which were specific to a certain development aspect did not represent modular and easily identifiable parts of the design or code, but were totally intermixed with the "main stream" functional development. The concentration on a single development aspect was therefore very hard. When, on the other hand, specific workbenches were developed or a certain development aspect (like simulative tools for performance optimization), they were too much decoupled from the ‘main stream’ functional development and therefore the aspects and the application were hard to keep consistent.
- Very often different tools existed for one single aspect, which were designed each for use in a specific phase of the software lifecycle. Input and output languages and formats as well as the underlying concepts and models of those tools usually differed a lot (one may think of performance optimization tools for the design phase, implementation phase, and the testing phase)

DOCASE tries to overcome those deficiencies with its *workbench concept.* A DOCASE workbench:

- Consists of a set of tools for a specific development aspect.
- Has all its tools controlled via a single graphical interaction tool (based on an ODE customization).
- Isolates the aspect specific parts of the distributed application, but clearly defines the mutual relations of the main stream functional development and of the development aspect; this is specifically supported by the DODL language, in particular via two concepts:
  - The concept of "superimposition" allows to interleave different algorithms which were developed independent of one another (e.g., the basic functionality of a distributed application may be interleaved with a fault tolerance algorithm).
  - The aforementioned "semantic relations", describing specific kinds of interrelations between objects; a workbench typically defines one or more such semantic relation types according to the development aspect they cope with (think of a 'primary / backups' relation in the context of a fault tolerance workbench).

In order to develop a new workbench, the tools have to be developed and adapted to DODL. A customization of ODE has to be implemented and DODL itself has to be extended to represent appropriate new types of semantic relations. Extensions to the runtime environments for the different target languages may be necessary as well.

DOCASE shell: The DOCASE shell consists of 'tool building tools' like ODE and tool integration concepts for inter-tool communication. It also contains a portability interface in every layer which allows it to be adapted to different user interface standards (X-Windows is currently used) operating systems, network architectures, and storage systems (currently used: Unix, TCP/IP, ATIS - see below).

Common object repository: In the storage layer, we adopted an emerging standard for object oriented development artifact management (this standard, called "ATIS", comprises models for resource, code, and version management).

3 ODE: An Universal Object-oriented Graphical Editor

3.1 Requirements for a Graphical Editor

The requirements can be roughly divided into three groups: *tool* requirements, *software engineering* requirement, and *computer assistance* requirements.

Tool requirements: Out of the requirements which are usually imposed on tools in general, we found the following to be most important in our context:

- extendibility,
- flexibility, and
- expressive power.

As pointed out in the last section, DOCASE supports the user in coping with the complexity of distributed applications and their design. One approach of DOCASE in this respect is the separation of aspects and the availability of several workbenches which analyze one aspect each. According to the DOCASE concept, the development steps are supported graphically with homogeneous graphical interfaces. Instead of building all those interfaces from scratch, we decided in favor of a 'tool building tool' which should be *extendible,* and *flexible* enough to be customized for all DOCASE graphical interfaces. In order for the necessary extensions, i.e. customizations, to be efficiently build - where efficiency means that the customization effort must be much smaller than the effort for building the interfaces from scratch -, the tool building tool was required to be offer sufficient *expressive power* for easy and appropriate extension.

Provided these requirements to be met, the 'tool building tool' approach was seen to have three major advantages:

- reduced effort for creating individual interfaces,
- increased homogeneity of the interfaces,
- better integration of the underlying tools.

Software engineering requirements: The following set of requirements concerns the software engineering aspect:

- support for design *methods*,
- support for *incomplete* specifications
- support for *hierarchical* structuring
- *locality* of substantial information, to be presented via particular 'views'.

The first point means that a 'tool builder' can provide method specific knowledge along with a customization, preventing the 'designer' (user of a customized editor) to violate against the rules of the method. To cite a (bad) example, the user may be forced to strictly apply top-down design. This type of method integration support was hardly offered by comparable tools in the past.

The second point reflects the nature of the design process: until the process is finished, it is related to an (increasing, but) incomplete amount of information about the system. While today's systems usu-

ally require specifications to be complete at the level of abstraction (or detail) they consider, we required explicit support for incompleteness.

The fact that the amount of information is growing ('particularization') motivates the third requirement: hierarchical structuring is one of the most important design principles to manage large amounts of information, so this kind of structuring was required to be offered already within the 'tool building tool'.

The last point touches an important idea of ODE. It is another issue of the philosophy of centralization and partial isolation of information pertaining to different development aspects which lead to the concept of workbenches as in chapter 2. With respect to graphical tools, the term *locality* means that all properties (data) which are relevant for the examination of a certain aspect are to be locally available. In most systems today, such data are distributed all over the system and sometimes are recorded even implicitly, so that global knowledge of the system is necessary in order to locate and derive (compute) such properties. For the graphical editor this means that it must be possible to collect such aspect specific data as defined by the designer and to present these collected data in a graphical manner, thus forming a special *view* of the system. The granularity for building such views should be kept as variable as possible. The graphical representation for the objects a the view are specific to the aspect which relates to the view.

Computer Assistance: It is generally accepted that graphical tools require a high degree of user input processing (input analysis and error detection). *Computer assistance*, in this context, means that the tool checks the user input and deduces new resulting data at least in part autonomously. The hole system should be checked against general properties like consistency and completeness. Domain specific semantic analysis should be part of the computer assistance, too. Hence we specify the following requirements for our graphical editor:

- syntax and semantic driven editing,
- extensive semantic analysis, and
- animation.

We use the term animation to denote the visualization of the system state and behavior as they evolve over time. Animation is very helpful for the developer to get a good imagination and understanding of a complex system behavior. A large amount of different parameters can be thought of for animations; these can be reflected via different graphical representations in the editor.

Especially for the DOCASE environment, where object relations play an important rule, an adequate graphical representation is necessary. In this respect, we concentrated mainly on directed graphs.

3.2 State of the Art

In this section the actual status of research in the domain of graphical tools will be reported. Up until the early 80es, CAD tools played a dominant role among graphical tools. Only with the advent of graphical user interfaces, sophisticated graphical software engineering tools were developed. Smalltalk [4,5,6], the most commonly known object-oriented programming language / environment, offers an interactive graphical user interface with overlapping windows. The user is able to switch between parallel activities without being enclosed in an fixed dialog sequence (‘modal free’). Alan Kay’s user interface paradigm [7, 9] served as a basis for the Smalltalk integrated environment which discards the borderlines between application and operating system. Smalltalk was the model for many window-based integrated environments and operating system-level windowing systems (such as the X window system [9]). These systems have many properties and mechanisms in common. Therefore, the remainder considers special aspects of graphical editors and graphical programming languages.

Because of the parallel representation in multiple windows, the problem of sharing common data and of consistency of the data and the representation came up. E.g., all instances of a class ’ClassBrowser’ have to watch the changes in the class hierarchy and reflect it in their view. The Smalltalk system handles such dependencies or constraints with its *model-view-controller (MVC)* approach. This approach will be described here shortly:

The *model* is an arbitrary collection of data that is to be represented on the screen. With the MVC approach, the model does not have any knowledge about its graphical representation in the windows. This graphical representation is taken care of by the so-called *view*, a mechanism for model representation. Only the view ‘knows’ how to display the components of a model. There are usually many different kinds of views in a system. The controller performs the interaction with the model which is represented in the view. The controller serves the input devices (e.g. mouse or keyboard) for the section of the screen which is occupied by the views window and manipulates the model if necessary. Note that the model has neither a reference to its views nor to the controller. Only view and controller can have a reference to one another.

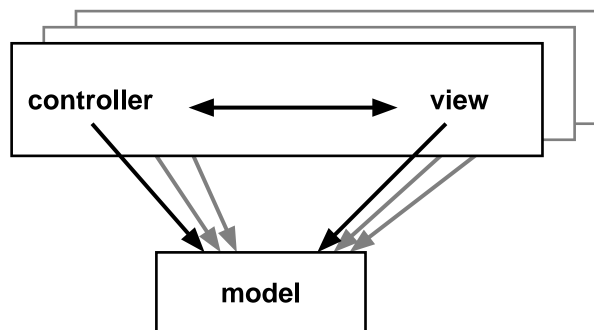

Figure 3-1 Model-View-Controller relationship

The solid lines in Fig. 3-1 represent one to one relationships, the gray ones have the meaning that a model can have different views. The picture illustrates the reason why references from models to controllers or views are not allowed: this way an application may be developed independent of its views, and the number and functionality of views can be easily altered without changes in the application.

Another approach for handling dependencies is the ‘daemon’ approach[10]. Daemons are processes in the background, watching certain activities or conditions. Daemons ’fire’ when a watched event occurs, i.e. they become active and perform a sequence of actions related to the event (e.g. actualization of the system).

The frame based system KEE [11] uses so called ’active values’. Whenever a slot is accessed or activated otherwise, a user defined action is started. Note that a new kind of programming style, called ‘data driven programming’, is introduced with this technique. Methods (= functions) are executed when certain data fields are altered, and as a result, other data fields may be altered, in turn causing other functions to be triggered. In KEE this mechanism is also used to trigger rules of a rule based inference machine.

3.2.0.1 Constraint-based programming

A further formalism for coping with dependencies between objects is represented by ’constraint-based’ languages and systems. ‘Constraints’ are logical dependencies between objects which have to be fulfilled in order to keep the system in a valid and consistent state. A so-called ‘constraint resolver’ takes care of fulfilling the constraints (constraint satisfaction). Constraints are further distinguished into unidirectional constraints (where the dependency of one object from other objects implies non-dependence in the reverse direction) and multidirectional constraints (expressing mutual dependencies). A given system of constraints (constraint net) need not have a unique solution. Depending on the kind of constraints there may be none, one, a finite set of, or infinite solutions. One advantage of constraint programming is the decoupling of the algorithm[1] for constraint satisfaction and the constraints themselves. Once an algorithm for constraint evaluation is developed, the programmer can reduce his task[2] to specifying only the constraints and run the algorithm as an abstract machine with the specification as input.

Alan Bornings ’ThingLab’ [12, 13] is a very impressive graphical constraint system. ThingLab allows to define the constraints graphically and the results of the constraint evaluation are displayed immediately on the screen. Example applications show geometric dependencies by experimenting directly

[1] for instance ’dependency directed backtracking’ or Waltz-algorithm [14, 15]
[2] in the reality the design of the constraints influences the evaluation heavily

with a graphic representation on the screen. The evaluation of constraint nets is very CPU intensive, so that time requirements for interactive systems are hardly met the workstations which are on the market today. The effective execution of constraint programs depends heavily on the fast evaluation of the constraint net. On the other hand, constraints are a powerful, uniform, and closed approach for handling object dependencies. Newer visual programming environments like 'ThinkPad' [16], 'Fabrik' [17] and 'Rehearsal World' [18] were strong influenced by ThingLab.

3.2.0.2 Programming by Demonstration

In this kind of visual programming technique, the program is described by demonstrating the function of the program through interactive manipulation on the screen. The programming system generates executable code based on these manipulations. In the 'ThinkPad' system, the manipulations on the graphical representation of data structures are translated into Prolog code. The 'Rehearsal World Theatre' generates Smalltalk code. A basic advantage of 'programming by demonstration' lies in the fact that application programmers do not have to learn the respective target language.

3.3 Concepts of ODE

This chapter describes the central concepts of the ODE editor, intended to meet the requirements introduced in section 3.1.

3.3.1 Data Model

A central property of ODE is its extendibility. ODE has two essential approaches to extendibility, namely a *simple internal data model* and an *embedded Lisp-like functional language*. The data model comprises simple types like integers, characters, booleans, strings, reals, and symbols as well as composed data types like lists and arrays. Besides, that it offers a universal object type which forms the basis for object-oriented techniques. The object model is canonical: it comprises the basic features of usual object-oriented systems and allows the adoption to individual (underlying) systems. The Lisp-like functional language serves exclusively for tool-building purposes in ODE, providing for customization to a special design method or technique. The end-user of a customized editor does not see this level, especially he *need not understand* the ODE functional language.

The reasons for using a functional programming language are as follows:

- The language is designed to model the data structures and the functional behavior of the system as far as they are specific to a given customization In this respect, the language is especially suited for extensive semantic checking of the user input.
- The language eases the implementation of simulations and animations within the editor.

3.3.2 Object Model

The design of the ODE object model was guided by the following two requirements: On one hand, the object type should be appropriate for the needs of a graphical editor; on the other hand, the object type should be elementary enough to augment it to the various object-oriented techniques which are used in software engineering, especially the design process.

An object in ODE is characterized by its class and its properties which result from the class and the class hierarchy. An object has exactly one class. The properties of an object are designated by symbols and can have either a simple type or a class type. As usual, classes are special objects that delineate a set of objects with commonalties. The class sets up the properties and operations of all objects of that class. Classes are ordered in an inheritance hierarchy, allowing 'multiple inheritance'. Apart from this common object model, an ODE object is related to other objects through relations (e.g., the inheritance hierarchy between classes is itself represented as an inheritance relation).

3.3.3 Object Relations

Relations between objects are objects themselves, i.e. instances of a special relation class. This relation class is a class just as any other class in the system, so subtyping of relations is possible. Besides, a relation instance can have properties and operations that characterizes the specific relationship. By defining operations of relations, they are given operational / functional semantics.

Relations are primitives of the ODE language. This approach has a couple of advantages which are also recognized by other authors [19, 20]:

- Powerful, uniform functions upon relations, like closures, traversing algorithms etc. can be set up independent of the special semantics of the relation.
- Relations make properties of the system explicit which are otherwise hidden in the programming code.
- Graphical editors are able to calculate layouts and draw graphs spanned by a set of relationships independent of the relation types.

As was mentioned already, the class hierarchy is realized via a special relation type, called 'is-subclass'. This is an example for a key concept of ODE: the restriction to a small but powerful set of primitives. This leads to a kind 'self reference' within ODE: the system is contained within itself to a certain degree. This property is called *self-contained* or *reflective* and has a number of advantages: e.g., the ODE editor can be modified via the functions and tools of the editor itself. For the class hierarchy, to cite another example, a radical change of the inheritance semantics of ODE can be achieved by modifying the methods of the 'is-subclass' class. The ODE class hierarchy has only one metaclass

In its basic version, the class ’class’ which is an instance of itself. Smalltalk, in contrast, defines a metaclass hierarchy in parallel to the normal class hierarchy. However, a Smalltalk-like behavior can be realized by changing the class ’is-subclass’ (and the class ’class’ if necessary). On the other hand, one can use the ODE graph editor for displaying and altering the class hierarchy since the class hierarchy is modeled via the is-subclass relation.

3.3.4 Mapping a Data Model onto a Graphical Representation

One of the major problems of graphical techniques and of course graphical editors is the choice of an useful visual representation for a design and programming language. For object-oriented systems, the issue is reduced to the question: how will the objects and their properties and relations be adequate mapped onto graphical representations? In the domain of communication psychology, this problem is coined as ’coding’ of information. The coding of a language fragment (natural or programming language) consists of sequences of words which in turn are sequences of letter symbols. A letter has no meaning of its own. The meaning comes from the words and the grammatical order of the words. The coding is one-dimensional and uses only a small set of basic symbols (the alphabet and some special marks). In analogy, we want to define graphical codes to be (at least) 2-dimensional and their symbols not to be restricted to characters. There are a set of measures and qualities which are relevant for graphical coding, e.g., absolute and relative position, absolute and relative size, thickness, extension, line styles, shapes, spatial relations, symbolic, etc. Good graphical techniques often use only a limited subset of those features because the picture should not be too complicated to read. Nevertheless various graphical techniques use various graphical characteristics that are to be supported by general graphical tools like ODE.

One important principle of ODE is the explicit distinction of a *model* and its graphical *representation*s (see Fig. 3-2), similar to the MVC-paradigm of Smalltalk. The main arguments for this concept are:

- One object can have multiple and, depending on the context in which it is displayed, also different graphical representations.
- The model is designed independent of the graphical views and the graphical representations of the objects in it.
- Views can be created upon parts of the model in a very flexible manner

This approach differs from that of many object-oriented systems where the *appearance* of an object is defined as part of the object definition.

ODE uses the abstraction of ‘*graphical dialog objects*’ introduced in the X windows toolkits [21] (DECwindows, OSF Motif). There, the term ’widget’ is used to denote graphical objects with a well

defined and determined meaning and behavior (on the underlying X windows layer, only output primitives such as windows, lines, fonts, etc., and input primitives i.e. events types, are defined). Example widgets are buttons, list boxes, editible texts, scrollbars, sliders, etc., all having both an appearance and a behavior. The appearance and behavior of widgets is defined by their class and can be parametrized for each instance. A user interface is constructed as a structured assembly and parametrization of widgets. The structure is a tree built via parent/child relations which express a control relation (children being controlled by their parents: e.g., a menu widget controls its menu-entry widgets). This structural widget hierarchy may be used for the representation of an aggregation in a graphical design editor. (Other user interfaces use a mechanism were the user can constrain the attributes of a number of graphical primitives either mutually or with respect to the application; a library of basic dialog objects is provided to ease the task of user interface construction).

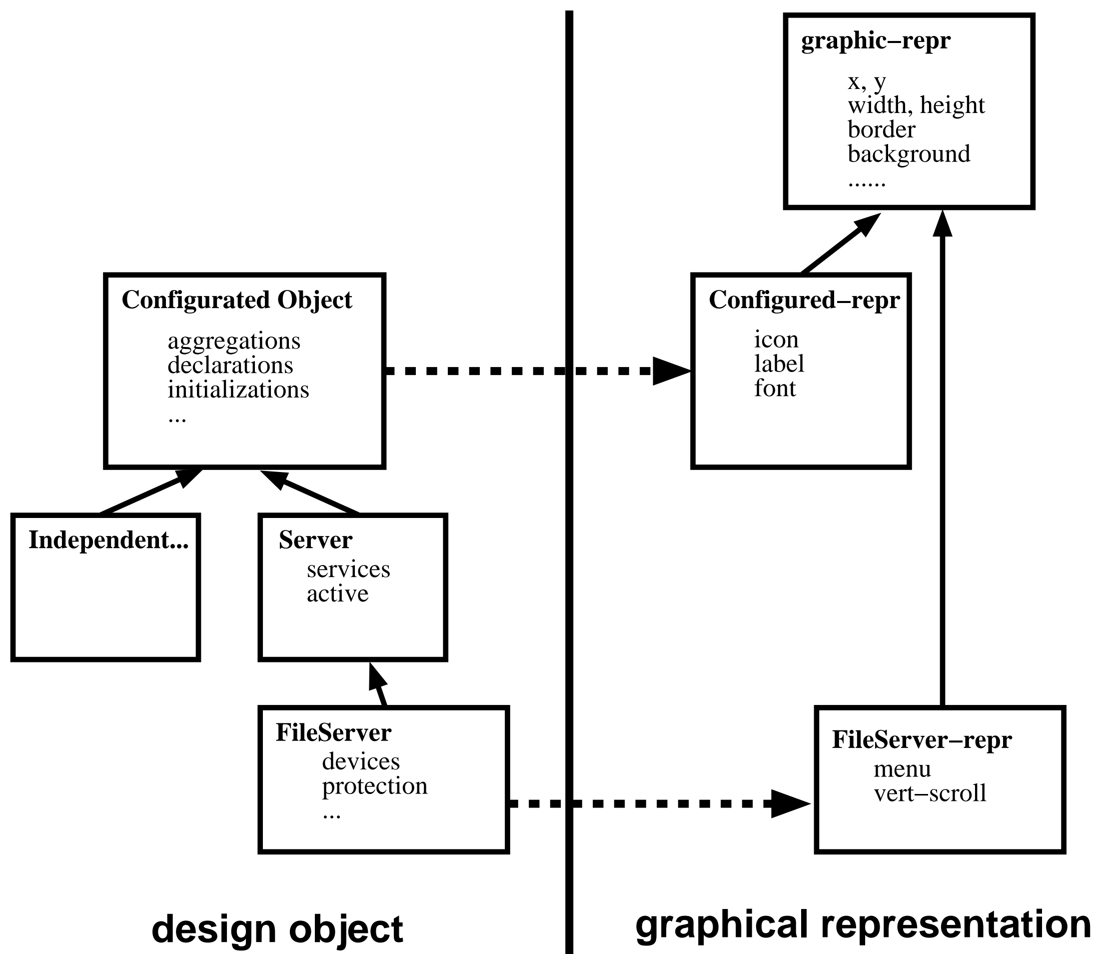

Figure 3-2 Relationship between model and graphical representation

3.3.3 Views

The term 'view' has a special meaning in ODE. A view is a perspective of a section of the system under a certain aspect of the model. In ODE, a widget is a graphical representation of a data objects, augmented by the specific reaction on user actions (input). A widget can be considered an atomic view. A composite view consists of an aggregation of widgets. A set of views can be combined and put into a window frame.

ODE offers a collection of different view types which correspond to specific ODE data types. 'Atomic views' exist, e.g., for editible texts, buttons and toggle buttons, and icons. Composite views comprise the ListView (which shows a list of objects) and the GraphView.

The GraphView is the most important composite view in ODE. It provides for displaying and editing the graph spanned by a set of objects and the relations between the objects. Objects in the graph are displayed as labeled buttons, the button may contain an additional icon for the object. Relations are displayed as edges between the object nodes and can be parametrized with attributes such as linewidth, arrow style, arrow size, label , etc. The relations considered in a graph can be restricted to a set of particular relation types. The layout is performed automatically whenever the graph changes. ODE is intended to offer a spectrum of domain-specific layout algorithms, although only a small number of alternatives is implemented at present. The kernel of the most sophisticated among the current layout programs was developed by another group [22]. Layout algorithms are hooked onto ODE with a special mechanism and can be added at any time.

As mentioned, a key concept of ODE is the generation of views from a model; views are maintained automatically if the model changes, allowing multiple independent views of a common model. This means that automatic layout is absolute required, since the user can usually not be asked for interactive object placement while a view is constructed. On the other hand, the user has to have the possibility to alter the model through manipulations of views. In ODE, such a manipulations does not change the view directly, but rather invokes functions on the model which in turn controls (and hence, changes) the views. (Note that many graphical editors work differently: often, the user edits persistent graphical objects which then are transformed into other representations for analysis or execution.)

Research on automatic graph layout is still in a comparatively early and unsatisfying stage. In particular, the work on general universal layout algorithms for planar graphs is not satisfying yet. Fortunately, many layouts for graphical design methods (e.g., trees and flow charts) are comparatively simple and the results of layout algorithms are acceptable. However, several graphical design methods require sophisticated planar layout. Often, the amount of relations between objects is so such large that edge crossings are unavoidable and an appealing layout is impossible (this can occur, e.g., if many

different relation types are combined in one view). Such effects reduce the acceptance of graphical tools drastically, since the typical advantages of graphs such as intuitive understanding, are lost.

One simple approach in the GraphView to reduce such effects is the support for abstraction of graphs. A specific relation type can be defined as the ‘abstraction relation type’ (e.g., ‘part-of’). The relationships of that type are not drawn as edges between the related object nodes, but are used to form subgraphs of the current graph. Note that any relation type (if it matches certain requirements) can serve abstraction relation type. Subgraphs can be displayed in the outer graph in three different manners:

A. A subgraph can be displayed as a simple node, hiding the contents of the subgraph.

B. A subgraph can be displayed as a box containing the subgraph; all edges from an outer node to nodes of the subgraph are drawn as a single edge from that outer node to the subgraph box (‘zoomed-in state’).

C. Analogous to B., all edges between the objects of inner and outer graph can be displayed (’glass-box’ or ’white-box’ state).

Figure 3-3 Sample Petri-net editor view

The Fig. 3-3 shows that the views which are combined into a frame set up a user interface through which the editor is managed.

3.3.6 Managment of Constraints

The dependencies between the graphical representation and parts of the model were described already in section 3.2. In ODE we were first following the ‘active-value approach’ as used in KEE. This approach has the advantage to be very fast, because the strategy for evaluation of constraints is supplied by the programmer and can be optimized according to the problem. A major drawback lies in the complexity of constraint satisfaction to be managed by the programmer, resulting in a high programming error probability. We currently implement a declarative constraint specification language plus constraint satisfaction machine. Another approach under investigation is the use of rule-based specifications plus a rule interpreter (inference machine). This would lead us to a true multi-paradigm environment.

4 Customization of ODE for the DODL language

We want to use the term *DODE* in the remainder to denote the customization of ODE to the design editor for the DOCASE design language DODL.

4.1 Application Domain: ‘Visual Design Languages’

Graphical editing of design specifications is a kind of visual programming as introduced in section 3.2.

While programs in textual languages are a sequential arrangement of words of the language, visual programs are an arrangement of graphical elements in a (at least) 2-dimensional space. The spatial ordering and the connections of graphical objects carry the syntactic and semantic information.

In the case of DODL (and most other design languages), it is not useful to use a pure graphical language, because not all parts of a system design can be given in an understandable and easy to grasp graphical representation. We therefore use a hybrid approach, i.e. we allow (any) part of the input to be textual (in addition, we use text for naming of course).

DODL is an object-oriented design language with a pre-defined initial type hierarchy, generic types, inheritance, methods and other typical properties of o-o lanugages. A lot of suggestions exist for graphical notations of object-oriented design. The standardization process for graphical conventions is progressing rather slowly. Respective techniques comprise OOSD [24, 25], HOOD [26] , Booch’s diagram set [27], and OMT [28]. These techniques are not described here in detail, we rather concentrate on the similarities and differences to DODE.

We created a set of diagram techniques for DODE (in part motivated by the related work cited above). Each diagram technique comprises one aspect of a distributed application.

The DODE notations for the major diagram techniques are: TypesDiagram, MatrixDiagram, and MethodDiagram.

4.2 Diagrams and Views of DODE

DODE offers a set of different views for the graphical notations for DODL described above. Views are provided for the various diagram techniques, mainly derived from the ODE GraphView. The objects in the GraphViews are represented through icons according to the view type (diagram type). It is possible to provide application domain specific icons if they are the result of an augmentation of the type hierarchy of DODL. N-ary Relations in DODL are represented as spine objects and thus resolved to binary relations.

The *TypesDiagram* editor view shows the type hierarchy of the DODL types. The user can browse through, augment, and change the type hierarchy with it. The type hierarchy is modeled by a subtype relation which is implemented as a subclass of the ODE relation class. The graph of the type hierarchy is hierarchical, directed, and non-cyclic (which is ideal for graph layout). The ClassDiagram can also be used to show other 'class-to-class' relations such as special generic relations between DODL types. The TypesDiagram is somewhat similar to the ClassDiagram of Booch [27].

Object types are defined via the *MatrixDiagram* editor (in analogy to the Oblog project). It shows both the instance variables of an object type *and* the operational interface of the object type (methods), plus the composition of the object type (most techniques in the literature are non-graphical , using form based textual approaches to specify instance variables and methods).

The *MethodDiagram* is used to edit the sequence of control flow statements inside an object method. The statements are displayed in a style similar to that of a flow diagram or a structure diagram.

The *ConfigurationDiagram* is used for editing the initial configuration of the objects of a distributed application. Here, real initial instances are considered instead of types.

More special views regarding special aspects of distributed systems design are under development. Other views are supplied for editing objects in the style of dialog boxes with various dialog primitives. User defined views can be established to perform special task like animation of specifications.

4.3 Realization with ODE

For customizing ODE into DODE, one had to select and implement the appropriate data representations of the DODL constructs in ODE data types first. The 'object type' of ODE is the most important type used here. Next, the ODE modeling of DODL was to be implemented. In this step, the level of detail depends on the intended use of the model. E.g., semantic analysis requires much more detailed

modeling than simple editing. The construction of the model consists the substeps of identifying objects, classes, and relations in the application. Note that relations in the application domain can be represented by direct references (implicitly) or by instances of a relation type (explicitly). Representation via relation types has the advantage that generic ODE functions can be used and the disadvantage of requiring more storage and computation time. Along with the class and relation hierarchy, syntactic and semantic requirements are defined within the class and relation methods (cf. Examples A in the appendix). In parallel to the modeling of DODL, the necessary views were derived from pre-defined views of ODE (cf. Examples B).

Views have to be assigned to windows. The windows in ODE that carry the views in ODE are called 'frames'. A frame consists of a title bar, pulldown menus, and a control panel (see Fig. 3-3). The menu entries are attached to ODE-Lisp expressions which are evaluated whenever the entry is activated. Thus, the frame represents a command interface to the contained views.

The description of style parameters of the GraphViews is given in a especial 'Graph Description Language' (GDL) [23], which has a look similar to the X default resource description (cf. Examples C).

The icons can be supplied using any X window icon editor, scanned images, etc.

Supplementary auxiliary functions are necessary, e.g., for generation of DODL language output from the model representation, for specific semantic checks, simulations, etc.

5 Summary and outlook

A powerful sophisticated tool building tool for the development of object-oriented graphical user interfaces has been described. Its customization to a specific object-oriented distributed programming language was sketched.

ODE is continuously enhanced. At present, new submodules for graph layout and for constraint resolution are added.

Apart from DODL, ODE has been customized with great success to quite a number of areas in and outside the DOCASE project, e.g., to a massive-parallel-program design system and to a CIM (computer aided manufacturing) programming environment.

6 References

[1]

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

[2]

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

[3]

C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt, "An Introduction to Trellis/Owl", *OPSLA'86 Proceedings*, ACM, Oct. 1986

[4]

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

[5]

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

[6]

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

[7]

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

[8]

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

[9]

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

[10]

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

[11]

Renate Kempt, "Teaching object-oriented programming with the KEE system", *ACM SIGPLAN Notice*, 22(12), pp. 11-25, Dec. 1987

[12]

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

[13]

A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, M. Woolf, "Constraint Hierarchies", *OPSLA'87 Proceedings*, ACM, pp. 48-60

[14]

R. Stallman and G.J. Sussman, "Forward Reasoning and Dependency-directed Backtracking in a System for Computer-aided Circuit Analysis, *Artificial Intelligence,* 9(2), 1977

[15]

D. Waltz, "Understanding Line Drawings of Scenes with Shadows", In: Patrick H. Winston, editor, *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975

[16]

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.

[17]

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

[18]

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

[19]

H. Boley, "RELFUN: A Relational/Functional Integration with Valued Clauses", SIGPLAN Notices 21(12), Dec. 1986, pp. 87-98

[20]

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

[21]

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

[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]

A. I. Wasserman, P. A. Pircher, R. J. Muller, "An Object-Oriented Structured Design Method", ACM SIGSOFT, *Software Engineering Notes*, Vol. 14, No. 1, 1989, pp. 32-55

[25]

A. I. Wasserman, P. A. Pircher, R. J. Muller, "Concepts of Object-Oriented Structured Design", *Proceedings of Tools'89*, Paris, Nov. 1989

[26]

M. Heitz, *"HOOD Reference Manual"*, CISI Ingenierie, Midi Pyrénées, Sep. 1989

[27]

Grady Booch, "Object Oriented Design with Applications", Benjamin/Cummings, 1991

[28]

M. R. Blaha, W. J. Premerlani, and J. E. Rumbaugh, "Relational Database Design using an Object-Oriented Methology", *Communications of the ACM*, Vol. 31, No. 4, Apr. 1988, pp. 414-427

7 Examples

7.0.0.1 Examples A:

```
(Class class 'DocaseObjectType)
```

Meaning: Create a subclass of class 'Class' that is named 'DocaseObjectType'.

```
(DocaseObjectType inherit: 'types-section )
```

Meaning: The 'DocaseObjectType' has the property 'types-section', default value **Null**.

```
(DocaseObjectType method: (print-it)
        (print "DocaseObject\n")
        (types-section print)
        ...)
```

Meaning: DocaseObjectType has a method named 'print-it' without parameters.
This method calls another method 'print' on a sub-object.

```
(Relation class 'component-of)
```

Meaning: Creates a new relation type named 'component-of'.

Examples B:

```
(GraphView class 'DODL-TypesView)
(DODL-TypesView inherit: 'zoom component-of)
(DODL-TypesView inherit: 'style "DODL-TypesView.gdl")
```

7.0.0.2 Examples C:

```
graph:
  xbase: 20
  ybase: 20
  xspace: 50
  yspace: 50
  scaling: 1
  orientation: top-to-bottom
  topsort: bottom
  displayedges: no
  layout.algorithm: sugi
...
  node.DocaseObject.borderwidth: 0
  node.DocaseObject.bitmap: "../icons/root.xbm"
  node.DocaseObject.font: "-Adobe-Times-Bold-R-Normal--18-*"
  node.DocaseObjectType.borderwidth: 0
  node.DocaseObjectType.bitmap: "../icons/basic.xbm"
  node.DocaseObjectType.font: "-Adobe-Times-Bold-R-Normal-18-*"
  node.ConfObjectType.borderwidth: 0
...
  edge.has-component.arrow.style: none
  edge.has-component.thickness: 1
  edge.in-relation.arrow.style: solid
  edge.in-relation.arrow.width: 5
...
endgraph:
```