From drafting to design: new programming tools are needed

C.Zelissen
College of Advanced Technology/13P Consulting Engineers (The Netherlands)

SUMMARY

The software needed by engineers and architects shows two new aspects. First, these programs get more and more graphic elements, secondly there is a trend from general purpose packages to more problem oriented programs.

Comparing several of these application depending programs, a strong similarity appears; a user builds up a representation of a (technical) model by placing, replacing, deleting and so on, representations of objects, belonging to this model.

From the programmer's point of view, it must be possible to abstract the several models and the actions on the components of a model, and therefore to build one program with a modeldescription as parameter.

Assuming the existence of such a program, the only remaining part needed to build a complete dedicated package has reference to the specific technical calculations.

In this contribution we touch on a number of the problems in developing and implementing such a program.

THE NEED FOR NEW TOOLS

The term CAD causes a lot of misunderstanding. In our opinion CAD software has to give to the designer a maximum of support, (not restricted to the graphical presentation of a design and the interaction on it only) also the software must warn the user against constructing non logical relations and furthermore the software should give information about the quality of the design.

A general purpose drafting system does not have these properties. At best one can link a library of symbols or one has a number of typical (for instance architectural) drafting functions at one's disposal. Sometimes it is possible to order the system to count the number of symbols used in a design or other, comparable, although very restricted information.

At this moment m growth in problem oriented software along two separate ways. Can be seen/observed on the one hand programs (or modules) appear which use output from a drafting system as input. An engineer or architect formulates his problem graphically with the aid of a drafting system and the (separated) additional module carries out the calculations.

The correct description of the problem (in terms of the mental model as well as the inputsyntax of the module) belongs to the responsibility of the designer and the batch oriented designprogress are great disadvantages of this strategy. (See figure 1).

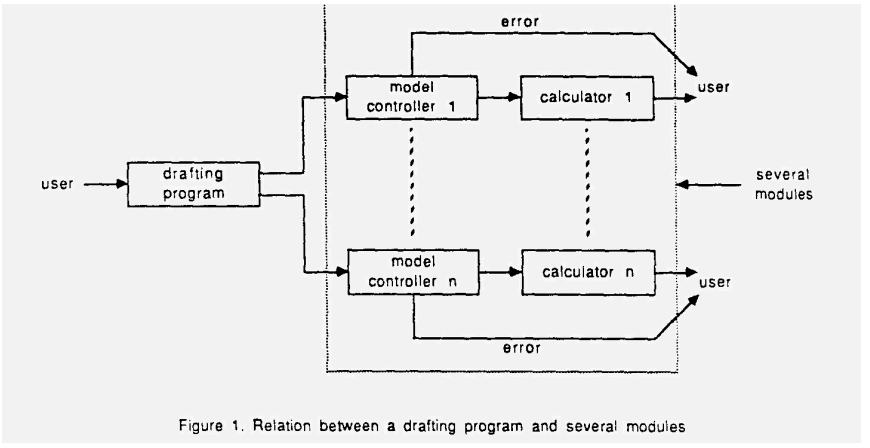

Figure 1. Relation between a drafting program and several modules

The second way shows integrated software, tuned on a specific problem, using the symbolism and algorithms of that type of problem. The software interactively supports the designer during the formulation of his problem and protects him against contradictions.

Integrated problem oriented software (IPOS) fits the designers' behaviour and the mental world much better. For this reason it is to be expected that integrated software will be of great importance in the future.

To build IPOS a lot of effort is required and the area of distribution is small, so one has to search for possibilities to decrease developing costs. Analyzing these packages one recognizes three parts:
* the workstation manager;
* the model manager;
* the calculator.

The tasks of the workstation manager are to control the workstation, (screen layout, graphic output and eventhandling), to inform the model manager about useractions and to pass messages from the model manager to the user. The model manager checks the correctness of the userinput by comparing the arised state with the allowed states. If the action was correct, then the workstation manager will be ordered to carry out the action, otherwise the workstation manager has to send a message to the user about the cause of the problem. If the modelmanager receives the request to calculate, it determines whether the current state is adequate enough and if so, it orders the calculator to do its task and to handle over the results, otherwise the user will be informed about probable causes. Figure 2 shows the relation between the managers.

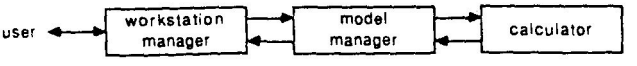

Figure 2. Structure of IPOS

The biggest parts of the source code are the workstation manager and the modelmanager. The calculator mostly consists of about 20 percent of the whole source code.

Until now a programmer of IPOS only has tools to solve the problems of the workstation manager. Graphic libraries like GKS or IGL contain a number of functions to represent a Virtual Device Interface (VDI), but have no standard facilities to build the user interface. They lack functions for window, dialogue or menumanagement. This part is called the Application Environment Service (AES). The more advanced graphic libraries consist of these two parts. A wellknown example is the MacToolbox.

For the other two managers of IPOS restricted tools are available. There are groups of problems which use the same mathematical methods to find a solution. Finite element problems are reduced to solve a set of linear equations.

If there are tools to support the calculator, then the main task remaining for the programmer is to convert the model information into the desired format, in other cases the calculator has to be developed completely. At this moment the only tools for the model manager are database functions, but they are very restricted. So now we have reached the base problem: the need of tools to build model managers.

Before going more deeply into the needed tools we must first pay attention to the concept of a model.

MODELS

Reality consists of a collection of objects and a collection of relations between these objects. A set of objects can form a new object. The objects "The black leather seat" and "The black leather back" are parts of "My desk-chair'. Models are used to describe the structure and behaviour of physical or abstract objects. We restrict ourselves in analyzing these models to engineering or architectural environments.

In the way objects define a reality, components are related to a model. Components with the same (physical) properties define a component type. Small and big doors, belong to the component type "door".

To each component type a list of characteristics is related. For the component type "door" elements of such a list are size, colour, quality.

Between component types there is also a number of relations. The relations can have reference to physical properties or topological relations. An example of a topological relation between types could be "a door is always a part of a wall", assuming that "door" and "wall" are component types. "The lengthening of a beam is proportional to the magnitude of the axial force" is a physical relation between the component types "beam" and "axial force".

In building a model one tries to describe the collection of component types and the collection of the relations between types in such a manner that they are in some way representative for a reality.

There are different models for frame design, electrical board design, interior design, each model with a collection of specific component types and their relationships. In analyzing the several collections of relations one discovers a similarity in relations belonging to different models. In the model used by printed circuit board design it is not allowed to locate two components "resistance" at the same place. A similar situation can be seen at the design of an interior layout: the designer will not position two tables (they are representatives of the component type "table) at the same location.

Especially the relations describing the topological connections between component types seem to have a more general character, not bound by a typical model. Is it possible to determine a number of abstract, topological relations, not chained up to component types of some model?

To get a better hold on the problem of topological relations we concentrate for the remainder on two dimensional geometrical models.

If a designer builds up (in his mind, on paper or display) a representative of a model, he defines representatives of components (occurences) and executes a number of geometrical actions like rotate, translate, on the occurences. During this process the designer takes care of the correctness of the topology (the set of the topological relations).

We have introduced a third element in our model definiton. A model consists of component types, a set of relations, but also to each component type there is a set of actions. Each component can be positioned or deleted, provided that it does not conflict with the relations. Some components are not to be rotated or scaled: a support in a frame model is not to be scaled.

The basic types of topological rules are:
* the exclude rule,
* the connect rule and
* the include rule.

In the exlude rules will be recorded which components may not have any contact: "two doors in the same wall are situated on a minimal distance of 30 cm." The connect rules describe in which manner components are linked up: "the door fits in the doorframe'. There are many types of connections, thinking of the way objects can be connected in reality. Components are sometimes locked up in other components: 'the door is positioned in the wall". These situations are the subjects of the include rules.

The topological rules are not bound to generally accepted rules belonging to one or an other typical model. It is imaginable that some rules have an individual character, depending on the preferences or experiences of a designer.

At this moment it is senseless to elaborate on these rules more, because there is a strong reference between these rules and the geometrical representations of components. If we summarize what has been discussed so far, a model is a set of component types and a set of relations. To each component type belongs a description and a set of permitted actions.

STRATEGIES FOR A SOLUTION

Suppose a system designer has to build several integrated problem oriented systems. He can choose between two strategies. Starting from the first strategy he constructs a number of functions, collect them in a 'model' -library and uses these routines in every IPOS. The functions enable him to implement a specific model. The second solution consists of building only one program, that can handle a more formal model. The actual model is described in a so called modelfile. During the start up the program consults such a modelfile and afterwards acts in accordance with the model described. Figure 3 shows this structure.

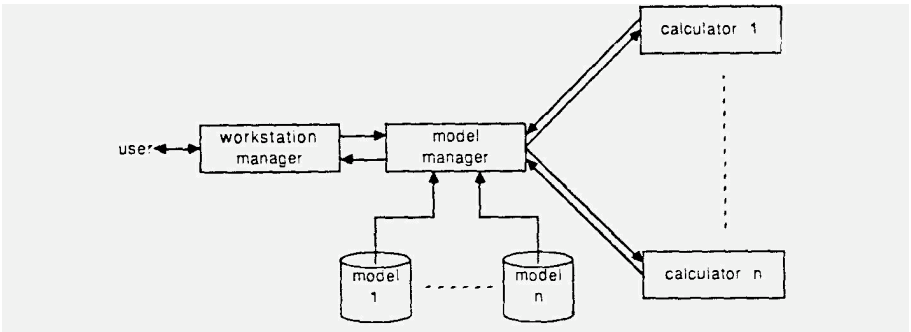

Figure 3. The structure of a general model handler

This last strategy is very complex, as one has to define the boundaries for all the actual models allowed. On the other hand in cases of generating IPOS it is more efficient. The only things to be done are to construct a model file and to program a calculator. Apart from that, if one would like to build an IPOS-generator, it is wise to do that in a modular way. Certainly one module should consist of a number of functions for the modelhandling, so the first strategy is in fact a part of the second one.

At 13P we are building a prototype of a general integrated problem oriented model handler.

PROBLEMS OF IMPLEMENTING A FORMAL MODEL

As we already stated before, there is, at the implementation level, a strong relationship between the geometries of the component types and the topology of the model. So first we focus our attention on the geometrical part.

When describing the geometry of a component type one has to decide to which coordinate system the component type belongs. There are two classes, depending on the coordinate system. The types of the first class are related to concrete objects. The sizes of these types are expressed in mastercoordinates. Components representing objects, like walls, are examples of this class. These component types are called realistic. Components related to 'abstract' objects are indicated by the term symbolic Examples of such objects are forces in a framework. The geometrical description of the symbolic types is done in a normalized coordinate system. Each component type has a geometry. A geometry consists of a number of (output) primitives. The allowed primitives determine the scope of the formal model. So if the only accepted primitive is a line segment, it is evident that an actual model can not simulate an enviroment of objects with circular shapes. The primitives that determine the geometry of realistic component types differ from symbolic component types. The geometry of the component type 'wall' for example is a rectangle, expressed in mastercoordinates. The component type 'force' is built up from a number of line segments, expressed in the normalized system.

A list of the allowed geometrical actions belongs to each component, too. Two actions are implicitly valid for every component, to place and delete (end therefore also translate). The other actions are scaling and rotating in several variations. The symbolic types mostly have no scaling rules, as contrasted with the realistic types. The transformation 'rotate' appears with realistic as well as symbolic types. To carry out the transformations reference points are needed as a part of the component type description.

Mostly each type has several technical parameters. The formal model has to offer possibilities to describe these parameters, including dimension, default and range values. In a framework model the component type 'beam' has parameters like 'moment of inertia' or 'area of intersection'. These values are needed by the calculator. The general model handler must allow the editing of the parameter values.

So until now the description of a component type consists of - a name; - a class typing (realistic or symbolic); - a list of reference points; - a list of output primitives and their attributes; - a list of geometrical actions; - a list of (technical) parameters.

The next step is to show a number of problems linked up with the topology of a formal model.

In viewing output pipeline the geometry of realistic component types is transformed to the normalized device. So the topology at the implementation level is based on geometrical properties of the normalized coordinate system. This system is a subset of the twodimensional euclidean space. The include and the exclude rules are relatively simple to implement.

The geometry of a component contains a number of points in [0,1] x [0,1]. Both rules can be expressed in terms of the set theory. Two component types exclude each other if the intersection of the sets of points belonging to them is empty. A component type A is a part of the component type B if the set of points of A is a subset of the set of points of B. Let us demonstrate these two rules. Suppose there are three component types A, B and C. The geometry of component type A is a filled rectangle with fill style hollow, B is a rectangle and C is a triangle. Assume the following topology rules are valid: A and C exclude each other and C is included in B. In figure 4 several valid and invalid states are shown.

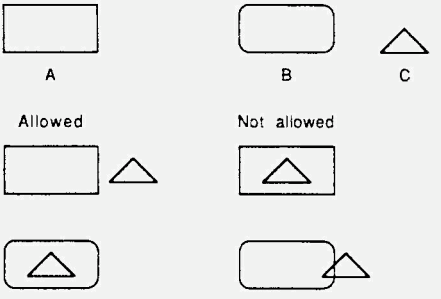

Figure 4. Include and exclude rules

The implementation of an include rule causes some special problems, e.g.: the effects of the actual attributes of both geometries on the final result. Suppose C is also a filled triangle and C is located in B, which attribute should get the priority? The include rule must also handle the attributes.

The most difficult rules are the connect rules. There is an extended range of possibilities and a second problem is the fundamental difference between the normalized and the device coordinate system. In a normalized system, between two points there is always another point, in the (discrete) device system it is possible to indicate two nearest points. So by implementing connect rules one has to attach different meanings (in different spaces) to connecting. Now the problem of how to formulate 'connecting two components' in the normalized system will be dealt with. In geometry and in reality the ideas 'connect' differ. In geometry two figures connect if there is at least one point of contact. A point of contact is a common point. So the intersection of the sets of points is not empty. Objects however connect if there are points of the two objects which are 'very near' to each other. The idea 'near' differs in several situations. We think the connect rules must not be based on geometry, but be comparable with reality. So a connect rule must hold information about the interpretation of 'near', expressed in the component type depending coordinate system.

Connect rules depend on shapes of the component types. For simplicity we assume that the only allowed primitive is a straight line segment.

Some possibilities to connect component types are:
- a point-point connection;
- a point-stroke connection and
- a stroke-stroke connection.

Two component types can have a point-point connection if each component contains a point that allows a connection and if the topology permits the connection between these two points. A similar definition is valid for a point-stroke connection. (A stroke is an interval of a line segment). More problems come forward in defining a stroke-stroke connection: there must be a 'common' stroke. To precise the meaning of 'common' forces us to use a very mathematical definition. In figure 5 several variations are given of stroke-stroke connections.

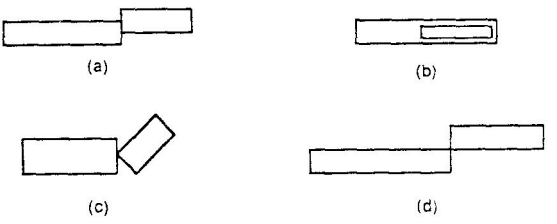

Figure 5. Several stroke connections

One can imagine that the difficulties will grow enormously if the allowed primitives to describe the geometry are extended to rounded shapes.

Not all problems have been solved yet. The connect rules must also solve the problem of the attribute choice. If two strokes with different attributes connect, which attribute will have the resulting connecting stroke?

The next simple example will illuminate how connect rules act. In our model we have 3 component types:
- a beampart,
- a support and
- a force.

A beampart is represented by a filled rectangle, a support by a triangle and a force by an arrow. The beampart has four line segments: the left and right ones allow a stroke-stroke connection with the opposite line segment of a beampart, the upper side allows a point-stroke connection with the 'arrowhead' of a force, and the lower side allows a point-stroke connection with the 'top' of the support. In figure 6 one result of this topology is shown.

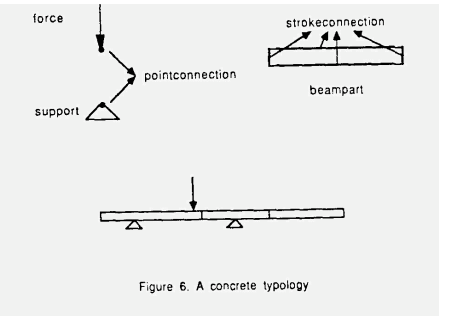

Figure 6. A concrete typology

MACBEAM, A FIRST PROTOTYPE

At 13P we have developed a prototype, based on the ideas discussed above. At this moment the model description is still an integral part of the program. A number of the topology problems have been solved and the program is a representative of IPOS. In future we hope to isolate the model description and to introduce facilities which will enable us to define topology rules based on the experiences of the designers.

REFERENCES

Encarnacao, J. and E.G. Schlechtendahl
Computer Aided Design Berlin, (1983).

Foley, J.D. and A. van Dam
Fundamentals of Interactive Computer Graphics Addison-Wesley, Massachusetts, (1982).

Newman, W.M. and R.F. Sproull
Principles of Interactive Computer Graphics McGraw-Hill, New York, (1978).

Spur, G. and F. Krause
CAD-Technik Hanser, Munchen, (1984).

Wagter, H.
'A realistic view on the use of CAD techniques in architecture and building design', Proceedings of CAD86 London, (1986).