Preliminary Proceedings

of the Third Eurographics Workshop on

Intelligent CAD Systems

"Practical Experience and Evaluation"

Hotel Opduin, Texel, the Netherlands

April 3 - 7, 1989

CWI

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

Preliminary Proceedings

of the Third Eurographics Workshop on

Intelligent CAD Systems

"Practical Experience and Evaluation"

Hotel Opduin, Texel, the Netherlands

April 3 - 7, 1989

FOREWORD

Welcome to the *Third Eurographics Workshop on Intelligent CAD Systems* (Workshop theme: *Practical Experiences and Evaluation*). It is our pleasure to have you here at Hotel Opduin, on Texel.

The present volume serves as a preliminary record of the workshop. A selection of the presented papers will appear in the final proceedings, to be published by Springer-Verlag (hopefully towards the end of this year). The proceedings of the first workshop were already available at the last workshop. Unfortunately, due to some delay in the processing of last year's proceedings, these won't be available right now. A draft can be found on the desk with publicity materials.

Please note that, in addition to the presented papers and additional contributions, there are a number of position papers of the participants who haven't contributed a full paper. Contrary to what happened last year, the programme committee members will decide which papers, based upon contents and presentation, will be included in the final proceedings.

We realise that this is the last workshop in a series of three, and we have some regret about that. However, we are already discussing the possibility to continue this event and we will keep you informed about forthcoming workshops. Meanwhile, we wish you a very pleasant and fruitful workshop.

Workshop Organisers:
Marja Hegt, Paul ten Hagen, and Paul Veerkamp
Centre for Mathematics and Computer Science (CWI), Amsterdam

TABLE OF CONTENTS

WORKSHOP PROGRAMME

Monday, April 3

15:15 - 15:30	Shuttle bus at Amsterdam Central Station
18:00	Arrival at workshop site
18:00 - 19:00	Registration for Programme Committee members and session chairpersons
19:00	Dinner
21:00 - 22:00	Registration for other participants
21:00 - 22:00	Meeting of Programme Committee and session chairpersons

Tuesday, April 4

7:00 - 8:30	Breakfast
8:30 - 8:45	Welcome speech by Prof.Dr. P.C. Baayen, scientific director of the Centre for Mathematics and Computer Science
8:45 - 9:15	Opening speech by co-chairperson P.J.W. ten Hagen
9:15 - 9:30	Coffee break
9:30 - 11:45	Paper session **Design Process** (moderator: Aart Bijl) *Jan Treur* : A logical framework for design processes *K.D. Baker, L.J. Ball, P.F. Culverhouse, I. Dennis, J.St.B.T. Evans, A.P. Jagodzinski, P.D. Pearce, D.G.C. Scothern, and G.M. Venner*: A psychologically based intelligent design aid *Jan Rogier*: The use of STEP in an Intelligent Design System for Architectural design
11:45 - 12:15	Coffee break
12:15 - 13:00	Discussion on Design Process
13:00 - 15:00	Lunch
15:00 - 17:15	Paper session **System Architecture** (moderator: Ken MacCallum) *Jean-Paul A. Barthes, Kamal El Dahshan, Patrice Anota*: An experience in adding persistence to intelligent CAD environments *A.T. Shenton, A. Taleb-Bendiab, and Y. Chen*: Computer-Aided Constraint Development Systems for Conceptual and Embodiment Engineering Design *Tapio Takala*: Application of history mechanism in architectural design
17:15 - 17:45	Tea break
17:45 - 18:30	Discussion on System Architecture
19:30	Dinner

Wednesday, April 5

7:00 - 8:30	Breakfast
8:30 - 10:45	Paper session **Languages** (moderator: Farhad Arbab)
	Paul Veerkamp, Ravic Pieters Kwiers, and Paul ten Hagen:
	Design Process Representation in IDDL
	Luis A. Pineda, Ewan Klein:
	On the Integration of Graphical and Linguistic Knowledge for CAD Systems
	Tetsuo Tomiyama, Deyi Xue, and Yoshiki Ishida:
	An Experience of Developing a Design Knowledge Representation Language
10:45 - 11:15	Coffee break
11:15 - 12:00	Discussion on Languages
12:00 - 13:30	Lunch
13:30	Bus leaves for excursion
18:00	Bus returns at workshop site
19:00	Dinner

Thursday, April 6

7:00 - 8:30	Bird watching (bicycle)tour for those interested
7:30 - 9:00	Breakfast
9:00 - 11:15	Paper session **Geometric Reasoning** (moderator: Paul ten Hagen)
	Rajiv S. Desai, Rajkumar S. Doshi, Raymond K. Lam:
	GARE: Geometric Analysis and Reasoning Engine
	Can A. Baykan and Mark S. Fox:
	Constraint Satisfaction Techniques for Spatial Planning
	Farhad Arbab, Bin Wang:
	A Geometric Constraint Management System in Oar
11:15 - 11:45	Coffee break
11:45 - 12:30	Discussion on Geometric Reasoning
12:30 - 14:30	Lunch
14:30 - 16:45	Paper session **User Interface** (moderator: Jean-Paul Barthes)
	Aart Bijl:
	POINTER: Picture Oriented INTERaction
	A Programme for ICAD/HCI Research
	Christine Giger, Michael Lutz, Luiz Ary Messina:
	An Intelligent NC-Programming-System as a Significant Extension of Intelligent CAD-Systems
	Zsófia Ruttkay, Paul J.W. ten Hagen:
	Intelligent user interface for intelligent CAD
16:45 - 17:15	Tea break
17:15 - 18:00	Discussion on User Interface
19:00	Closing dinner with music

Friday, April 7

7:30 - 9:00	Breakfast	
9:00 - 11:00	General Discussion	(Moderator: Paul Veerkamp)
11:00 - 11:30	Coffee break	
11:30 - 12:00	Closing session	
12:00 - 13:15	Lunch	
13:30	Shuttle bus leaves for Schiphol airport and Amsterdam Central Station	

Paper Session *Design Process*

Jan Treur

A logical framework for design processes

A logical framework for design processes

Jan Treur
Programming Research Group
University of Amsterdam

0. Introduction

The aim of a design process is to construct an explicit description of the structure of a certain object satisfying a list of design requirements given beforehand. This explicit description must be sufficiently detailed to guarantee that the object with this structure actually can be built (it should be realizable). On the other hand it may happen that the outcome of a design process is the assertion that there does not exist an object satisfying the requirements (inconsistent design requirements). In this case one may change some of the requirements and try a new design process.

Designing often involves rather complex processes of reasoning. For building expert systems for designing, a detailed logical analysis of these processes is important and may appear inevitable. This kind of analysis has hardly been discussed in the literature; some papers even aim at discussing that 'design is logically impossible' (see [4]). However, it seems that the interest in theoretical and methodological aspects of design processes in general is growing (see [3], [5], [6], [7]). This paper aims to present a logical framework for describing the steps in a design process as logical inferences. To represent the various kinds of reasoning involved, this framework uses reasoning about viewpoints and reasoning about partial models. These are particular kinds of meta-level reasoning.

In [8] it is discussed how complex reasoning tasks for expert systems may be formally modelled by interactions of uniform reasoning tasks. Since design tasks often involve complex reasoning, the logical framework as presented here may offer a basis for a formal specification of expert systems for design.

1. Incomplete descriptions of objects by partial models

Constructing an object-description satisfying the design requirements is a process involving a number of steps. Each step fills out more details of the structure of the object. Also choices may be made and, if a choice turns out wrong, backtracking may occur. For describing traces of such a process, descriptions of structures are needed which may be given in different levels of detail. For this purpose we use *partial models* as known from logic (for instance, see [10], p. 68).

1.1 Partial models

We shall discuss partial models for propositional logic and for many-sorted predicate logic. When approppriate we use propositional descriptions. In this case a partial model is an assignment of truth-values **true** or **false** to a *subset* of the set of atoms occuring in the logical language; this may be viewed as a partial function from the set $\mathbf{A}$ of atoms into the set of truth values:

$$\mathbf{M : A \rightarrow \{true, false\}}$$

If we assume a certain order, say $\mathbf{A} = \{a_1, a_2, \ldots\}$ then a propositional partial model $\mathbf{M}$ may be represented by a sequence of symbols from **{true, false, *}** or **{0, 1, *}**.

$$< 1, 0, *, 1, 1, 0, *, *, 0, ... >$$

Here 1 in n -th position means the n -th atom is true, 0 that it is false and $*$ that it is (as yet) undecided. For finite cases this corresponds to a conjunction of atoms and negations of atoms

$$a_1 \wedge \neg a_2 \wedge a_4 \wedge a_5 \wedge \neg a_6 \wedge \neg a_9 \wedge \ldots$$

A partial model may be viewed as an incomplete description of the structure of an object. It is called *complete* if all atoms have a truth-value assigned to them, i.e. if no $*$ occurs. The *signature* of a partial model is the set of symbols of the language that is considered.

For the propositional example above this signature is the set $\mathbf{A}$. In the case of many-sorted predicate logic the signature is the set of all constants, function symbols and predicate symbols in the language under consideration.

1.2 Reducts and expansions

Partial models are closely related to reducts and expansions of (complete) models. Suppose L_2 is a signature extending the signature L_1 and $\mathbf{M_2}$ is a model of signature L_2. The *reduct* $\mathbf{M_1}$ of $\mathbf{M_2}$ is the model of signature L_1 obtained from $\mathbf{M_2}$ by deleting from it the part corresponding to symbols of L_2 which are not in L_1; the model $\mathbf{M_2}$ is called an *expansion* of $\mathbf{M_1}$. If, for instance, $\mathbf{M_2}$ is the complete model

$$a_1 \rightarrow 1$$
$$a_2 \rightarrow 0$$
$$a_3 \rightarrow 0$$
$$a_4 \rightarrow 1$$

of signature $\mathbf{A} = \{a_1, a_2, a_3, a_4\}$ then the reduct of $\mathbf{M_2}$ with respect to the restricted signature $\mathbf{A_0} = \{a_1, a_2, a_4\}$ is the model $\mathbf{M_1}$ given by

$$a_1 \rightarrow 1$$
$$a_2 \rightarrow 0$$
$$a_4 \rightarrow 1$$

Formally spoken this simply may be expressed by saying that the restriction of the function $\mathbf{M_2}$ to the set $\mathbf{A_0}$ is $\mathbf{M_1}$. This reduct $\mathbf{M_1}$ is a total function from the part $\mathbf{A_0}$ of the signature $\mathbf{A}$ to the set of truth values. Therefore $\mathbf{M_1}$ may be identified with a partial function from $\mathbf{A}$ to the set of truth values which is only defined for atoms in $\mathbf{A_0}$. Thus a reduct $\mathbf{M_1}$ of a complete model $\mathbf{M_2}$ of signature $\mathbf{A}$ may be identified with a certain kind of partial model of the same signature $\mathbf{A}$. In other words: with respect to the complete signature A the reduct M_1 may be viewed as a partial model; with respect to the reduced signature A_0 it is a complete model. In the propositional case every partial model may be obtained as a reduct of a complete model. In the case of many-sorted predicate logic partial models may occur where some functions or relations are only partially defined in the domain (see [10], ch 3.2, ch 5.4). These are not reducts of complete models. Below in fig 1 a simple example will be given; in this example partial models will be reducts.

Reducts and expansions of partial models may be defined as well. This may be done similar to the definition of reducts and expansions of complete models.

1.3 Refinement relation

Processes of incremental refinement in designing may be described by *incremental construction of partial models* which grow more and more complete. This provides a trace of partial models. Each of these partial models satisfies only a part of the design requirements and its structure is filled out only for a part of the logical language of the final model. This final model is a complete model of a certain signature expanding the signature of the design requirements and satisfying all these design requirements. To describe the steps of refinement in such a trace of partial models we define the *refinement relation* between partial models of arbitrary, possibly different signature: $M_1 \leq M_2$ if all atoms with a truth-assignment in M_1 also have a truth-assigment in M_2 with the same truth values (the truth-assignment of M_2 , considered as a partial function, is an extension of the truth-assigment of M_1). The refinement relation is reflexive and transitive. It may happen that for different partial models M_1 and M_2 we both have $M_1 \leq M_2$ and $M_2 \leq M_1$. We call such M_1 and M_2 *equivalent*; this determines an equivalence relation. Examples of equivalent partial models are provided by partial models of signatures S_1 and S_2 where all symbols not in the intersection of the signatures are undefined. This equivalence subsumes the identification between reducts of complete models and partial models as discussed in 1.2.

In design processes incremental refinement may often be viewed as introducing additional structure that cannot be described in the language of the requirements. For instance this is done by describing the object as a structure consisting of a number of independent components with some kind of interaction between them. In this case the refinement is created by introducing an expansion as described above. In the many cases that the design requirements do not explicitly prescribe the structures for such an expansion, the search for a useful expansion may be one of the aims of the design process. It may be a variable in the design process. The example in fig 1 below illustrates this.

1.4 A simple example

We give a simple, mathematical example; an example in a more realistic context will be given in section 4. The example below deals with finding (designing) a positive solution of a quadratic equation. Although this may be not the first issue considered in designing, it is useful to explain a number of our ideas. The design process proceeds as given in fig 1. Here **R** denotes the model of the real numbers with constants **0** and **1**, the operations of addition, subtraction, multiplication and the order relation; the language ***L*** is the language for this model extended by the constant **x**. The design process aims at constructing a model $< \mathbf{R}, \mathbf{c} >$ for a certain element **c** of **R** such that the initial design requirements (of **I**) are satisfied if the constant **x** is interpreted as **c**. Initially **c** is not given an interpretation (denoted by *****); we deal with a partial model. In **II** to this partial model some additional structure is added: the constant **y** is added to the language and will get an interpretation **d** later on (the second *****). This expanded partial model is filled in (in **III** and **IV**) to a complete model. Here two possibilities are involved (notice that the requirements placed between [] are left out of consideration temporary). Of these two possibilities only one satisfies (**V**) the requirement $\mathbf{x} \geq \mathbf{0}$. This remaining model satisfies all initial design requirements. The solution (in **VI**) is a reduct of this complete model: deleting **y** from the language and its interpretation **d** from the model.

Notice that during this design process both the partial models and the design requirements are changed dynamically: in **II** the requirement $\mathbf{x}^2 + 4\mathbf{x} - 5 = 0$ is transformed to $\mathbf{y}^2 - 9 = 0$. Also some new requirement $\mathbf{x} + 2 = \mathbf{y}$ is added to describe the connection between the model with **x** and the model with **y**. This dynamical behaviour of the design process essentially is the phenomenon we have to describe by our logical framework. Also notice that the expansion of the model by the constant **y** is not

design requirements and language	*models*
I. language: L	
$x \geq 0$	$< R, * >$
$x^2 + 4x - 5 = 0$	* not yet interpreted (constant x)
II. language: L + constant y	
$[x \geq 0]$	$< R, *, * >$
$x + 2 = y$	* not yet interpreted (x resp y)
$y^2 - 9 = 0$	
III. language: L + constant y	
$[x \geq 0]$	
$[x + 2 = y]$	
$y = 3$	$< R, *, 3 >$
or	or
$y = -3$	$< R, *, -3 >$
	* not yet interpreted (x)
IV. language: L + constant y	
$[x \geq 0]$	
$y = 3 \wedge x = 1$	$< R, 1, 3 >$
or	or
$y = -3 \wedge x = -5$	$< R, -5, -3 >$
V. language: L + constant y	
$y = 3 \wedge x = 1$	$< R, 1, 3 >$
VI. language: L	
$x = 1$	$< R, 1 >$

Fig 1 Solving an equation as a design process

determined by the initial requirements. To use just this expansion is a choice which is made in the design process; other choices could be made as well. This is part of the dynamical changing of the requirements.

By a slight change the example of fig 1 may be transformed into one involving functions. For instance, if the quadratic equation $x^2 + 4x - 1 = 0$ is chosen then the square root function may serve as a structure to express the solution. Another example is solving an inhomogeneous linear differential equation under certain boundary conditions. In this case the solution is some function. A common strategy splits this design problem into two other design problems: (a) determining all solutions of the homogeneous equation corresponding to the given equation without taking the boundary conditions into account, and (b) determining a particular solution of the given inhomogeneous equation, again without taking the boundary conditions into account. As a final solution a combination of these two kinds of solutions is constructed which does satisfy the boundary conditions. This strategy is used more often in designing (hierarchical decomposition; see below).

1.5 World of objects

The *world of (realizable) objects* will be denoted by $\mathbf{W}$; these are all those formal descriptions which give rise to objects which may be constructed in reality. For instance descriptions concerning objects with very small dimensions, which may exist in a theoretical sense but not in a material form, may be excluded, or one may restrict the choice of the design of a certain component of the object to a certain class of designs which are already known from the past. So it is important to notice that $\mathbf{W}$ does not contain all models which may be thought of in a theoretical sense. The formal descriptions in $\mathbf{W}$ are models according to the signature of the language of the design requirements, or extensions of that language. The class $\mathbf{W}$ serves as a domain from which possible solutions of a design process may be chosen. Since taking expansions is also a variable in the design process, $\mathbf{W}$ is multiform with respect to signature.

In many cases there exists some minimal language L which is interpreted in all these models (containing the language of the initial design requirements). Then the reducts of all models in $\mathbf{W}$ with respect to L may be taken; they form a class $\mathbf{W_0}$. In the example of fig 1 one may think of W_0 as the class of models of the form $< R, c >$ for all $c \in R$ and for W one may take W_0 extended by all models of the form $< R, c, d >$ for all $c,d \in R$. For the general case we do not assume that W_0 is a subclass of W; this means W may be not closed under taking reducts. By P we denote the class of all partial models involved in the design process; we assume $W \subset P$. In these partial models some symbols of the language of design requirements may have no interpretation, or only a partial interpretation. In the example of fig 1, for P we may take W extended by the partial models $< R, c, * >$, $< R, *, d >$ and $< R, *, * >$ for all $c, d \in R$.

One could think of the condition that every $M \in P$ has some refinement M^* in W, i.e. every partial model may be filled out and realized. However, to leave open the possibility of a negative outcome of the design process (asserting that there does not exist an object satifying the requirements) we de not assume this condition. This means we may reason with partial models of inconsistent theories.

1.6 General domain theory of the objects

The objects from W share certain general properties or laws, for instance physical laws. The knowledge about these is called the *(general) domain theory of the objects*. This theory is denoted by T. Every $M \in W$ is a (possibly partial) model of T, i.e. every expression from T which may be interpreted in M is true in M. We assume, moreover, that also every partial model from P is a model of T.

During the design process, T may be used to derive information on the partial model M at hand. For instance, if T contains a Horn proposition of the form $a \wedge b \rightarrow c$ and a and b are both true in M and c is not filled in, then M may be refined to a partial model in which also c is true. This again gives an assumption on P, namely that such a refinement indeed may be found inside P, unless there is an inconsistency. The case that inconsistencies occur will have to be considered more closely; at this moment we leave this open. In the example of fig 1 T consists of all the wellknown axioms for calculations with real numbers, such as associativity, etc. This general knowledge is used frequently during the design process. Notice that W may not contain all models of T; for instance W may be restricted to models built up from known components. One could try to express such restrictions in the form of knowledge in T, but this may be hard.

Summarizing, we have the following logical description:

$\boldsymbol{L}$	a language of a certain signature
T	a theory of objects in a language extending the language $\boldsymbol{L}$
W	a class of models of T of different signatures extending the signature of $\boldsymbol{L}$; these are descriptions of realizable objects
P	a class of partial models of T of different signatures extending the signature of $\boldsymbol{L}$; these are incomplete descriptions of objects being designed; W is a subclass of P.
W_0	the class of reducts of the models of W to the language $\boldsymbol{L}$

1.7 Logical descriptions of refinement steps

The process of refinement often plays a central role in design. Therefore we summarize the logical techniques which may be used to describe refinement steps:

- extending the language
  - . propositional logic: adding new atomic propositions
  - . predicate logic: adding new constants, functions, predicates

- extending a model
  - . propositionial logic: assigning truth values to more atoms
  - . predicate logic:
    - (a) extending existing, partially defined functions or predicates
    - (b) assigning (partially) interpretations to new constants, functions or predicates

The question may be raised whether or not the predicate-logical (model-theoretic) notion of model is adequate for representing the type of structures as used in design. In this paper this question is left open.

A number of special cases of refinement processes may be distinguished (for instance, see [7]); we mention:

- *incremental refinement across abstraction levels*

Here the domain is structured according to a number of abstraction levels; the design process proceeds by first determining solutions on the more abstract levels and then expanding these solutions incrementally to the more concrete levels. An example of this form may be found in section 4 below.

- *hierarchical decomposition*

In this case the object is built up from a number of components which may be designed separately (also called top-down design) and put into some interaction afterwards. Of this strategy too, an example may be found in section 4.

2. The theory of requirements; reasoning from and about it

The logical description of the design objects in various measures of detail as given above leaves open the role of the design requirements. The design requirements give an implicit description of the object being designed; this may be compared with an equation which gives an implicit description of some number. The aim of the design process is to use this implicit information to construct an explicit description of the object (solving an equation).

On the one hand the design requirements play the role of a logical theory for which a model is asked (for instance, see the example in fig 1). From this logical theory conclusions may be derived. On the other hand we also noticed that the set of requirements is changing dynamically during the design process. At some stages of the design process some other (sub) design problems are solved with different sets of requirements. Major design strategies are based upon some idea of splitting up the design problem into subproblems; each of these subproblems has its own set of design requirements. The sets of requirements of these subproblems are derived in some way or other from the initial design requirements (transforming them, choosing a suitable subset, etc.). The reasoning for such a derivation can only be described by considering the theories of requirements on a meta-level: one has to reason *about* theories. Sometimes such a form of reasoning is called *reasoning across viewpoints* (see [1]). We go into these two roles of the set of design requirements in some more detail.

2.1 The set of design requirements as a logical theory

To start with we consider a set of design requirements as a (finite) *logical theory* **R** which should be satisfied by the object being designed. In this sense the theory of design requirements supplements the domain theory **T** of objects as defined above. The design process succeeds if some explicit model of the theory $\mathbf{R} \cup \mathbf{T}$ has been constructed. The design process has a negative outcome if it is proved that such a model does not exist, i.e. the theory $\mathbf{R} \cup \mathbf{T}$ is inconsistent. The partial models used in the various stages of the design process are partial models of the theory of design requirements.

If during the design process a partial model **M** is considered, the theory **R** may be used to derive information about **M** ; this is similar to the use of **T** as described above. For instance, if **R** contains the disjunction $\mathbf{a} \vee \mathbf{b}$ and in **M** we know that **a** is false and **b** is undecided, then **M** may be refined to a partial model in which **b** is true. In the example of fig 1 such reasoning from the design requirements occurs to conclude from $x + 2 = y$ and $y = 3$ the statement $x = 1$. Therefore a refinement of the partial model may be created where x is given the value 1.

Below we first try to sketch a clear picture of the kind of reasoning that is intended in our framework (in 2.2). In 2.3 we study the role of the theory of design requirements in reasoning in some more detail.

2.2 The dynamics of reasoning

In the above we gave examples of reasoning from the domain theory of objects (in 1.6) and reasoning from the theory of design requirements (in 2.1). It may be noticed that this provides a *dynamical description* of reasoning as *making inferences* followed by *refining the partial model* at hand. This means that the result of the reasoning is not a true statement but a more detailed description of the object involved. According to this view the reasoning may be described dynamically by a trace of inference steps combined with a trace of partial models. If we permit the possibility of partial models which cannot be refined to a complete model in **W**, sometimes the refinement step in the reasoning may be impossible. In that case backtracking to an earlier choice point is required. In this sense the dynamical view of reasoning we suggest supports nonmonotonic reasoning. It may even be the case that the theory $\mathbf{R} \cup \mathbf{T}$ is inconsistent; in that case it only has partial models.

Reasoning from a fixed theory is not the only kind of reasoning that is done. As noted earlier the theory of design requirements changes dynamically during the design process. In fact, to utilize the implicit information embodied by the requirements two different types of reasoning are used:

- reasoning *from* the domain theory and the theory of design requirements;
  (combining knowledge from the theory with information from the partial model to new information which is used to refine the partial model)

- reasoning *about* the theory of requirements;
  (meta-reasoning where the requirements are used as objects to reason about, represented by terms in the meta-language and for which variables may be used)

The first of these two types of reasoning is described in sections 1.6 and 2.1 above. The latter type of reasoning will be illustrated below in section 2.3 by a number of specific forms. In section 2.4 we will discuss the interaction between the two types of reasoning.

2.3 Reasoning about the theory of design requirements

Four specific uses of reasoning about design requirements may be mentioned; this is not an exhaustive list:
- transforming the theory of requirements into an equivalent theory in the same language
- translating the theory of requirements into a theory in a different language
- first choosing a part of the theory of requirements
- hierarchical decomposition
- adding a specific requirement to the theory of requirements (making a choice)

We will discuss these below.

2.3.1 *Transforming into an equivalent theory*

The first specific type of reasoning we consider is the transformation of the theory of design requirements into a different but equivalent theory, i.e. a theory with the same class of models. Transforming a set of equations into an equivalent set, as occurs in some steps in fig 1 is an example of this. In these cases it is important to be able to describe the connection between the two theories and to reason with that connection. Therefore a meta-language is needed in which one can speak about theories, models, satisfaction and derivability (for instance see [2]). In the case of a set of requirements $\mathbf{R_1}$ which is transformed into an equivalent set $\mathbf{R_2}$ the connection between them may be expressed as:

$$\mathbf{true(M, T) \Rightarrow [\, true(M, R_1) \Leftrightarrow true(M, R_2)\,]}$$

Here the atomic meta-statement **true(M, A)** means that the model **M** is a model of the theory **A**. This is a semantical assertion which may be derived (using soundness) from provability relations like

$$\mathbf{provable(T \cup R_1, R_2) \wedge provable(T \cup R_2, R_1)}$$

One may reason with the meta-knowledge given above. For instance such meta-reasoning is needed to derive that a solution found for $\mathbf{R_2}$ is also a solution for $\mathbf{R_1}$. A knowledge based design system should be able to make such inferences.

2.3.2 *Translating into a theory in a different language*

It may happen that the theory of requirements is transformed into a theory in another language. In this case the connection between the two theories does not quite conform to the description in 2.3.1. For instance in the example in fig 1 the step from **I** to **II** consists in translating the theory $\mathbf{R_1} = \{x^2 + 4x - 5 = 0\}$ into the new theory $\mathbf{R_2} = \{y^2 - 9 = 0\}$ which is formulated in a different language. The model with **x** is translated into a model with **y**. Here the connection between the two theories may be expressed by taking the translation relation $\mathbf{I} = \{x + 2 = y\}$ as an additional condition:

$$\mathbf{true(M, T \cup I) \Rightarrow [\, true(M, R_1) \Leftrightarrow true(M, R_2)\,]}$$

A different way to express the connection is by considering the two extended theories $\mathbf{I} \cup \mathbf{R_1}$ and $\mathbf{I} \cup \mathbf{R_2}$. The connection between these extended theories may be expressed by

$$\mathbf{true(M, T) \Rightarrow [\, true(M, I \cup R_1) \Leftrightarrow true(M, I \cup R_2)\,]}$$

In both cases we deal with expansions with respect to the initial theory $\mathbf{R_1}$.

2.3.3 *First choosing a subset of the set of requirements*

In the example of fig 1 the condition $\mathbf{x \geq 0}$ has been left out for some time. This strategy is often used in design processes: first find a solution for a part of the set of requirements, then refine this solution to a solution for the complete set of requirements. In this case the theory of requirements $\mathbf{R}$ is split up into two new theories: the chosen subset $\mathbf{R_1}$ and the set of remaining requirements $\mathbf{R_2}$. Here the connection may be expressed as:

$$\mathbf{true(M, T) \Rightarrow [\, true(M, R) \Leftrightarrow [\, true(M, R_1) \wedge true(M, R_2)]\,]}$$

In all of the three specific types of reasoning about theories of design requirements considered until now, there turns out to be some form of equivalence. This is not the case for hierarchical decomposition, as described below.

2.3.4 *Hierarchical decomposition*

Often an object is designed by building it up from a number of components; these components may be built up from subcomponents, and so on. This is a wellknown strategy in designing called *hierarchical decomposition* (for instance see [7]). For each of the components a new theory of design requirements will have to be created. This new theory of requirements defines a new design problem, a subproblem of the initial design problem. The design process may proceed by solving this subproblem first (top-down strategy). Furthermore, an additional theory of design requirements will have to be created to describe the interactions between the components. Since the whole construction is intended to provide a solution for the initial theory of requirements there must exist some logical connections between all these theories. These legical connections may be described as follows.

Suppose **R** is the initial theory of design requirements and we decompose into two components with theories of requirements $\mathbf{R_1}$ en $\mathbf{R_2}$. Let **I** be the theory of requirements which describes the interaction between the components and let **M** be an arbitrary model, built up from the components $\mathbf{M_1}$ and $\mathbf{M_2}$. Then the logical connection between the theories may be expressed by

$$\mathbf{true(M, T)\ \&\ true(M_1, R_1)\ \&\ true(M_2, R_2)}$$
$$\mathbf{\&\ true((M, M_1, M_2), I) \Rightarrow true(M, R)}$$

Here the fourth conjunct expresses that in **M** the components $\mathbf{M_1}$ and $\mathbf{M_2}$ have an interaction which satifies **I**. This will have to be described more precisely. The semantical description above is implied by a provability relation of the form:

$$\mathbf{provable(R_1' \cup R_2' \cup I \cup T, R)}$$

Here the primes indicate that the requirements of $\mathbf{R_1}$ and $\mathbf{R_2}$ should be taken relative with respect to the component concerned.

The logical description of hierarchical decomposition, as well as the other specific transformations in 2.3.1 to 2.3.3, shows us that reasoning about theories (or viewpoints) plays a major role in design processes.

2.3.5 *Adding a specific requirement; generate-and test*

Sometimes the design process will proceed by making a choice for some specific aspect of the object being designed. In this case a subclass of the class of possible models is tried first. If this subclass does not contain a solution, backtracking is necessary to where the choice was made. The wellknown method called *generate-and-test* (for instance see [7]) makes use of such choices. A specific form is obtained if the additional requirement gives explicit information about the model.

From the logical point of view adding a specific requirement means creating a stronger theory which may be inconsistent (the case that backtracking will occur). As a logical peculiarity such an inconsistent theory is used not only to reason about but also to reason from. The logical connection between the two theories is a simple one:

$$\mathbf{true(M, T)} \Rightarrow [\, \mathbf{true(M, R_2)} \Rightarrow \mathbf{true(M, R_1)} \,]$$

As in the case of hierarchical decomposition there is no equivalence; therefore backtracking may be necessary.

2.4 The interaction between object-reasoning and meta-reasoning

In the sections 2.1 to 2.3 above we described independently the two types of reasoning involved in our logical framework: object-reasoning (reasoning from the domain theory and the theory of design requirements) and meta-reasoning (reasoning about theories of design requirements and partial models). In a design process these two types of reasoning are mixed to a complex pattern of reasoning. To give a complete logical description of such a complex pattern of reasoning it is important to give logical descriptions not only of the types of reasoning themselves but also of the interactions between them.

For a logical description of the interaction between the object-reasoning and the meta-reasoning we make use of *reflection principles* similar to [11]. These are transformations of object knowledge to meta-knowledge *(upward)* or vice versa *(downward)*. The conclusion $\mathbf{A}$ of a derivation from the domain theory and the theory of design requirements $\mathbf{T \cup R}$ using data from the partial model $\mathbf{M}$ is transformed by an upward reflection principle to the meta-datum that $\mathbf{A}$ is provable from $\mathbf{T \cup R}$ and data from $\mathbf{M}$. This reflection principle may be formulated as follows:

$$\frac{\mathbf{T \cup R \vdash_M A}}{\mathbf{provable(T \cup R, M, A)}}$$

A similar upward reflection principle may be formulated for partial models:

$$\frac{\mathbf{M} \models \mathbf{T} \cup \mathbf{R}}{\mathbf{true(M, T \cup R)}}$$

The upward reflection principles may be used to provide the meta-reasoning with meta-data. The downward reflection principles are the converse of the upward ones.

An example of the use of the reflection principles in the whole design process may be sketched as follows. First the theory of design requirements **R** is transformed to another one **R'** by meta-reasoning (for instance according to 2.3.1). Then a partial model is chosen for this transformed theory of design requirements. This partial model is refined to a complete model **M** by object reasoning. This fact is reflected to the meta-datum

$$\mathbf{true(M, T \cup R')}$$

By meta-reasoning it is derived that also

$$\mathbf{true(M, T \cup R)}$$

Finally, by the downward reflection principle it is concluded that **M** satisfies **T ∪ R.** Here object-reasoning may continue.

Notice that we did not go into the strategy of reasoning; we will say a word about that in the section below.

3. The design strategy

In sections 1 and 2 we sketched a number of logical connections between the theories and models before and after a step in the design process. These logical connections are described as logical inference steps on the object-level or on a meta-level or as upward or downward reflections. A complete survey of the steps which are possible and correct from the logical point of view is a *basic layer* of description; this layer still has to be described more exhaustively.

We may already say, however, that the search trees produced at this level of description will tend to be huge. To avoid exhaustive search an additional *strategic layer* is needed which contains knowledge about the inference strategy of the basic layer and can reason about the strategy. For instance in fig 1 for each of the steps it is not argued why it is chosen instead of the many alternatives that are possible. The strategic layer is implicit in this example.

The content of the strategic layer depends heavily on the kind of design process. In routine-design, where a fixed structure of the object is given, only the components have to be specified explicitly. In a more creative kind of design the structure itself is also left open. In [6] these kinds of design processes are described in some more detail. In the case of routine-design the strategy may be expressed by an explicit algorithmic description for the steps to be taken. This may be compared to the solving of an equation by someone who has learned a fixed algorithmic schema for it. The more creative case may be compared to someone who tries to solve an equation for the first time, without having been told how to do it. In this case a different more general problem solving strategy is needed.

In this paper we only mention the fact that such a strategic layer has to be worked out; we do not give suggestions for this. In [7] the two layers (basic and strategic) considered here are called 'knowledge level' and 'functional level'.

4. An example: making a floor plan for a house

This example is described in an informal manner in [7], p 136/137; we give a formalized description of a simplified form. Our intention is to show how the framework described in the earlier sections applies in more realistic situations. The example deals with a single-storey house of rectangular form. For example as in fig 2.

4.1 The logical language

In the logical language it should be possible to express connections between rooms such as 'room x shares a door with room y', or 'room x shares a door or a window with outer wall z'. Further one should be able to express that certain dimensions satisfy certain restrictions: 'wall x of room y has a length less than some value'.

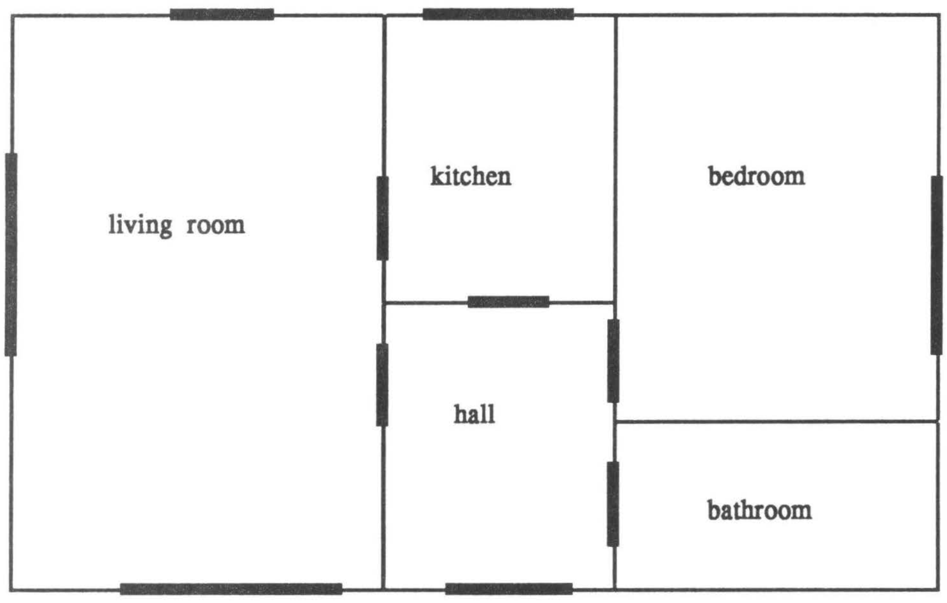


Fig 2 Floor plan of a house

To realize these expressivity requirements we define the logical language as follows. The objects to talk about are partitioned into the sorts **rooms, walls, corners, natural numbers.** There are constants to name specific rooms: **k** (kitchen), **b** (bathroom), **l** (living room), **s** (bedroom), **h** (hall), **w** (the whole house). Every room has four corners, defined by the functions **cor1, ..., cor4** from the sort of **rooms** to the sort of **corners.** The corners are numbered from left-front anti-clockwise to left-back; they have coordinates defined by the functions **x-co, y-co** from the sort of **corners** to the sort of **natural numbers** (taken in decimetres). Every room has four walls: front, back, left, right; the same holds for the whole house. These are defined by four functions from **rooms** to **walls**, namely **lw, rw, fw, bw**. Besides a predicate **wall** on **walls** x **rooms** may be used to denote that a certain wall is one of the walls of the given room. The symbol **length** denotes a function from **walls** to **natural numbers** (the length of the wall in decimetres) . The **natural numbers** have the usual ordening, addition and subtraction.

The predicate **door** on **walls** x **walls** is used to denote sharing a door, and the predicate **window** on **walls** x **walls** to denote sharing a window. Some function **overlap** is needed from **walls** x **walls** to **natural numbers** to denote the overlap of two walls. For a complete description some more functions and predicates may be needed. For instance predicates to express that two rooms are disjoint. For reasons of exposition we do not give all technical details for that; we confine ourselves to a subset of a complete specification. Summarizing, we define the following signature:

* sorts

    rooms, walls, corners, natural numbers

* constants

    k , b , l , s , h , w in **rooms**

* functions

    lw, rw, fw, bw : rooms → walls

    co1, co2, co3, co4 : rooms → corners

    x-co : corners → natural numbers

    y-co : corners → natural numbers

    length : walls → natural numbers

    +, – : natural numbers x **natural numbers → natural numbers**

    overlap : walls x **walls → natural numbers**

* predicates

    wall on **walls** x **rooms**

    ≤ on **natural numbers** x **natural numbers**

door on **walls x walls**

window on **walls x walls**

Next we describe the general domain knowledge, the theory of the objects.

4.2 Theory of objects

The theory T consists of the following conditions; these are the properties shared by all design objects. This list is not intended to be complete.

wall(fw(x), x)
wall(bw(x), x)
wall(lw(x), x)
wall(rw(x), x)
	Here it is summed up that front wall etc. of a room are specific cases of a wall of that room.

door(x, y) $\rightarrow$ **overlap(x, y)** ≥ 12 **window(x, y)** $\rightarrow$ **overlap(x, y)** ≥ 17	Doors and windows should have certain minimal dimensions.
length(x) ≥ 15	The walls should have certain minimal dimensions.

x-co(cor1(x)) = x-co(cor4(x))
x-co(cor2(x)) = x-co(cor3(x))
y-co(cor1(x)) = y-co(cor2(x))
y-co(cor3(x)) = y-co(cor4(x))
	The rooms are rectangles.

Additional knowledge should be added that the rooms do not overlap and together form the whole house. As mentioned earlier we do not go into the technical details.

4.3 Theory of design requirements

The design problem we consider may be described by the following design requirements; they form the theory **R**:

$\text{length}(lw(w)) \leq 70$ $\text{length}(fw(w)) \leq 100$	These are the maximal permitted dimensions for the house.
$\text{door}(lw(k), lw(w))$	The kitchen should have an outside door in its left wall.
$\text{door}(bw(w), bw(w))$	The living room should have a back door to the garden.
$\text{door}(fw(h), fw(w))$	The hall should have a front door.
$\text{window}(fw(w), fw(w))$	The living room should have a window in the front wall of the house.
$\exists\, x,y\ \text{wall}(x, s) \wedge \text{window}(x, y)$	The bedroom should have a window.
$\exists\, x,y\ \text{wall}(x, k) \wedge \text{wall}(y, w) \wedge \text{door}(x, y)$	The kitchen and the living should be connected by a door.
$\exists\, x,y\ \text{wall}(x, h) \wedge \text{wall}(y, l) \wedge \text{door}(x, y)$ $\exists\, x,y\ \text{wall}(x, h) \wedge \text{wall}(y, s) \wedge \text{door}(x, y)$ $\exists\, x,y\ \text{wall}(x, h) \wedge \text{wall}(y, b) \wedge \text{door}(x, y)$	The hall should be connected by a door with living, bedroom and bathroom.

If necessary, the existential quantifiers may be replaced by additional (Skolem) constants.

4.4 Two examples of design strategies and their logical description

In the floor plan example the following two design strategies may be useful (see also section 1): *incremental refinement using different abstraction levels* and *hierarchical* *decomposition.* We sketch a logical description for both.

4.4.1 *Incremental refinement using different abstraction levels*

This may be thought of as first solving the design problem in a qualitative, topological manner (connections between the rooms) and subsequently refining this qualitative solution to a quantitative, metric solution (the exact sizes).

From the logical point of view this means that first a restricted language is used, containing only qualitative expressions, i.e. all expressions containing numbers are deleted from the theory **R**. For the language given above this is realized by deleting the sort **natural numbers** and the operations on it and the functions **x-co, y-co, length** and **overlap**. For this reduced theory a model is constructed; this is a partial model for the theory **R**. Subsequently this partial model is refined to a complete model of the theory **R**.

4.4.2 *Hierarchical decomposition*

This strategy may be applied by splitting up the house in two main components (kitchen/living room and hall/bedroom/bathroom) and further splitting up these components into the different rooms, as in fig 3. A logical description of this strategy requires an expansion of the theories **T** and **R** to be able to express the hierarchy of fig 3 (for instance predicates for the two main components should be added and a predicate for the decomposing relation). A further logical description may be obtained by working out the general description given in section 2.3.4. By the axioms for the theory **I** for instance it has to be expressed that the size of the one component together with the size of the other component is just the size of the whole house (which is subject to certain restrictions).

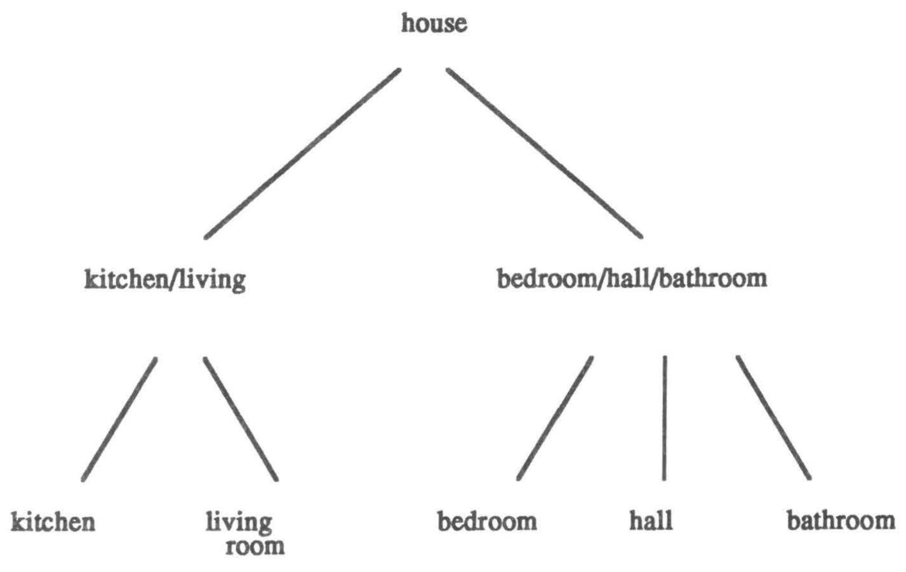


Fig 3 Hierarchical decomposition for designing a floor plan

5. Conclusions

In the above sections we sketched logical descriptions of the steps involved in a design process. The logical means used for these descriptions are the following:
- partial models to represent the object being designed on different levels of detail
- the domain theory of design objects; partial models are models of this theory
- the theories of design requirements; these theories vary during the design process
- reasoning from these theories, while making use of data from the partial model at hand
- reasoning about viewpoints (theories of design requirements), while making use of (meta)data from the viewpoints at hand

The use of partial models as proposed here is consistent with the use of partial models to represent changing situations in [9]. Reasoning from a given theory while making use of data from the model at hand is described in more detail in [7]. Reasoning about viewpoints and the interaction with other kinds of inference processes is similar to the descriptions studied in [8] concerning diagnostic processes and in [9] concerning process control.

It may be clear that representing the knowledge and the reasoning involved in a more realistic design process is not a simple issue. A number of aspects are not even considered above. For instance the strategy of making a choice for the dimensions of a certain room and then trying to adapt the other rooms is not considered. Some of these aspects are described in [7].

Acknowledgements

Earlier drafts of this paper have been read and commented upon by Pieter Geelen, Piet Rodenburg and Roel de Vrijer (Programming Research Group, University of Amsterdam). Also some discussions have taken place with Paul ten Hagen and Paul Veerkamp (III-CAD project, CWI, Amsterdam) and with Leo van Geffen and Lucienne Stomp (Engineering Design Laboratory, University of Twente).

References

1. Attardi, G. & Simi, M., Metalanguage and reasoning across viewpoints,
Proceedings ECAI 1984

2. Bowen, K. & Kowalski, R., Amalgamating language and meta-language
in: Clark, Tarnlund, Logic Programming, Academic Press, London, 1982

3. Mostow, J., Toward better models of the design process,
AI Magazine spring 1985, p 44-57

4. Ocathain, C.S., Why is design logically impossible ?,
Design Studies 3 (1982), p 123- 125

5. ten Hagen, P.J.W. & Tomiyama, T. (eds), Intelligent CAD systems I:
Theoretical and methodological aspects,
Springer Verlag, 1987

6. Tomiyama, T. & Yoshikawa, H., Extended general design theory,

Artificial Intelligence in Engineering 2 (1987), p 133 - 166

7. Tong, C., Toward an engineering science of knowledge-based design,
Artificial Intelligence in Engineering 2 (1987), p 133 - 166

8. Treur, J., On the use of reflection principles in modelling complex reasoning,
Report P8812, Programming Research Group, University of Amsterdam, 1988

9. Treur, J., Reasoning about partial models, actions and plans,
Report P8813, Programming Research Group, University of Amsterdam, 1988

10. Turner, R., Logics for Artificial Intelligence,
Ellis Horwood, Chichester, 1984

11. Weyhrauch, R.W., Prolegomena to a theory of mechanized formal reasoning,
Artificial Intelligence 13 (1980), p 133 - 170

K.D. Baker, L.J. Ball, P.F. Culverhouse,

I. Dennis, J.St.B.T. Evans, A.P. Jagodzinski

P.D. Pearce, D.G.C. Scothern and G.M. Venner

A psychologically based intelligent design aid

A PSYCHOLOGICALLY BASED INTELLIGENT DESIGN AID

By K.D. Baker*, L.J. Ball, P.F. Culverhouse, I. Dennis, J.St.B.T. Evans, A.P. Jagodzinski, P.D. Pearce, D.G.C. Scothern and G.M. Venner

Plymouth Polytechnic, Plymouth PL4 8AA, England

1. INTRODUCTION

The human creative process of electronic circuit design is a complex iterative activity drawing upon multiple sources of knowledge derived from prior experiences. The designer seeks to optimise the required functionality of an artifact within the constraints of the bounding external parameters. The early or conceptual stage of the process is dominated by the generation of ideas which are subsequently evaluated against general requirements' criteria. There follows a process whereby additional data are incorporated allowing decisions to be made between competing alternatives as more tangible evidence of function is derived. Once a final selection is made the design process becomes a routine transformation of function into the low level, circuit implementation detail.

Many, if not all, commercially available electronic CAD systems are merely sophisticated draughting tools that assist with this final stage of circuit level design. Few systems address the early stages of the design process where, to be a usefult adjoint to human performance, the tools must closely match the cognitive processes associated with the various design activities. An accurate psychological model of the human designer is therefore an essential prerequisite to any successful implementation of a computer based design assistant. Accordingly, the research reported involves a collaboration between a team of electronic engineers, computer scientists and cognitive psychologists with the aim of developing an intelligent software aid for the facilitation of engineering design. The work completed to date includes a psychological study of the cognitive processes involved in engineering design and the development of a prototype software system, within the Sun/ART environment, known as the Plymouth Engineer's Design Assistant (PEDA).

The PEDA system is intended to provide support for the higher levels and earlier stages of engineering design, in contrast to conventional CAD systems which provide low level tools for use at a stage when the major planning and structuring decisions have already been taken. Hence the system can be viewed at one level as an interactive decision aid which develops a model of the user and his/her problem and facilitates the formulation and assessment of alternative scenarios or versions of the design solution. The system is also intended to provide more specific and lower level aid in the form of direct simulations of design concepts and the inclusion (eventually) of relevant databases with intelligent search routines and rule bases.

The system we are developing is designed to take account of an understanding of the psychological characteristics of the engineering design process and any attendant cognitive limitations and weaknesses for which the system may compensate. Hence, our work is distinguished from most other approaches to engineering design aids in that it is firmly based upon psychological study of the mental processes involved. The cognitive psychologists in the group have applied their theoretical knowledge of the processes of thinking and decision making to a systematic study of the thought processes used by engineers. In view of the fundamental importance of the psychological analyses of design processes to the PEDA system, we start with a discussion of this aspect of the research.

2. PSYCHOLOGICAL ANALYSIS OF THE ENGINEERING DESIGN PROCESS

Psychologically oriented research on engineering design remains extremely scarce, and that which has been undertaken tends to have been narrowly focused on trivial and unrepresentative design tasks. Psychological research dealing with other design domains such as architectural and software design, while somewhat more plentiful, may clearly have limited applicability to the engineering domain. In contrast to the paucity of knowledge derived from rigorous psychological analyses of design, an abundant speculative literature - written by practitioners - is available that claims to shed light on design processes and that advocates methodologies and formal procedures which should be adhered to for the production of enhanced designs. It is felt, however, that such intuitive and conjectural information provides an inadequate basis for the development of interactive design environments, particularly in the light of abundant psychological evidence which suggests that people - experts included - have very poor self-knowledge of their higher cognitive processes (eg those involved in problem solving and decision making) despite belief that they know the strategies and procedures that they use (see Evans, in press, for detailed discussion).

* Present address Dept. of Computer Science, University of Reading, Reading, Berkshire, RG6 2AX, England

The formulation of our understanding of engineering design processes has therefore been based upon:

(i) review and assessment of the limited directly relevant literature referred to above,

(ii) consideration of general theoretical principles established in the wider psychological literature on thinking and problem solving, and

(iii) the conduct of our own empirical studies of cognitive processes in engineering design.

The psychological objectives of these studies have been:

(i) to discover the nature of any general design schemas or strategies utilised by engineers, and

(ii) to determine the nature of any cognitive limitations (eg working memory capacity, judgemental biases) which constrain the attainment of effective designs.

Such understanding of the underlying cognitive processes has direct implications for the development of an intelligent design aid that allows the engineer to design flexibly and naturally while at the same time helping to counteract cognitive failures (eg by problem structuring, reminding, advising and guiding). It should be emphasised that CAD systems derived from potentially erroneous, speculation-based models of design processes could conceivably hinder rather than assist a designer's creative work. Such hindrance could be caused, for example, by the system enforcing adherence to an unnecessarily rigid sequence of design activities or imposing added demands on cognitive resources at points when the designer's cognitive system is already strained.

One exploratory study of the design process that has been undertaken focused on eight semi-expert electronic engineers (final year undergraduates) tackling individual long-term design and development projects. All the engineers were pursuing projects within their favoured subdomain of electronic engineering and had previously undergone a period of placement in professional design settings. Regular structured diary entries and intermittent interviews were obtained from subjects over a period of several months during which the projects were run. This information therefore related to activity occurring throughout a design project - from the foremost stages of conceptual design to the final stages of realising a prototype in hardware. The protocols acquired in this study were subjected to detailed qualitative analysis with particular emphasis on the goals and subgoals which were motivating design activity. This analysis also reflected other features of the designers' behaviour, for example, the nature of evaluation processes such as those involved in assessing the viability of a solution concept or those used for selecting one design option from a set of alternatives. The question of what type of evaluation procedures are used by engineers in these latter kinds of situations is of clear psychological importance and has obvious relevance to the development of a design aid that can lend appropriate assistance.

The results of this protocol analysis indicated that subjects' initial processing effort was directed toward

(i) determining the functional requirements of the intended artifact (ie what the artifact was required to do)

(ii) detailing high-level constraints relating to the artifact's performance, resource usage and style (eg efficiency, reliability, testability, power consumption, speed and the like), and

(iii) defining constraints on the design process itself (eg budget, time allocation and equipment availability).

Subsequent to this initial phase of processing, subjects were seen to use a "problem reduction" strategy in their continuing design work. In the first instance this strategy led to a division of the overriding design problem into a collection of manageable and minimally interacting subproblems concerning the design of separate functional modules of the desired artifact. Once identified, the subproblems were then focused on in an essentially sequential manner with functional modules being developed depth-first to completion. In many cases subjects tended not to expend cognitive effort in the search for and modelling of substantially alternative (and potentially more optimal) design solutions. Instead they operated a "satisficing" principle, focusing selectively on a single satisfactory high-level solution concept rather than comparing alternatives with the aim of optimising choices. That is to say, subjects seemed to accept solutions that were "good enough" (eg cheap enough to implement or fast enough in operation) rather than looking for solutions that, even if not "best" (ie cheapest, fastest), were at least somewhat "better" than the first solution generated. It is possible, of course, that subjects were engaging in some form of rapid and covert - and perhaps therefore superficial - exploration and evaluation of alternative solutions. If this was the case, then it is possible that encouraging a more explicit mode of thought in which options are more thoroughly modelled and assessed could lead to enhanced design performance.

The notion of satisficing was introduced by Simon (1969) who advocated it as the basis of an acceptable procedure for finding a satisfactory design solution in the absence of a method for finding an optimum. Now while this argument is reasonable enough, it seems certain that in a competitive, profit-oriented design climate, designs that are merely satisfactory are rarely going to prove to be a truly cost-effective solution. Clearly, then, a design system that facilitates the search for and evaluation of alternative design concepts would be of much value to contemporary designers who generally are required to produce as optimal a design as possible within as short a time as possible.

In the present study it was additionally observed that whenever a designer selected a satisfactory high-level solution concept, this concept would often then be developed gradually to completion by the production of a variety of slightly improved versions. This suggests that the rudiments of an optimisation strategy was possessed by these engineers and emerged at lower levels in the design hierarchy. Considerable evidence was also derived in this study which indicated that subjects were constructing and manipulating mental models to simulate the dynamic behaviour of aspects of the developing design (cf de Kleer and Brown, 1983; Adelson & Soloway, 1986). Such modelling would seem to provide a potentially important means for comparison and evaluation of the viability of alternative solutions to a design problem or subproblem.

The main outcome of the study described has been the derivation of a theoretical model of the global processes that control and coordinate the designer's mental movement between subproblems and his/her development and evaluation of design solutions. This model incorporates the idea that a designer possesses generalisable abstract knowledge concerning how to produce good designs in the form of a "design schema" (cf Jeffries, Turner, Polson & Atwood, 1981). This high-level schema is built up through experience and can be applied to a wide range of domain problems having similar fundamental structures but which differ in terms of their content. When applied in a design situation the design schema controls the decomposition of the design problem into subproblems relating to the design of component functional modules. The schema then coordinates the order in which these subproblems are worked on and the search, retrieval and evaluation of solutions as well as initiating further subproblem decomposition if this is necessary (in which case the design schema is invoked recursively). The basis of the design schema is clearly the problem reduction strategy outlined above. Problem reduction is a general purpose, domain independent, problem solving method - one of the so-called weak methods of Newell and Simon (1972). When, however, the problem reduction method is augmented with procedures that ensure the effective retrieval and utilisation of domain specific technical knowledge, the outcome is a sophisticated higher-order knowledge structure that can function to produce fairly expert solutions to a wide range of design problems. In our theoretical model of design we have summarised the major processes of the design schema as a set of abstract production rules. A production rule representation of high-level design knowledge seems to provide a useful means to capture the flexibility of design activities as well as their fundamental structuredness.

To conclude the discussion of this study we would like to stress that it was very much an exploratory attempt to penetrate into a fairly unchartered area of psychological interest. In light of the inductive nature of the research we are aware of a definite need to be cautious in generalising the results to design situations involving (i) expertise levels different to those of the subjects studied and (ii) problems having characteristics different to those undertaken in the present case. A further problematic aspect of the study relates to the data collection methods that were adopted - in particular the use of cognitive diaries. Whilst this was the most practical solution we could think of for studying thought processes extended over many months, it clearly has its limitations. For example, subjects may sometimes have presented rationalised accounts of their behaviours or at other times may simply have forgotten to mention important design activities. Having voiced the need to treat the results of this study in a tentative manner we must emphasise that we remain particularly encouraged by the distinct commonalities that were observed in the design styles of the individuals studied who were engineers of varying ability pursuing designs in a range of technological subdomains.

In an attempt to further our understanding of engineering design processes we have recently performed another investigation, this time using a more sophisticated method for studying ongoing activities as well as focusing on engineers with a considerable level of design experience and expertise. This second study involved the collection of concurrent video-protocols of individual professional engineers "thinking aloud" as they attempted a small-scale design task in a laboratory setting. Whilst less realistic than the longitudinal study described above, this method affords the considerable advantage of providing a continuous and concurrent record of a specific act of thought permitting more accurate study of the underlying cognitive processes. The most powerful theoretical arguments concerning the value of think aloud protocols in cognitive research have been provided by Ericsson and Simon (1980, 1984). These authors argue that concurrent verbal reports provide details of the information attended to or heeded by the subject at any point in time in that they reflect the current contents of short-term memory (a term they use that is analogous to the notion of working memory).

The problem used in this study was intended to simulate a real-world design task and was therefore couched in terms of an informal "design specification". The specification expressed a requirement to design an integrated circuit in an image processing application. Certain of the circuit's functional requirements were detailed as mathematical formulae while other functional requirements necessitated that the engineer spend time in actually developing appropriate mathematical formulae. Clearly then, the problem was selected to address the nature of the design strategies employed in dealing with problems at highly abstract levels of the design hierarchy. All the engineers who participated in this investigation had expertise in working on integrated circuit design problems at these higher conceptual levels.

The verbal protocols acquired in the study provided the main starting point for various analyses while the visual protocols of the engineers' pen-and-paper work (eg sketching and note making) formed an important supplementary source of data. Transcribed protocols were each segmented into a sequence of verbal "chunks" using a detailed taxonomy of design behaviours that had been derived from a prior cursory analysis of protocol content. While analysis of the annotated protocols at the fine grained level of minute-by-minute behavioural transitions has yet to be undertaken, a more gross analysis of higher level activity patterns has confirmed some of the conclusions of the previous study. For example, subjects' design strategies were again leading to top-down decompositions of functional modules through levels of increasing concreteness. In the present case, however, the expansion of modules was generally seen to occur in a way that catered for their functional interdependency at each level, that is, a set of modules at one level was developed to a more concrete level in a sequential breadth-first manner.

The pursuit of substantially alternative high-level design concepts, again appeared not to be a pervasive aspect of these engineers' design methods as they tended to become selectively focused on satisfactory solutions rather than investigating other design options that may have proved more optimal. The gradual iterative improvement of selected design concepts through levels of increasing detail was, however, clearly evident. We realise that the inclination observed for these engineers to pursue a limited number of solution alternatives could have arisen as a consequence of the time pressure on them to derive designs within a restricted period of ninety minutes. Certainly, then, this particular manifestation of a satisficing principle should be interpreted tentatively - although when one considers that real-world design activities are usually heavily constrained by time factors, it may well be another specific instance of a more general tendency.

The data has also indicated that working memory limitations and attentional deficits were affecting the exhibition of expertise by these subjects and in some cases were leading to fairly significant inconsistencies and omissions. Such cognitive failures were seen to be particularly likely to occur when subjects were engaged in the evaluation and modelling of design alternatives. One interesting example was observed in the performance of an engineer who was engaged in elaborating (as a new functional block diagram) a previous solution model for the overall design problem (which again existed on paper as a block diagram). In this process of solution elaboration the engineer was seen to actually miss out two fundamentally important iterative loops that existed in his original diagram. In another case a clear example of inconsistency was provided by a designer who, within the space of several minutes, used the same mathematical notation (ie a theta symbol) to refer to two different design parameters. At a later point these two parameters were incorrectly being considered as one and the same, leading to a basic design mistake that was propagated through to lower levels. Interestingly, too, some subjects were also seen to formulate incorrect representations of information contained within the problem statement. Noteworthy is the point that these engineers were the ones who spent the briefest initial time in understanding the input, output and functional requirements of the circuit as specified in the problem statement, and were also least likely to refer back to the specification in later stages of their work.

When the results of our two empirical studies are considered together, we are impressed by the fact that the design strategies and cognitive limitations observed appear to generalise across very different tasks, design time-scales and levels of expertise. It still remains, however, to confirm the generality of these findings by further research. In this regard we suggest that one potentially valuable study would involve investigating the design activities of professional engineers tackling long-term design problems for real world applications. This kind of longitudinal investigation could provide more insight into the dynamics of the evolution of design solutions in professional settings and could give a clearer indication of the nature of the design strategies adopted by experienced engineers.

Our findings derived from these psychological analyses of engineers' design processes have suggested certain facilities that an intelligent design system could offer. In general, these facilities would be geared toward assisting with the search for alternative and potentially more optimal design solutions as well as toward reducing the time of the enginering design process. More specifically they would:

(i) encourage the designer to consider an increased number of initial high-level solution concepts and enable the efficient formulation of alternative versions of each solution concept through levels of increasing design detail,

(ii) assist with the choice of competing design solutions, for example, enabling evaluations of solutions to be made on the basis of comparative functional simulations,

(iii) superintend the designer's exploratory activity, for example, helping the designer to backtrack if a path proves unpromising (ie by providing a record of paths taken together with the current point of exploration) or suggesting worthwhile paths of investigation (ie by suggesting design alternatives),

(iv) ensure the designer's awareness of design conflicts (eg if crucially important constraint requirements have been overlooked when the designer is focusing on a narrow aspect of the overall design solution),

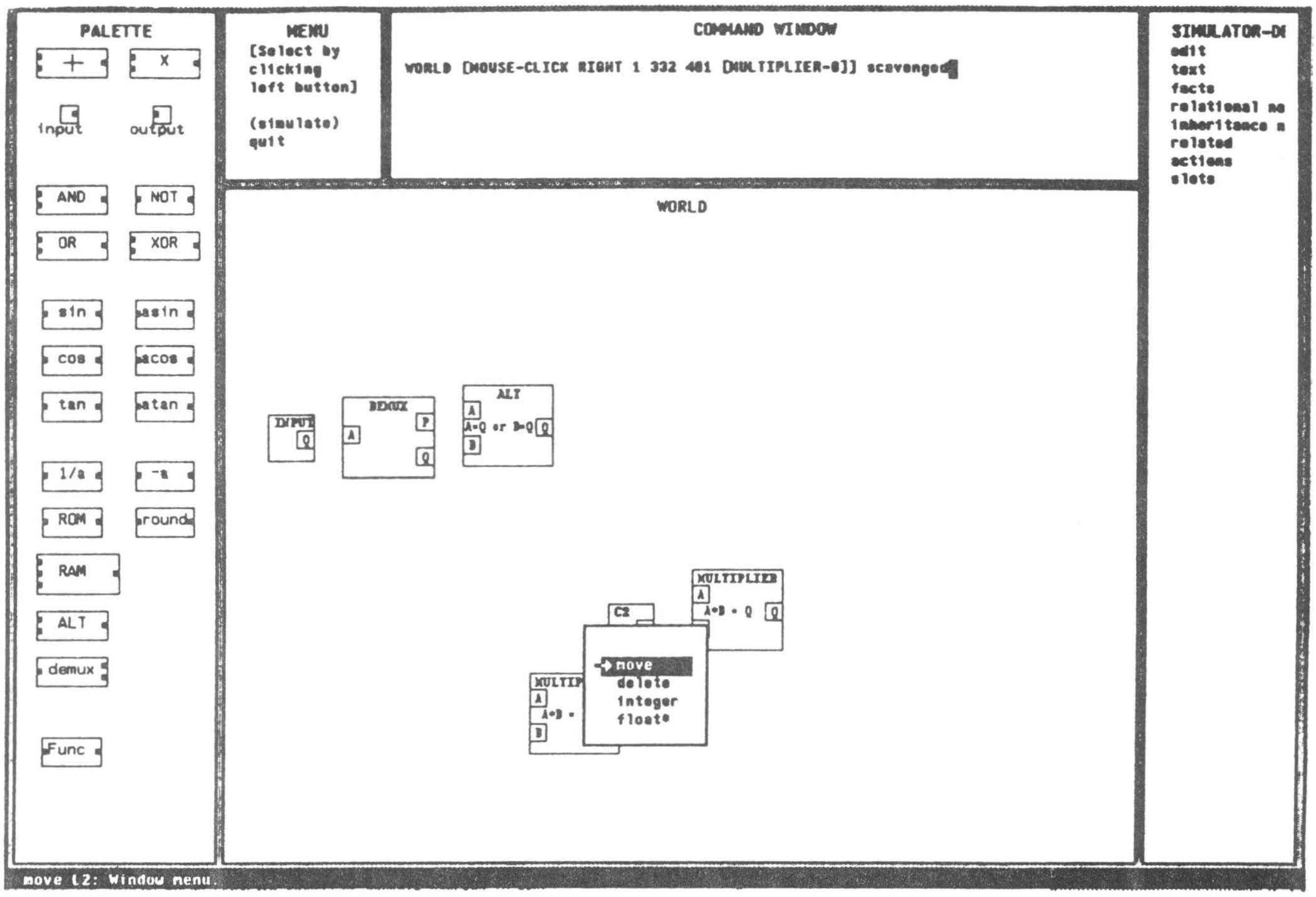
(v) ensure the designer's awareness of inconsistencies in the notation that is being used (eg if two different design parameters have been given the same symbolic label).

3. OVERVIEW OF THE PEDA SYSTEM

3.1. Overall form and function

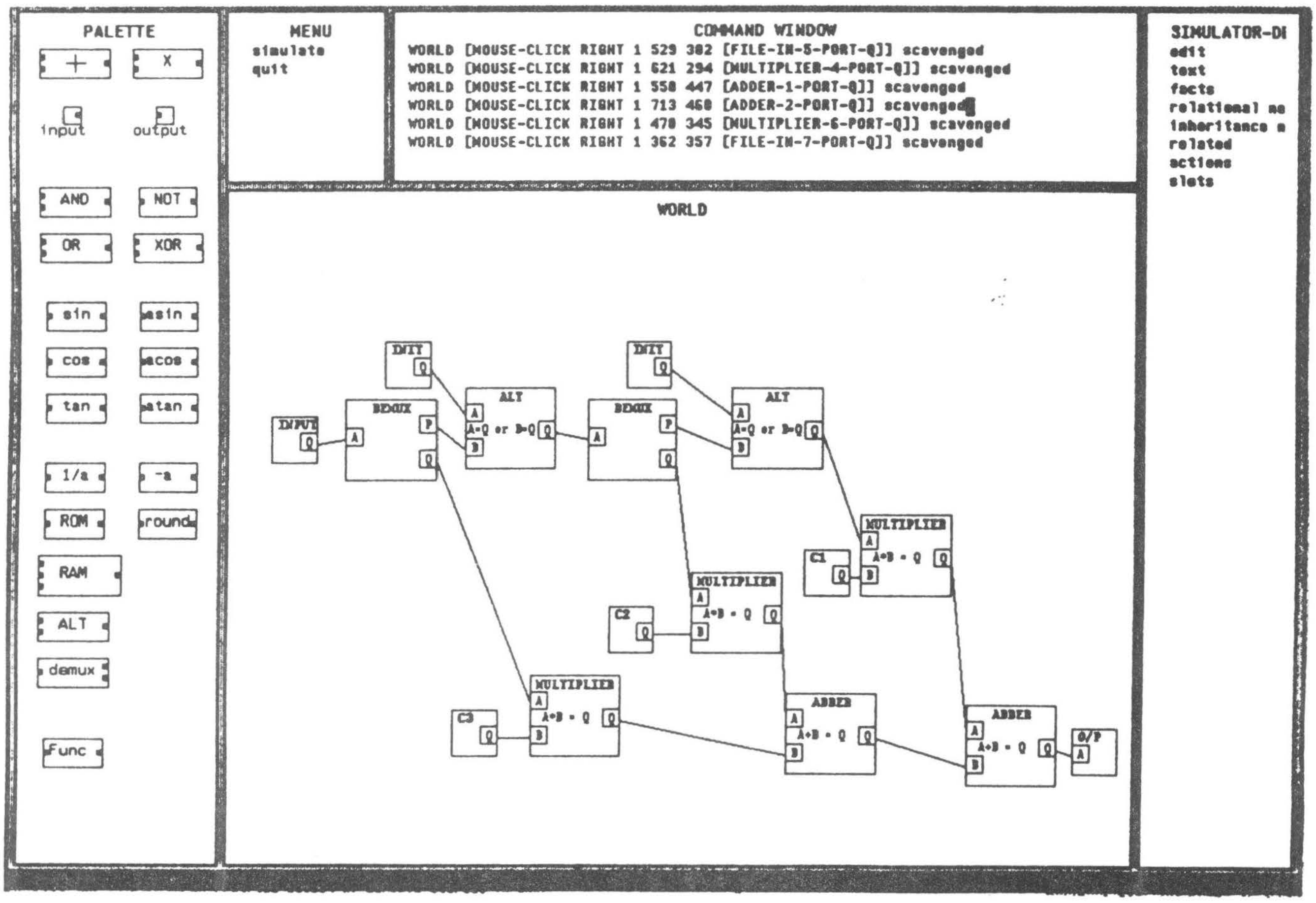
The form and function of PEDA have been derived from the psychological study with the objective of enhancing the performance of the design engineer without inhibiting his/her natural style of working.

The system is presented to the user as a screen-based drawing board providing mouse-driven direct-manipulation of block diagrams, which are the preferred notation of electronic engineers. Blocks are selected from a palette and represent any level of the functional decomposition of the design from user-specified mathematical functions down to individual electronic components, such as ROM or RAM.



```
WORLD [MOUSE-CLICK RIGHT 1 332 481 [MULTIPLIER-8]] scavenged
```

Figure 1: diagram construction



```
WORLD [MOUSE-CLICK RIGHT 1 529 382 [FILE-IN-5-PORT-Q]] scavenged
WORLD [MOUSE-CLICK RIGHT 1 621 294 [MULTIPLIER-4-PORT-Q]] scavenged
WORLD [MOUSE-CLICK RIGHT 1 558 447 [ADDER-1-PORT-Q]] scavenged
WORLD [MOUSE-CLICK RIGHT 1 713 460 [ADDER-2-PORT-Q]] scavenged
WORLD [MOUSE-CLICK RIGHT 1 470 345 [MULTIPLIER-6-PORT-Q]] scavenged
WORLD [MOUSE-CLICK RIGHT 1 362 357 [FILE-IN-7-PORT-Q]] scavenged
```

Figure 2: completed diagram

A typical palette appears on the left hand side of figures 1 to 3. The largest window in figure 1 shows the construction of a diagram to represent the operation of a FIR filter. The popped-up menu displayed on the lowest multiplier shows the options

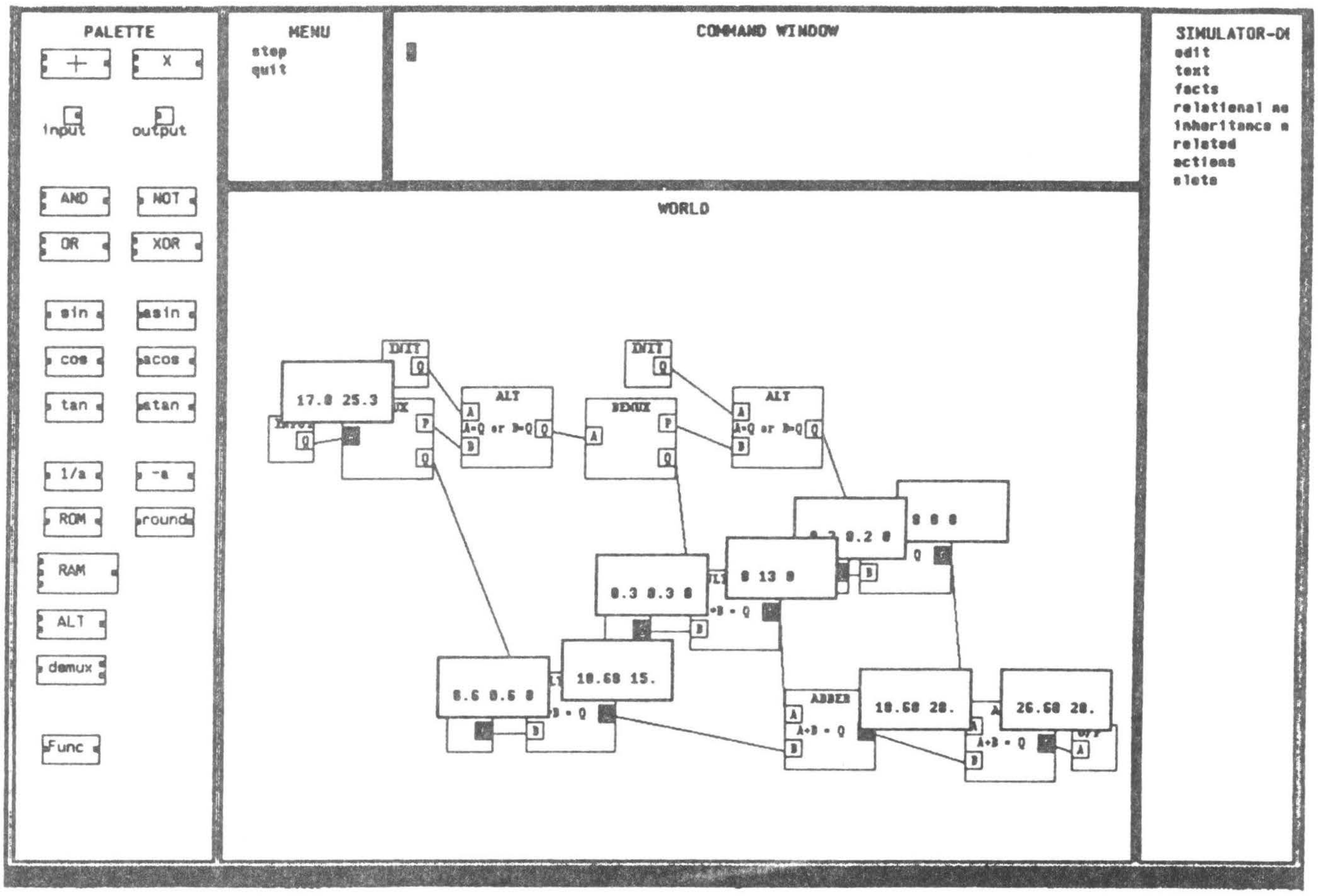


Figure 3: simulation results

available to the designer for manipulating that block. Figure 2 shows the diagram completed, with connectivity between components expressed as lines linking communicating parts.

The functionality and connectivity represented by the user interface of PEDA is modelled by an underlying functional layer which simulates the actions of individual design objects and operation of the connected whole. The user is able to probe intermediate results and values by opening windows at points to evaluate the dynamic behaviour of a design. Figure 3, for example, shows the effect of sending data through the FIR filter, with intermediate results displayed at different points.

PEDA supports the hierarchical nature of design by permitting several levels of function and the facility to zoom in on high level design objects to manipulate their constituent functions or components. For example, the filter shown in figure 3 may be expressed as a single block, with the same overall functionality, in a diagram at a higher level of design.

Optimal, rather than just satisfactory, designs are encouraged by the facility to explore, develop and store many alternative designs in parallel as inter-related versions. The PEDA also has the ability to monitor the evolution of such versions and provide advice on the actual process of design, for example reminders of incomplete activities, a record of crucial assumptions and constraints, and a note of instances when a particular solution has already been explored elsewhere. The system also facilitates backtracking through the design process by means of the logging of design activity.

Knowledge-based functions will eventually enable the validation and optimisation of designs using a data-base of component characteristics in conjunction with a rule-base of electronic design heuristics.

3.2 Implementation system

PEDA is implemented on a Sun 3/60 workstation using ART. The main components of the ART language are facts, schemata, actions, rules and viewpoints. The ART compiler maps the language into LISP and integrates it into the LISP environment. Facts are separate pieces of declarative knowledge. Schemata are used to organise knowledge about objects that are related to each other. A schema is a collection of facts that represent an object or a class of objects that share certain properties. Actions provide a means of attaching procedural code to objects (ie schemata or their attribute slots) so that elements of a design can be modelled in terms of their characteristics and behaviour in an object-oriented style. Rules may use forward or backward chaining (ie from facts or from goals). There is a powerful pattern-matching language to describe the data that drives the procedures. ART Viewpoints provide a means of modelling hypothetical alternatives and situations that change dynamically with time. It is recognised as one of ART's most sophisticated and powerful features and is particularly useful for the manipulation of related but distinct versions of the users' designs.

This feature of ART would be still more useful if it allowed alternative versions to be stored within. As it stands, additional schemata have to be created to enable versions of designs to be stored for future reference.

4. FUNCTIONAL ASPECTS OF THE PEDA SYSTEM

4.1. Design Simulation

The simulator provides the basic underlying mechanism by which data is moved around a block diagram. A block diagram consists of a series of hierarchically structured blocks which have been interconnected in such a way to provide a specified function. Within the block diagram data exists as objects called packets which move between the blocks via links between them. When the functional input requirements to a block are satisfied, the functionality of the block is applied to the input data packets, destroying them in the process. After evaluation, new packets are created at the block's outputs as necessary. Block functionality is expressed as a set of mathematical equations, as this affords a greater deal of flexibility than implementations using a fixed level of functionality, such as logic gates. A library of common low level functions is available - however the important distinction here is that blocks need not be built up from these primitives. An example would be the generation of a block with the function $Q= \sin(A*A) + \cos(B*B)$. The primitive function approach would involve creating an equivalent block from a collection of primitives. The mathematical equation approach, however, has sufficient flexibility to use the function as it is.

The simulator architecture is based on a data flow model and as such there is no explicit representation of time. This choice is in accord with the results of the psychology study, indicating that exact timing considerations are not a necessary part of the early design process, as the engineer derives form and function from the project specification. This type of approach has two main advantages, firstly time information can be treated the same as any other data and fed into a block diagram, and secondly the model maps very well onto an implementation scheme using production rules in ART.

4.2. Design Process Record and Versions

The psychological studies suggest agreement with the general consensus that designs evolve along a number of dimensions. Walker and Thomas (1985) describe such a model for design representation in that a design can be viewed as being hierarchically structured with the greatest degree of abstraction at the top. This links in with the fact that in electronic design, engineers have been observed to describe the functionality of a design in a top-down manner. As they descend into the hierarchy the description at each level becomes more specific until, at the bottom level, actual components are specified. Walker and Thomas subdivide their representation into three subdomains - Behavioral, Structural and Physical - and categorise each element of the design into its respective area. The psychological studies, however, indicate that a simple representation containing both functional and constraint requirements together with the actual block attributes at the block diagram level is quite adequate for design representation in the early stages of the design process. Within this context a functional requirement is the most general statement of what a block must do, for example, "an adder must add". Constraint requirements limit the range of possible solutions, for example, "the adder must be fast", whilst attributes describe what a block can actually do, for example, "the maximum propagation delay for this adder is 100ns". Within the PEDA a block diagram can be considered a hybrid structure consisting of elements from all three subdomains - but at a relatively high level in their hierarchy. Blocks can then be represented as schemata with slots for attributes and constraint requirements, with attributes being generally inherited along class lines, and constraint requirements propagated down from higher levels in the block hierarchy (for an example see figure 4).

The psychological studies have shown that engineers tend to move between subproblems in a hierarchy of problem spaces which transcend abstraction levels. Within these levels the principle of "separation of concerns" is applied to a limited degree. In its pure form this implies that notation and parameter definitions within a level relate specifically to design considerations within that level and rarely between other level. In actuality propagation of constraint requirements does occur between levels, but the information tends to be symbolic and quite abstract in nature, such as, "the required chip area for the function square root is large".

Additional evidence has indicated that the pursuing of alternatives within a design is in general suboptimal. When exploring alternatives engineers tend to lock onto one "satisficing" solution, instead of adequately pursuing a range of reasonable

```
(defschema design-1
  (instance-of world) ; a design alternative
  (must-have (speed fast))      ; stated constraint requirement
  (must-have (chip-area large)) ; stated constraint requirement
  (must-have (power medium))    ; stated constraint requirement
  (must-have (design-time medium)) ; stated constraint requirement
  (design-time low) ; attribute inferred from: integer-multiplier-blocks
  (chip-area large) ; attribute inferred from: integer-multiplier-blocks
  (speed fast)      ; attribute inferred from: integer-multiplier-blocks
  (power medium)    ; attribute inferred from: integer-multiplier-blocks
  (contains-blocks mult-block-1 ..... mult-block-3) ; contains 3 multiplier blocks
)
(defschema integer-multiplier-block-1
  (contains-blocks mult-1 ..... mult-16) ; contains 16 multipliers
  (result-data-type integer) ; attribute inherited from: 8-bit-multiplier
  (bits 32)
  (design-time low) ; attribute inferred from: 8-bit-multipliers
  (chip-area large) ; attribute inferred from: 8-bit-multipliers
  (speed fast)      ; attribute inferred from: 8-bit-multipliers
  (power medium)    ; attribute inferred from: 8-bit-multipliers
)
(defschema 8-bit-multiplier
  (instance-of process-x-block)
  (instance-of integer-multiplier)
  (result-data-type integer)    ; attribute inherited from: integer-multiplier
  (function ((setq Q (* A B)))) ; attribute inherited from: integer-multiplier
  (bits 8)                ; stated attribute
  (design-time very-low)  ; attribute inherited from: process-x-block
  (chip-area medium)      ; stated attribute
  (speed fast)            ; stated attribute
  (power low)             ; stated attribute
  (has-instances mult-1 ..... mult-16)
)
(defschema integer-multiplier
  (result-data-type integer)
  (instance-of multiplier)
  (function ((setq Q (* A B)))) ; attribute inherited from: multiplier
)
(defschema process-x-block
  (design-time very-low) ; a standard cell
  (technology process-x)
)
(defschema multiplier
  (function ((setq Q (* A B))))
)
```

Figure 4: example of a block hierarchy showing blocks represented as schemata

alternatives - presumably this is due to time constraints or experiential biases. PEDA supports the switching of attention between alternatives by maintaining several concurrent versions of any part or all of the design. It is hoped that this environment will partly achieve the desired goal of increasing the number of alternatives that are considered in the design process. Additionally the versional structure within the assistant is used to provide the skeleton upon which the tool builds up a model of the user design process. This is in accord with Mostow (1985) in that the design can be used as a basis of the design process model. At a simple level the versional structure is used to hold the individual block diagrams that the engineer has been working on. At a higher level additional information is stored regarding the engineering constraint requirements, functional requirements and assumptions used in the formation of a particular version, together with a history of events. As a design evolves multiple versions of the design will be created containing decisions and criteria which may lead to different final designs. These designs will be interrelated, sharing many hierarchical subcomponents and possibly whole levels. In this way the evolution of the design may be regarded as one of the fundamental dimensions of the design process. A versional structure is outlined in figure 5.

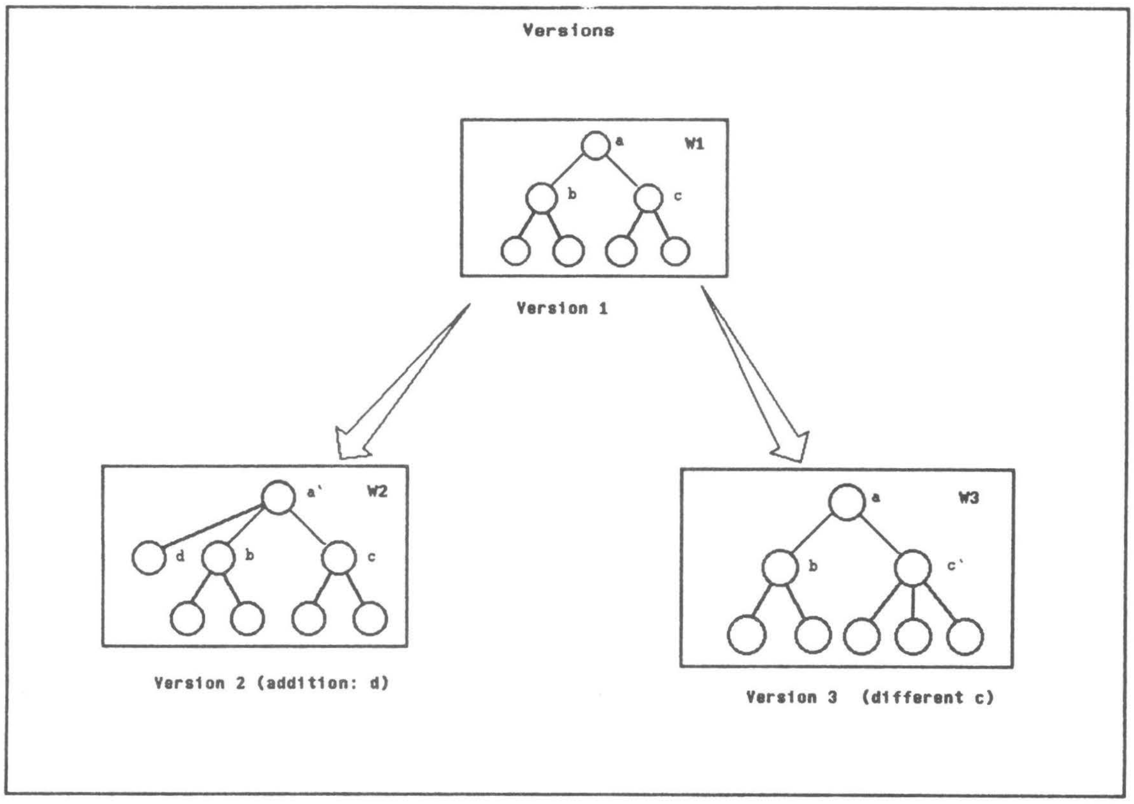


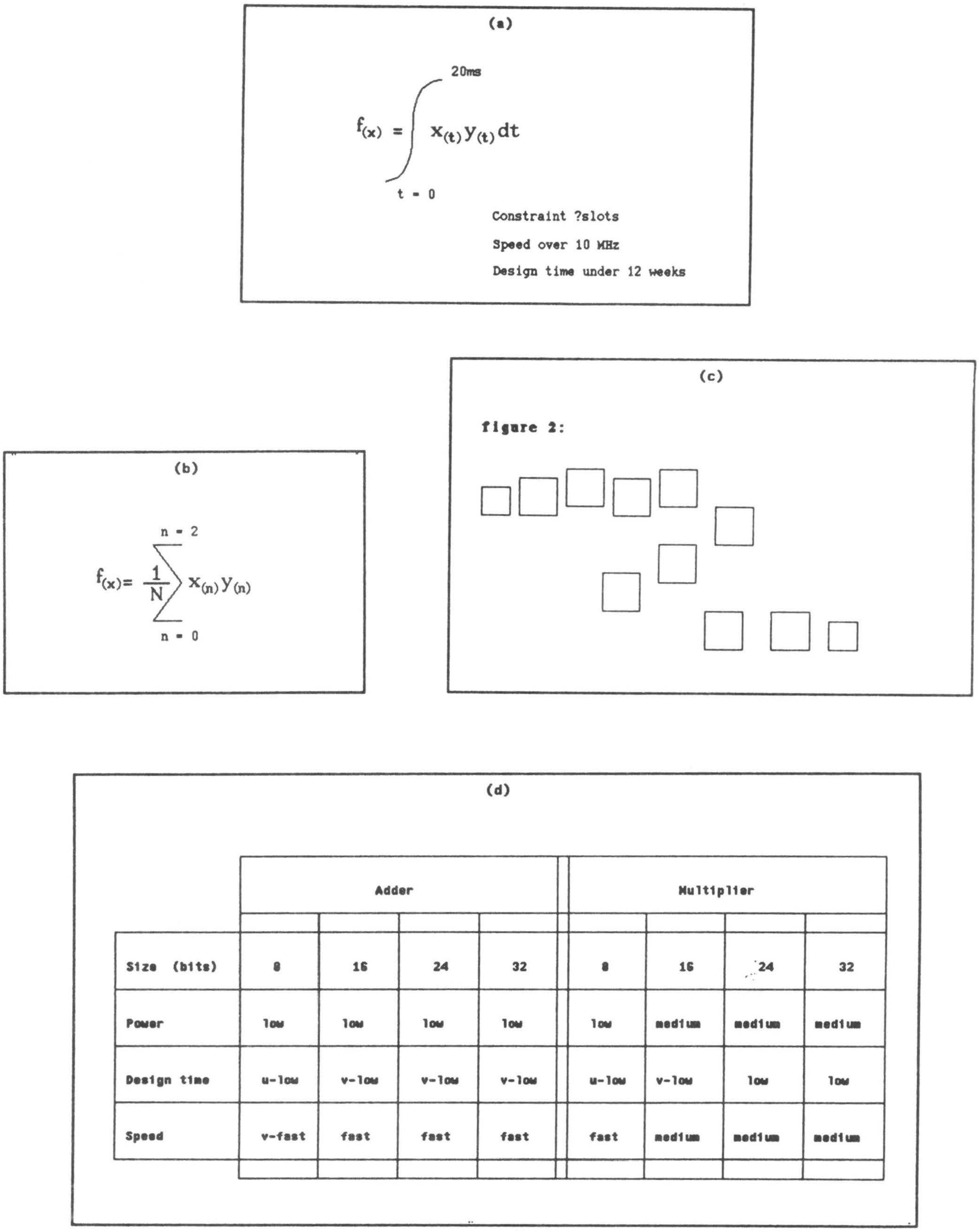
Figure 5: outline of a versional structure depicting interrelated designs

4.3. Monitoring

A monitor module within the PEDA is used to convert the interaction between the user and the system into a design trace. This is essentially a record of all the activity since the working session started and exists primarily to prevent the loss of information due to changes in the design. The trace is converted into a design history using an expert system approach to identify and abstract only those parts of the design trace that are considered important to the decision process.

4.4. Design Heuristics

Again an expert system approach has been adopted - here to offer advice upon design choices. Rules are derived from both an examination of the literature, especially in the area of decision making in electronic design and a study of designs done within



$$f_{(x)} = \int_{t=0}^{20ms} x_{(t)} y_{(t)} \, dt$$

Constraint ?slots

Speed over 10 MHz

Design time under 12 weeks

$$f_{(x)} = \frac{1}{N} \sum_{n=0}^{n=2} x_{(n)} y_{(n)}$$

	Adder				Multiplier			
Size (bits)	8	16	24	32	8	16	24	32
Power	low	low	low	low	low	medium	medium	medium
Design time	u-low	v-low	v-low	v-low	u-low	v-low	low	low
Speed	v-fast	fast	fast	fast	fast	medium	medium	medium

Figure 6: application of design heuristics in the evolution of a hardware design version from a customer specification (see text for details)

the group, with possible additions suggested by the results of the psychological study. The current heuristics set involve various costing algorithms in design that allow comparative optimisation and validation of designs. These heuristics address simple concepts in both symbolic and actual terms on constraint requirements such as power, chip area and rough speed estimates (see figure 6 for an example).

Figure 6(a) shows what might be a typical mathematical expression declared as part of the customer specification. This would be placed as seen into the system and simulated using an algebraic manipulation package integrated into PEDA. The design engineer developing the project may be specifically exploring a digital implementation and be concerned with the minimal accuracy the user can tolerate. To this end figure 6(b) might be a discrete time version of figure 6(a). Currently to evaluate different versions of this, PEDA supports arbitrary accuracy functional blocks. Thus mapping figure 6(b) to a hardware version in figure 6(c) allows the engineer to verify his developments by simulation and subsequent compararison to figure 6(a). By selecting different simulation accuracies for each block in figure 6(c), several versions of the functional block may be created. For example, using multipliers with 8 bit integer accuracy and 24 bit adder/accumulator accuracy may be a version. There may be many such versions, utilising all possible combinations of figure 6(d). The accompanying engineering constraints are available instantly for the whole functional block, for each version, allowing comparisons to be made quickly by the engineer.

Ultimately PEDA will allow similar rules of thumb to be made directly on mathematical expressions. The data in figure 6(d) will ultimately be derived via heuristics from real circuit performance characteristics. At the moment PEDA uses a small test set of data in the area of VLSI design - although in reality the standard libraries would apply to a range of specific technologies and include, for example, data on 1.5um CMOS processing. In outline the heuristics are used to compare constraint requirements at a level with a corresponding collective attribute derived from the attributes of the blocks at that level and below it in the block hierarchy. Each versional structure has a corollary in a created ART viewpoint or world which is used by the heuristics to reason about that specific version, with merged viewpoints catering for cross versional inferences. Additional rules act upon the design history and will be able to offer the engineer the reasons for a particular design decision, including the decision path up from the current to the base design. Later, PEDA will offer advice regarding the design strategy adopted by the engineer by comparing the user's design process with an idealised model derived from the psychological study.

5. The User Interface

5.1 Underlying principles

The design of the user interface is based not only on the results of the associated psychological study, but also on psychological work beyond the project. The objectives are to reduce the cognitive load borne by the user and to enhance the human talent being brought to the task of designing. The approach takes account of the fact that humans and machines have complementary strengths and weaknesses; for example, humans are creative but forgetful. A well-designed user interface is particularly important in a co-operative system, since successful co-operation between the user and the machine depends on good communication. The discussion of the user interface will be divided into three parts: the basic interface to the simulator, the representation of knowledge about the instantiated design, and the representation of the design process.

PEDA has a direct manipulation interface in order to reduce the gulf between the user and the system during execution and evaluation, and to give the user a feeling of direct engagement (Hutchins et al, 1985). The system naturally uses a drawing board, rather than the more familiar desktop, metaphor. The user creates his diagram using the mouse to select objects from a palette. Thereafter, a pop-up menu associated with a particular instantiated object presents the user with the choice of relevant, valid functions that can be performed on it. An example is the identification of those ports that the user wishes to probe during a simulation. Figures 1-3 illustrate three stages in the use of an early version of the system to simulate a FIR filter, namely: diagram construction (showing a pull-down menu), the completed diagram, and the results of a simulation.

The operation of the interface, described in 3.1, is designed to reinforce the users ability to hold a consistent conceptual model across all of the system's prototype will investigate the success of these and alternative mechanisms.

In addition to the basic facilities of diagram construction and simulation display, the user interface is concerned with the representation of more abstract concepts, namely: design knowledge and the evolution of the design.

5.2 Communication of design knowledge to the user

Design knowledge embedded in the system (see 4.3) enables it to perform heuristics such as optimisation and validation, which must be represented in a meaningful way. The requirement is somewhat analogous to the provision of explanations by expert systems, which are notoriously hard to achieve satisfactorily (Berry & Broadbent, 1987). A highly graphical approach can be more appropriate in an ICAD system. For example, tradeoff criteria, such as size versus speed, naturally lend themselves to a graphical rather than a textual display.

Alternative methods of representing and explaining design knowledge and advice will also be explored with a number of prototyping studies.

5.3 Communication of the evolution of the design process to the user

The representation of the evolution of the design process is an even more difficult problem, requiring the interface to display an abstract slice through a concrete process, as the system attempts to derive a goal-orientated explanation of the progress of an actual design session.

A simple sequential trace of the session may in some circumstances be useful, for example as an aid to memory. However, the designer is also likely to want other views such as a representation of the "main road" from goals to final design, uncluttered by side-turnings, diversions and dead-ends.

Theory suggests that a pictorial representation of the version tree, showing the relationships between the different versions of the design that the user has explored, will be particularly useful if it highlights such things as constraints, assumptions, conflicts and changes, and thus enables the user to keep track of what he/she is trying to do.

Techniques similar to those used for program visualisation (Myers, 1986) will be evaluated in the prototype.

6. CONCLUSIONS

The system described in this paper is distinguished by its basis in a sound psychological study of the design processses used by professional engineers. The psychological work has identified weakness inherent in design practice - in particular, failures to consider sufficient alternative versions to provide optimal design choices - which the system is being designed to correct. In addition, the system addresses the observed need for effective simulation as part of the design process at all levels of the hierarchy and demonstrates the advantages of a direct manipulation interface in this type of application.

7. REFERENCES

Berry, D.C. & Broadbent, D.E. (1987). Expert Systems and the Man-Machine Interface. Expert Systems,Vol 4, No. 1, 18-27.

de Kleer, J. & Brown, J.S. (1983). Assumptions and Ambiguities in Mechanistic Mental Models. In D Gentner & A.L Stevens (eds), Mental Models. Hillsdale, NJ.: Lawrence Erlbaum Associates.

Ericsson, K.A. & Simon, H.A. (1980). Verbal reports as data. Psychological Review, 87, 215-251.

Ericsson, K.A. & Simon, H.A. (1984). Protocol Analysis: Verbal Reports as Data. Cambridge, Mass.: MIT Press.

Evans, J.St.B.T. (in press). Bias in Human Reasoning: Causes and Consequences. Brighton: Erlbaum.

Hutchins, E.L., Hollan, J.D. & Norman, D.A. (1985). Direct Manipulation Interfaces. Human-Computer Interaction, Vol 1, 311-338.

Jeffries, R., Turner, A.A., Polson, P.G. & Atwood, M.E. (1981). The Processes Involved in Designing Software. In J.R. Anderson (ed), Cognitive Skills and Their Acquisition. Hillsdale, NJ.: Lawrence Erlbaum Associates.

Mostow, J. (1985). Toward better models of the design process. AI magazine, 6 (1), 44-57.

Myers, B.A. (1986). Visual Programming, Programming by Example, and Program Visualisation: A Taxonomy. Proceedings of ACM/SIGCHI, 1986, 59-66.

Newell, A. & Simon, H.A. (1972). Human Problem Solving. Englewood Cliffs, NJ.: Prentice-Hall,

Reichgelt, H. & van Harmelen, F. (1986). Criteria for choosing representation languages and control regimes for expert systems. The Knowledge Engineering Review, Vol 1, No. 4, 2-17.

Simon, H.A. (1969). The Sciences of the Artificial. Cambridge, Mass.: MIT Press.

Walker, R.A. & Thomas, D. (1985). A Model of Design Representation and Synthesis. Proceedings of the 22nd Design Automation Conference.

Jan Rogier

The use of STEP in an Intelligent Design System for Architectural design

The use of STEP in an Intelligent Design System for Architectural design

Jan Rogier

Institute of Applied Computer Science (ITI)
Netherlands Organization for Applied Scientific Research (TNO)
Schoemakerstraat 97, 2628 VK Delft
fax +31 15 623313
Telex 38071 zptno nl
Tel. +31 15 697061

Interactive Systems (IS)
Centre for Mathematics and Computer Science (CWI)
Kruislaan 413, 1098 SJ Amsterdam
Telex 12571 mactr nl
Tel. +31 20 5924144
Usenet: rogier@cwi.nl

Abstract
STEP stands for 'STandard for Exchange of Product model data'. In its current state STEP is a more or less structured set of definitions of entities that are used to exchange product model data between different computer systems. To be used in the context of a design system, it should be possible to incrementally extend the knowledge about a product. This is possible when STEP is applied in combination with an object description as is used in the IIICAD system [HAG]. In that case STEP is used as a reference vocabulary in order to be able to apply evaluation rules on an *incomplete* object description. The structure of the object description as used in the IIICAD system offers STEP a formalism that makes it possible to extend the knowledge about a product incrementally. At the same time, the application of STEP in the IIICAD system guarantees a consistent product model in all stages of the design process.
The subject of this paper concerns the application of STEP for topological and geometrical aspects of an object description.

Keywords: Design Theory, Intelligent Design System, Design Object Description, Product Modelling, Standard for Exchange of Product Model Data (STEP).

1. Introduction

The result of a design process is twofold. On the one hand a description of the product to be realized is generated, on the other hand a formal description of the type, the product belongs to, is generated. A product type description is a parametric description of a product. A parametric description of the product type can be reused in other design processes. An important prerequisite for the reusability of a parametric description, therefore is that this description can be used as a prototype. This prototype is referred to in the context of a new design process. For a parametric description its usability as a prototype demands that the description is changed. This, combined with the fact that a product definition will be declared incrementally during a design process, has consequences for the actual form of the description. For several reasons, this form should be standardized. A good basis for this standard is supplied by STEP [STE]. However, in its current state STEP does not support an incremental declaration of a product description and a product type description. This paper describes an implemention of STEP in such a way that it can be used as a reference model during the declaration process of an object designed with the aid of an intelligent design system [HAG]. The paper therefore, implicitly describes the way STEP may be extended towards a language for describing and building descriptions of products: a product modelling language.

The contents of this paper is based on a number of axioms.

In the first place, this paper is based upon the architecture of the IIICAD system and the design object description used in the IIICAD system [HAG]. The contents of the paper is therefore implicitly based on the theoretical background of the architecture of the IIICAD system. The design theory describes the design process as a process in which a number of individuals contribute to the design object description at the same time. Each person is responsible for a contribution that is generated from his own view point and expertise. The goal of a design process is a complete and integrated design object description.

In the second place it is assumed that there exists a STandard for Exchange of Product model data. At this moment that standard is being developed. In its current state it covers the description of a large number of terms that are used to describe products.

In the third place the paper deals with a subset of the vocabulary of STEP concerning topologic and geometric aspects of design object descriptions. STEP however, covers a lot more aspects of product definition. The paper therefore assumes implicitly that elaboration of a topologic and geometric subset of STEP may be seen as representative for STEP as a whole.

In the fourth place it is assumed that it is not the aim of a design system to only apply parametric design object descriptions, but to be a mechanism that helps to construct, evaluate and apply a parametric design object description. The background of this statement is that history has learned that parametric descriptions are never completely satisfying under all circumstances. Also it is noticed that parametric descriptions of products have a tendency to be used as design standards and therefore on the long term will restrict designing extremely [BIJL]. Previously designed product type descriptions therefore are to use as prototypes [GER][EAS]. In order to use these prototypes, flexible design scenarios are used to guide the user's actions [VEE].

In the fifth place at last, it is assumed that the only way to coordinate a design process that is carried out by a number of people in cooperation, is to make them base their contributions on an integrated design object description. During the design process this integrated design object description is incomplete. In order to be able to reason about the design object description it is necessary to have information about the complete development of this description. This paper is concentrated on a way to describe the evolution of an object description by extending the description language.

2. The architecture of an intelligent design system.

The paper refers to a specific architecture of an intelligent design system. This architecture is based on a design theory. The theory claims that the most important contribution of a design system to a design process is to support the communication process between experts. The subject of this communication process is the incrementally extending design object description. Contributions of different experts are aimed towards the extension of the design object description. Every expert contributes to the design object description from his own viewpoint. This viewpoint is determined by the expert's knowledge domain. During each design process the task of one of the experts is to integrate all of the contributions of the other experts into one integrated design object description.

As such, the design process determines the architecture of a design system. This design system is built from a number of communicating expert systems, one for each applied knowledge domain. Examples of knowledge domains are those that concern construction calculation, cost analysis, etc.. During the design process each of the expert systems generates from its own viewpoint, comments on the current state of the design object description concerning incompleteness and inconsistency of this description. The task of only one of these expert systems is to support the integration of all comments into one coherent design object description. This system is called the 'designer system'. The designer system generates, based on all produced comments from other systems, a set of coherent design object descriptions. The designer selects out of this set the most promising description while the rest is stored to be used when a previously chosen design path turns out to have a dead end. In the next pass of the process, the selected design object description will be used as the basic description for all expert systems. The formal description of this design process on which the architecture of the design system should be based is illustrated in figure 1.

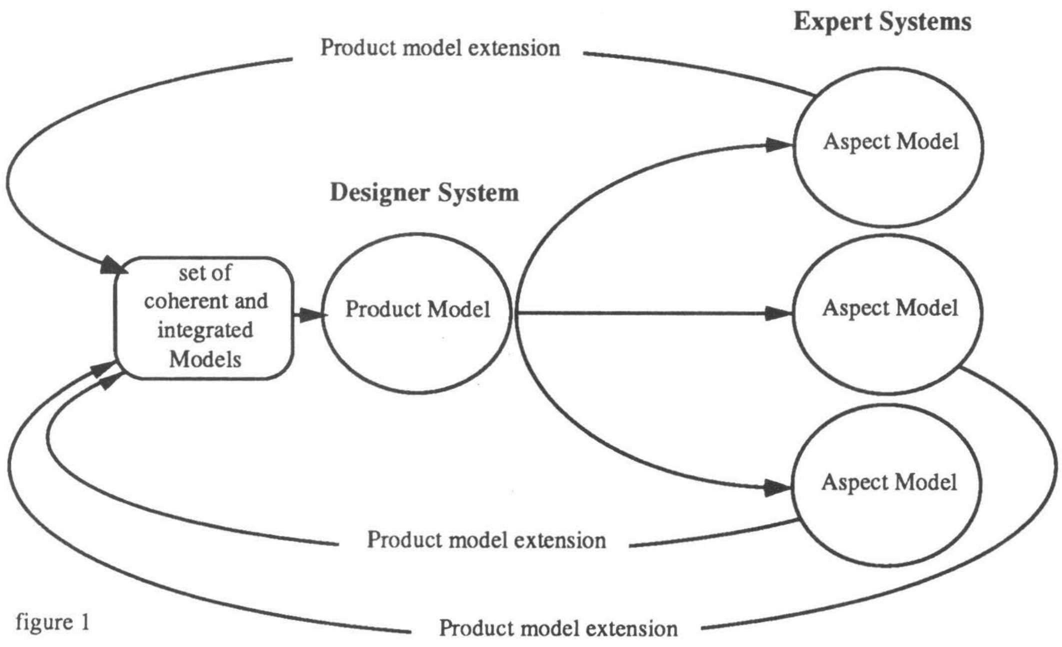


figure 1

From the viewpoint of one of the expert systems it is impossible to determine if the comment produced by one of the other expert systems is generated by the system itself or by human interaction with this system. One of the definitions of artificial intelligence says that one can use the term artificial intelligence when one can not any longer distinquish system's and human's contributions, actions, etc.. Both consideration justifies in this case the fact to talk about an 'intelligent' design system.

3. The conceptual application model.

Comments on the design object description are based on knowledge that is stored in an expert system, and that is more or less extended with interactively applied human knowledge. In order to be able to apply this knowledge, the design object description should be mapped on a model of the knowledge that is used by the expert. This model may be seen as the description of the semantics of the vocabulary that is used by an expert. The model of the knowledge will be referred to as 'the conceptual application model'. A conceptual application model contains, theoretically speaking, all knowledge that belongs to a specific knowledge domain that is applied by an expert. Examples of conceptual application models are a model that describes construction calculation of buildings (e.g. a FEM model extended with rules that describe how to apply the FEM model on buildings) and a financial model that is used to calculate (an indication of) the building's costs .

In real life practise it is impossible to build a conceptual application model that contains all knowledge that belongs to a specific knowledge domain. No model can contain all knowledge about a certain field of application or all knowledge about the justification of the applicability of certain knowledge. This makes it necessary to incorporate human knowledge interactively when the conceptual application model is used. Only in the case that a parametric model of an object turns out to be completely applicable, knowledge application could have been applied automatically. By incorporating previously build design object description as prototypes in the design system, it is possible to extend the knowledge about the applicability of the conceptual application model in concrete cases.

The mapping process of the design object description on the conceptual application model results in the determination of the incompleteness or inconsistency of the design object description. This result may be used as a guide for the extension of the description. Application of design knowledge by using previous design object description as prototypes will simplify this process.

4. The description of the design object

The design object description that is used within the design system describes the decomposition structure of a design object. Each part of the design object is described with a combined class and instance declaration. The class-part of the declaration concerns the parametric description of the product type. The instance-part of the declaration covers the knowledge of the concrete product. The decomposition structure is logically described by means of attributes that describe object types that are part of the specific object. The values of these 'part-of' attributes are references to instances of other object type descriptions.

Beside the description of the decomposition structure of a (part of an) object, every object has a set of attributes that are filled with constant values and a set of attributes that refer to expressions that describe constant value dependencies.These dependencies describe the mathematic relation that exists between the constant values of attributes of an object or parts of this object.

The structure of the object description is illustrated in figure 2.

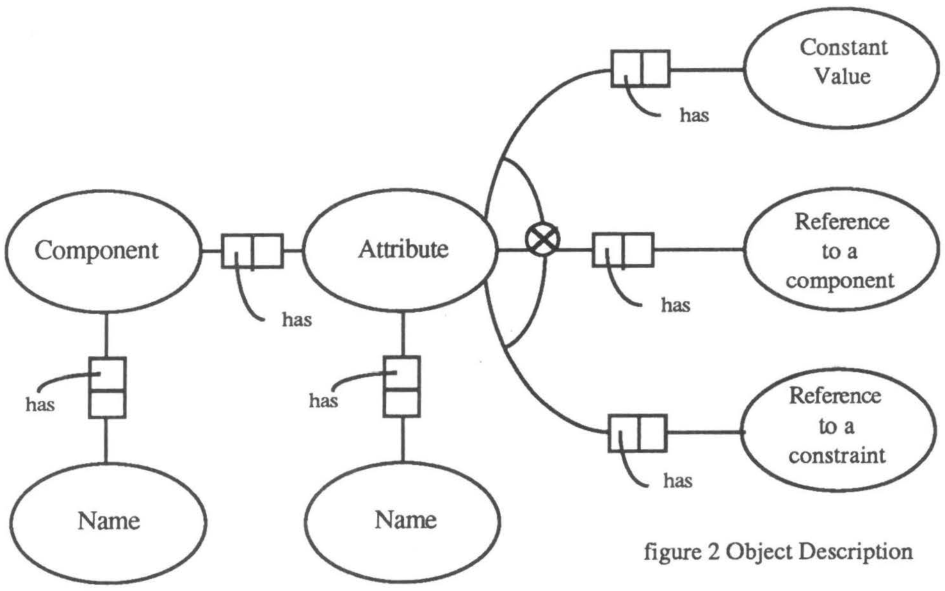


figure 2 Object Description

5. The use of STEP in an intelligent design system.

Using an intelligent design system in a design process has two concrete results. In the first place a complete and integrated description of a product will be realized. In the second place a parametric description of the product will be generated. This last description is the definition of the product type. The use of the parametric description of the product however, is not to reduce designing to a straightforward application of this description by 'filling in' values of parameters. Every future design process will continue to be a process of describing the product type, a process of describing the product and the evaluation of both in the context of the design constraints. The advantage of using an intelligent design system lies in the possibility to reuse a previously built (part of) a parametric description of an object and the knowledge about how this was translated into the different vocabularies of expert systems, in the context of a new design problem. From the point of view of the architecture of the design system, it is demanded that the parametric description of a product type is exchangable between the different components of the design system (the different expert systems) and is multi-interpretable for the purpose of mapping on different conceptual application models. The completeness of the parametric description should be guaranteed by documenting the evolution of this description.

A considerable advantage in the exchange of information is reached when the description is formalized by means of a standard. STEP has been developed for this purpose. In order to document the evolution of the description and to be able to use this to trace back this evolution, there needs to be a logical relationship between the different stages of the design object description. In order to express this logic relationship, the language that is used to describe the design object should have certain capabilities. These characteristics are described in the context of an object oriented implementation of a modeler. The modeler described in this paper is limited to application of topology and geometry.

Topology and geometry were originally intended to *describe* certain aspects of objects. In the context of a design system however, the object to be described is unknown. What is 'known' or assumed of the objects, is described by associating certain characteristics (attributes with values) to the object. These characteristics are also used to derive and describe certain consequences. Some of these consequences may be derived by associating topology and geometry with the object description. When the designer does not agree with the result of an association in terms of unacceptable consequences, he has either to redefine the object's characteristics or the topological or geometrical entity that is associated with the object. In order to be able to do so, the connection between the object's description and its topology or geometry should be loose. In order to be able to apply loose connections between an object's description and its topology and geometry, the entities that are used as values of the topologic and geometric attributes of the object should be evaluatable. In this case, the topologic and geometric entities that are used, have to structurize the declaration of the mapping between attributes of the design object that have a constant value and the attributes of the topologic and geometric entities (see example 1).

6. Object oriented modeler.

The object oriented modeler is described by a number of characteristics. The characteristics are illustrated by referring to the topology and geometry of a design object. It is assumed that the topology of every object is described by the entity 'region' and that all other topological properties are implicitly available depending from the object's shape. It is also assumed that the corresponding geometry of a 'region' may always manifest itself as a number of connected three-dimensional surfaces. To the other specific characteristics of the geometric attributes of an object is referred using a set of primitives. In the description of each primitive, the origin of the number of three-dimensional surfaces is specified. Each basic entity is described as a set of attributes and behaves like a frame [WIN].

7. Topology

The topology of each object is described by the entity 'region'. Every region is enclosed by a 'shell'. Every shell is enclosed by a 'loop'. A loop is a 'closed' list of 'edges'. An edge is defined by two 'vertices'. In fact each object has at least three different topological representations. One describing its global shape, one describing its shape depending from its context (e.g. the shap of the object being a component of another object) and one describing the object as a set of components (see example 2). The relations between components of a object are described by means of dependency expressions. These expressions are built using a set of operators. In the current implementation of the object oriented modeler only the entities 'shell' and 'region' are used (other entities like 'vertex' 'edge' and 'loop' should become implicitly available). The set of operators covers :Disjunct, :Conjunct, :Minus, :Inverse, :Materialize and the boolean operators :Cuts and :Inside. All non-boolean operators result in the instantiation of a new region or may be interpreted as a complex declaration of a region. The operators :Disjunct, :Conjunct and :Minus result respectively in the enclosing region, the common region and the first region minus the common region of two regions. The operator :Inverse inverts the region in such a way that its inside becomes the outside and vice versa. The operator :Materialize materializes the shell of a region into a region.

8.Geometry

Analogous to the topologic difference between the region and the shell at a geometric level, a distinction is made between a 'volume' and a 'surface'. The volume is defined as a set of connected surfaces enclosing a space, the surface is the description of a flat plane enclosed by a set of lines. Definitions of these entities are supplied by STEP. STEP also supplies the definition of a number of basic (geometric) entities. As a common characteristic all geometric entities may be represented as a set of connected surfaces enclosing a space. This makes it possible to define a number of common basic operations for all entities corresponding to the above mentioned operations on the topologic entity region. The number of these operations is reduced to two basic ones. The first basic operation calculates the geometric intersection of the set of surfaces describing two volumes. The second operation inverts the orientation of a volume by inverting the normal direction of each of its surfaces and inverting the direction of the loop of lines and the directions of the lines themselves, that encloses each of the surfaces. The result of the intersection operation contains the description of a set of outer shells of the combined space of two volumes and a set of outer shells of the common space of two volumes. When two volumes do not intersect, the result of the topologic operator :Cuts applied on the associated regions is 'nil' (False). When one of the volumes lies completely inside the other volume, the result of the topologic operator :Inside applied on the associated regions is 'T' (True). The topologic operator :Disjunct results, on a geometric level, in a set of volumes that are enclosed by at least one of the original volumes. The topologic operator :Minus result in a set of volumes that are enclosed by the first original volume and are not enclosed by the second original volume. The topologic operator :Conjunct results in a set of volumes that are enclosed by both original volumes. The topologic operator :Inverse inverts the orientation of the volume in the above mentioned way. The operator :Materialize results in a volume that encloses both sides of the shell of the original volume. This is done by adding a thickness to each surface that belongs the shell of the original volume (separately !).

9. Reference vocabulary

Within the context of a design object description, the geometric vocabulary functions as a reference vocabulary. Every object has at least one geometric attribute. The value of this attribute refers to a geometric primitive that is used as the geometric representative of the object's topology as a region (global shape). In the case of the previously mentioned other two topological descriptions, the value of the geometric atteribute is derived from the geometric attribute valiues of the entities mentioned in the topological expression.

The relation between the attributes of the object and the geometric entity it refers to, is described explicitly. This is done by describing the mapping between each of the attributes (of both the object and the geometric entity it refers to). In this way the calculation aspects of a large number of characteristics of the object are delegated to the geometric entities they refer to.

The 'value' of the geometric attribute is declared explicitly in the case of its global shape, and is derived from geometric description of other obejects in the cases of describing the objects shape depending from its context and the shape of the set of components. This makes it possible to incremently extend the design object description. Changing values will result in the need the change a number of indirectly dependend values. In realistic cases the number of dependend values is enormous. Which values are dependend and which values are to be changed interactively, depends from the current design process. In order to simplify this process the user chooses 'design scenarios' that guides the process. These scenarios describe on a higher semantic level how the declaration process and the dependend value replacement process should take place [VEE] (a replacement of values is described in example 2).

Example 1
To give an example of the use of delegation on a conceptual level: the calculation of the contents of a 'kitchen' during the design process does not belong to the characteristics of the kitchen. The specific calculation that is used is an aspect of the geometric entity that is used to represent the kitchen at that stage of the design process (for instance: a 'block'). The contents of the geometric entity 'block' is used as an indication of the contents of the design object 'kitchen' at that stage of the design process. This is done by mapping the 'height', 'width' and 'length' and the 'contents' of the kitchen on the 'x', 'y' and 'z-axis' and the 'contents' of the geometric entity 'block'. The responsibility of the application of the mathematic relation between the 'height', 'width' and 'length' and the 'contents', is *'delegated'* by the object 'kitchen' to the geometric entity 'block'.

Example 2
During the design process it is possible to extend the value of the geometric attribute. In fact each design object has (at least) three different geometric descriptions. A default one that describes its global shape, one that depends on a topologic constraint and describes the shape of the object as depending on its context (e.g. the 'kitchen' as a space in the context of the house woth all its other spaces), and one that results from the addition of the geometric attributes of all the object's components. All geometric descriptions shopuld be consistent with each other i the sense that they will not produce different values for the same constant valued attributes of the object. Addition of a geomtric attribute by means ofdeclaration of a topological constraint to the object description or addition of a component forces the user to declare both the mapping of attributes of the object on attributes of the resulting geometric shape and the relation between the different geometric attributes. This makes it possible to make the above mentioned 'kitchen' in its preliminary stage refer to the entity 'block'. In a later stage this reference can be extended with one that refers to the outer shell of a volume that result form a number of topologic constraints on other regions. When the extension takes place, the user is asked to declare data that describes the mapping between attributes of the object 'kitchen' and attributes of the associated geometric entity and the relation (:in, etc.) between the newly added geometric description and the former ones. The already existing values of attributes of the kitchen are checked in the context of the newly associated geometric entity. This may result in declaration of new values.

Example 3
Independent specification of both the object description and the definition of the geometric entities has an important advantage. Due to the independent specification it is possible to detect incompleteness or inconsistency of an object description. To give an example: During the specification of a 'kitchen' a geometric attribute may be declared that refers to the entity 'block'. Based on the description of the entity 'block' the user is asked to specify the attributes of the kitchen that are to be mapped on the attributes of the 'block'. In this case, it is the description of the entity 'block' that structures the description of the object 'kitchen'. In fact the user delegates the structurisation of the object description in terms of constant valued attributes, to the internal description of the entity 'block'. This mechanism will be fully used in

the case that the geometric reference of the 'kitchen' is extended with a reference to the more elaborated volume description that results from topologic constraints. In that case the user is asked to make a statement on all attributes of that complex shell. This process implicitly extends the design object description to a higher level of detailed information.

Example 4
The characteristics of the geometric and topologic primitives that are used for the description of the aspects of an object, make it both possible to exchange information between different experts and to establish a logical relation between the different stages of the design object description. The designer (in his role as design expert and not in his role as design process supervisor) will, in a first approach, see a building as an interrelated set of spaces. The property of a region that its shell may be materialized into a region itself, will be used by the construction calculation expert to transform the description of the building as a set of 'spaces' into the description of the volumes that should lay in between these spaces. These volumes will be used by the construction expert to create space for its construction elements (e.g. walls, pillars, floors etc.). The, in this way, extended description of the design object may be used by the designer to generate the descriptions of the resulting volumes of its set of spaces and to verify if these still meet the constraints from he designer's point of view. Doing so, the interrelation between all volumes is declared implicitly and in a retractable manner, so that, for instance, when the contents of the object kitchen should be extended, the enclosing construction elements are automatically replaced because their location depends on the position of the shell that encloses the volume 'kitchen'.

10. Conclusion

STEP defines a standard vocabulary for describing products. The aim of this vocabulary is to make it possible to exchange product model data between different systems. During a design process a comparable process takes place. The main difference with exchange of product model data is that during a design process exchange of *incomplete* product model data takes place. On the other hand, the result of each design process will be a product model that will be used in the same way as all completely described products STEP is designed for. The second reason for application of STEP in the context of a design system is, that during each design process at a certain stage, descriptions of standard components will be used. Both reasons plead for the use of a standard for product model data like STEP is. The exchange of incomplete product models however, has as a goal to extend these models, being able to interpret them in different ways. For instance, being able to map them on different conceptual application models. An important consideration in this matter is that every extension of the design object description should be retractable. Only in this case is it possible to trace back the design process without loss of data.

In this paper an implementation has been described of the topologic and geometric part of STEP. The specific use of this part in a design system, makes it possible to use the same description of a design object in different contexts and to extend it incrementally. At the same time it is possible to document the evolution of the product description in a simple way. Last but not least, by using the same primitives as are described in STEP, it is guaranteed that each design process automatically results in a product model and that during a design process, already existing product models stored in catalogues may be used as standard components.

Acknowledgements
The author wishes to thank Paul Veerkamp, Daan Otten, Ravic Pieters Kwiers, Frits Tolman and Wim Gielingh for the many fruitful discussions, which have contributed a lot to this paper.

References

[BIJL] A. Bijl, 'Strategies for CAD', in; Intelligent CAD Systems I, Theoretical and Methodological Aspects, Record of First Eurographics Workshop on Intelligent CAD, P.J.W. ten Hagen, T.Tomiyama (Eds.) Springer-Verlag, Berlin, 1987.

[EAS] C.M. Eastman, 'Prototype Integrated Buildingsmodel', Computer-Aided Design, Vol. 12, No. 3, pp.115-119.

[GER] Knowledge Based Computer Aided Architectural Design, J.S. Gero, A.D. Radford, R. Coyne and V.T.Akiner. Knowledge Engineering in Computer Aided design edited by J. Gero, Proceedings of the IFIP WG 5.2 Working conference on Knowledge Engineering in Computer Aided Design Budapest Hungary, 17-19 September 1984, North Holland Publishers 1985

[HAG] 'An Environment for Knowledge Representation in Design', Paul ten Hagen, Jan Rogier, Paul Veerkamp. Proceedings of 'Civil Engineering Expert Systems' 6-10 Febr. 1989 Madrid

[STE] STEP/PDES Testing Draft; St. Louis Edition ISO TC184/SC4/WG1 Document Number 165, Peter R. Wilson. Philip R. Kennicott, 10 sept. 1987

[VEE] P. Veerkamp, 'Multiple Worlds in an intelligent CAD system', in 'Intelligent CAD - Record of IFIP WG 5.2 Workshop on Intelligent CAD', North Holland, Amsterdam,

[WIN] T. Winograd: Frame Representation and the Declarative/Procedural Controversy, in :Representations and Understanding: Studies in Cognitive Science, D.G. Bobrow and A.M. Collins (ed.), New York: Academic Press, 1975, p.185-210

Paper Session *System Architecture*

Jean-Paul A. Barthes, Kamal El Dahshan,

Patrice Anota

An experience in adding persistence to intelligent CAD environments

ABSTRACT

full paper to be distributed

AN EXPERIENCE IN ADDING PERSISTENCE TO INTELLIGENT CAD ENVIRONMENTS

by
Jean-Paul A BARTHES(1,2)
Kamal EL DAHSHAN (1)
Patrice ANOTA (1)

Submitted for presentation to the Third Eurographics Workshop on Intelligent CAD Systems, to take place in Texel, the Netherlands, in April 1989

(1) Université de Technologie de Compiègne
Dépt de Génie Informatique
C.N.R.S. UA 817
BP 233 60206 COMPIEGNE Cédex
Tél: (33) 44-20-99-60
Telex: 150 208 UTC
Fax: (33) 44-20-18-73

(2) SGN
1 rue des Hérons
78184 ST-QUENTIN-YVELINES
Tél: (33-1) 30-58-65-81
Telex 698 316 SGN (Ms CAPEL)
Fax: (33-1) 30-58-65-22

Extended Abstract

In a previous paper we reported our efforts in using *object centered representation*, *object-oriented languages*, and *constraint propagation*, for helping the designer in the context of *mechanical assembly problems* [El Dahshan & Barthès 88]. A major criticism of such an approach is the lack of *efficient disk storage*. Indeed object-oriented languages use quite specific environments, and usually the only possibility for saving objects in between work sessions is to keep a core image on disk. However for practical tasks involving thousands of components kept over a long period of time, or for tasks involving cooperative work between several designers or several computer programs, this is clearly inadequate, as already reported by [Fox & McDermott 86]. A simple idea for solving the problem would be to use existing commercial databases to store the objects permanently. To do so one needs first to express the objects in the relational model format. This, however, does not seem to be a good idea as reported by [Rumbaugh 87]. Furthermore, when actually implemented, as was done by Katz and Rowe at Berkeley in the domain of electronic design, with POSTGRES an extension of the relational database INGRES, there seems to be a strong "impedance mismatch" between the process handling objects in core and the DBMS. Another experience comparing relational databases with object-oriented databases is reported by [Smith & Zdonik 87] in the domain of multimedia applications. Thus we conclude that there are delicate problems when trying to interface intelligent CAD environments with permanent storage. What are the problems and what can be done? Such are the questions we try to answer, illustrating them with some results from our

research work. (Curiously the database research community does not mention CAD as a potential important application as can be read in the Laguna Beach report about the future directions in DBMS research [Laguna 88].)

The usual difficulty when talking about design comes from the non unicity of the field. CAD has different meanings in different domains, leading to different requirements. However there seems to be features some common to several domains.

In the past users have been sensitive to the graphical representation of objects on the screen; this is a problem of external or "surface representation" which must be adapted to displaying hardware, to the design phase, and to the designer mentality. It is now clear that graphics although important is secondary in the general representation of objects. We shall ignore graphics interfaces in the paper.

Objects are represented according to a model which has some expressive power. Slightly different models may have quite different possibilities. Semantics must be specified. Then, in practice objects are implemented using a physical format, which has consequences on the efficiency. Finally if the objects are represented as pointer structures in core, they also need a disk or an exchange flat format. Transformation between in core and disk formats may be costly, which is in particular the case between Lisp type core format and relational tables (the so-called "impedance mismatch").

Finally *objects must be shared*, otherwise the applications are severely limited. However sharing objects is not easy, because traditional locking mechanisms break down for long transactions as found in CAD applications.

In the paper, we detail the above mentioned problems and show how we have dealt with some of them in the OPAL system, implemented on top of the LOB system (an object-oriented database) [Barthès 87a] itself written using BOSS, an object-oriented programming environment [Barthès 87b]. The obtained result is a *seamless shared environment*, however unlike in [Thatte 87] it is not a permanent virtual memory, which we find not appropriate for long term design applications.

Objects in OPAL are represented using a "frame-like" recursive format, called PDM (for Property Driven Model), in which attributes are themselves represented as objects. The format allows a precise semantics of specification in the traditional database style, rather than of prototyping found in artificial intelligence. The effective modeling of the representation structure in addition to its meta-circular definition, allows to change the models dynamically by programs, which cannot be done with languages such as SMALLTALK or in systems derived from it like GemStone [Maier & Stein 86]. Furthermore PDM emphasizes the role of attributes which can be shared by several classes. Thus, attributes factor a given behavior across such classes, which is contrary to the classical database approach that attributes are strictly dependent on a given class.

OPAL is implemented in a Lisp environment which is used as a *global cache mechanism* for each user sharing the same data (objects) on disk. Several problems had to be solved:

-implementing a good disk interface between the Lisp environment and the disk; this was solved by designing a *new disk access method* (on a VAX/VMS system), called MLF (Monitor for Lisp Files), allowing to swap objects in and out of core efficiently,

-designing a *specific core and disk formats* for minimizing page faults in the core virtual memory, for storing objects on disk compactly, and for minimizing the restructuring time when bringing the objects from disk into core; thus unlike in GARDEN [Skarra *et al* 86] we do not work on the objects directly in the buffer in their the disk format. Some measurements are given for three possible representations (ASCII, FASDUMP, and PDM-compact format).

Finally we address the problem of *sharing objects* which is a difficult one. There is no clear agreement on the mechanisms which should be used for locking, whether they should be classical like two-phase lock, or of the trigger or alerter type, nor on the granularity of the part that should be locked. [Andrew & Harris 87] advocate using triggers, [Penney & Stein 87] lock disk segments and use an optimistic approach, Anota implemented a classical two-phase lock using a single object as the basic locking grain on G-BASE† [Anota & Barthès 88]. We also tried a simple mechanism of deferred update, which seems to work quite well in most cases, and which avoids some problems found in other approaches; it could be combined with an alerter system to be entirely general (but this has not been done yet). A good discussion of locking problems can be found in [Kim *et al* 87].

In conclusion we try to indicate, based on such early results, what we think are the problems which remain to be solved if one want to add efficiently persistence to intelligent CAD environments.

† G-BASE is a trademark from GRAPHAEL

REFERENCES

[Andrew & Harris 87]
Timothy ANDREWS, Craig HARRIS
Combining Language and Database Advances in an Object-Oriented Development Environment
OOPSLA 87, ORLANDO Proceedings, Norman MEYROWITZ Ed., Special Issue of SIGPLAN Notices, Vol 22, N.12, pp 430-440, october 1987
[Anota & Barthès 88]
Patrice ANOTA, jean-Paul A BARTHES
Le Contrôle de Concurrence dans les Bases de Données Orientées Objets
Journées INRIA/AFCET sur les Bases de Données Orientées Objets, Paris, Déc 1988.
[Barthès 87a]
Jean-Paul A BARTHES,
LOB 1.0,
Mémo UTC/GI/DI, U de Compiègne/CNRS UA 817, 1987.
[Barthès 87b]
Jean-Paul A BARTHES,

BOSS 2.2,
Mémo UTC/GI/DI, U de Compiègne/CNRS UA 817, 1987.

[El Dahshan & Barthès 88]
Kamal EL DAHSHAN, Jean-Paul A BARTHES
Implementing Constraint Propagation in Mechanical CAD Systems
Conférence Invitée - Second Eurographics Workshop on Intelligent CAD Systems
Proceedings pp 235-248, Koningshof, Veldhoven, The Netherlands, April 12-15, 1988.

[Fox & McDermott 86]
Mark S FOX, John McDERMOTT
The Role of Databases in Knowledge-Based Systems
in On Knowledge Base Management Systems, ML BRODIE & J MYLOPOULOS Ed., Springer Verlag, 1986.

[Kim *et al* 87]
Won KIM, Jay BANERJEE, Hong-Tai CHOU, Jorge F; GARZA, Darell WOELK
Composite Object Support in an Object-Oriented Database System
OOPSLA 87, ORLANDO Proceedings, Norman MEYROWITZ Ed., Special Issue of SIGPLAN Notices, Vol 22, N.12, pp 118-125, october 1987

[Laguna 88]
Future Directions in DBMS Research
The LAGUNA Beach Report, April 1988

[Maier & Stein 86]
David MAIER, Jacob STEIN
Indexing in an Object-Oriented DBMS
Proceedings of the 1986 International Workshop on Object Oriented Database Systems, Computer Society Press, IEEE, pp 171-182, 1986
TH0161-0/86/0000/0171$01.11 © 1986 IEEE

[Penney & Stein 87]
D. Jason PENNEY, Jacob STEIN
Class Modification in the GemStone Object-Oriented DBMS
OOPSLA 87, ORLANDO Proceedings, Norman MEYROWITZ Ed., Special Issue of SIGPLAN Notices, Vol 22, N.12, pp 111-117, october 1987

[Rumbaugh 87]
James RUMBAUGH
Relations as Semantic Constructs in an Object-Oriented Language
OOPSLA 87, ORLANDO Proceedings, Norman MEYROWITZ Ed., Special Issue of SIGPLAN Notices, Vol 22, N.12, pp 466-481, october 1987

[Skarra *et al* 86]
Andrea H.SKARRA, Stanley B.ZDONIK, Stephen P.REISS
An Object Server for an Object-Oriented Database System
Proceedings of the 1986 International Workshop on Object Oriented Database Systems, Computer Society Press, IEEE, pp 196-204, 1986

[Smith & Zdonik 87]
Karen E SMITH, Stanley B ZDONIK
Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems
OOPSLA 87, ORLANDO Proceedings, Norman MEYROWITZ Ed., Special Issue of SIGPLAN Notices, Vol 22, N.12, pp 452-465, october 1987

[Thatte 87]
Satish M.THATTE
Persistent Memory: A storage Architecture for Object-Oriented Database Systems
Proceedings of the 1986 International Workshop on Object Oriented Database Systems, Computer Society Press, IEEE, pp 148-159, 1986
TH0161-0/86/0000/0148$01.11 © 1986 IEEE

A.T. Shenton, A. Taleb-Bendiab

and Y. Chen

Computer-Aided Constraint Development Systems for Conceptual and Embodiment Engineering Design

Computer-Aided Constraint Development Systems for Conceptual and Embodiment Engineering Design

A.T.Shenton, A.Taleb-Bendiab and Y.Chen
Department of Mechanical Enineering, University of Liverpool

SYNOPSIS

A systems engineering design philosophy and representation scheme is presented. This approach forms the basis for a logic-programming computer aided design system for the early stages of engineering design.

The features of a suite of logic-programming facilities which is under development at the Department of Mechanical Engineering, University of Liverpool are described. This facility provides a computer-aid for the conceptual and embodiment stages of engineering design.

Programs are written in Core Prolog Ref[1] primarily using Waterloo Prolog version 1.6 Ref[2] running on the University IBM3081 mainframe. Graphics is through Tektronix 4107A facilities.

1 INTRODUCTION

In engineering design discovery of constraints, requirements and trade-offs proceed along with the concretisation, firming up and progress of the design. The most successful design strategies require deferred decision making not only in design realisation but also in design requirement specifications. A computer aid in this process must necessarily be interactive with the designer. It is suggested that a primary role of such a system should be the identification illustration and explanation of the trade-off's between constraints and requirements for the discovery of new and revised constraints and requirements.

In the logic programming system described in this paper primary design requirements are formulated in terms of the transformation of energy, materials and information from inputs to outputs (Pahl and Beitz [3], Shenton [4]). A unified underlying representation of the design object is enforced. This embodies the representation of a physical process allied with input and output energy ,material and information flows. The input output flows may be associated with geometric form interfaces which may be constrained by the process rules. The representation may be considered as an extension of automata and dynamical-systems representation schemes (Arbib [5], Zadeh and Desoer [6]).

A novel facility is the representation of geometrical relations and features explicitly within the interpretable rule-base in the form of 2D variational geometry (Aldefeld[7]). This scheme separates topological and relational rules from metric dimensional data. Consequential relations may be generated by forward chaining to generate geometric realisations. This involves only symbolic manipulation and quantification and numeric manipulation is a separate process.

Explanation of the state of the search tree is available in this case with the provision of a 'why' facility. The interpreter (inference engine) thus allows a wide explanation and discovery of the consequences of assumed design artefact geometry.

In the associated systematic design philosophy the design object system functions are realised by the selection of 'physical principles' (Pahl and Beitz[3]) here thought in terms of the two separate concepts of 'physical processes' and 'process applications'. Physical principles incorporate physical and geometric effects but only require partial definition of the component geometry. In the later embodiment design stage physical principles are embodied in realisable solid shapes with complete geometric forms which are the proposed 'physical solutions' (Pahl and Beitz [3]) to the design.

A key feature of the system is that backtracking to alternative processes and applications or variants is under the control of the designer. Backtracking during the embodiment stages returns to the highest level geometric detail processes first. This 'dependency directed backtracking' (deKleer [8], Stallman and Sussman [9])is used to retain selected physical principles during the evolution and development of physical solutions. For example a wedge effect requires a minimum of two half-planes (tracks) and an angle of inclination as geometric features.

A realisation of a wedge as a potential 'physical solution' to the design might involve additional half-planes to form a complete bounded solid. During design backtracking the designer should thus be made aware that alteration of the wedge angle alters a feature of the physical principle. The inclusion of geometric reasoning and explanation procedures is clearly crucial in the 'discovery' of these processes.

A module structure (Shenton [4]) is used for component representation. The design components are represented by process 'applications'. The module structure is used to represent assemblies of systems and components and the hierarchical level of abstraction of the design at different stages of elaboration. A module includes component module relations and constraint rules. Component module statements define the module hierarchy. Constraint rules must be true when active but may be deferred by incomplete instantiation.

The 'function' and 'application' modules are distinguished from other representation schemes (Barrow [10], Popplestone, Ambler and Bellos[11], Aitchison and Wilkie [12] ,Chan and Paulson [13]) by associating input and output lists of data-structures with these relations and constraints. The physical 'process' modules however have a conventional 'port' interface representation. Physical process modules may be used to represent interfacing relations such as 'force-fit' for relating a shaft to a hole. A significant advantage of input-output lists is that they may be readily matched and unified for functional synthesis. In particular they allow the interpreter to conduct an explicit examination of the unification (input-output matching) process. This matching is reflected in subsequent examination and discovery by the generation of special interface ('link') constraint rules included within combined ('synthesised') modules.

A principal advantage of the unified representation is that it allows the integration of different modes of designer interaction with a universal rule base. This may be done through 'design-specification operators' for energy material and information flows or through graphical input of relational data in addition to direct assertion of module constraints and features.

The levels of abstraction represented by the module hierarchy is a mechanism to retain distinction between primary and secondary requirements.

2 <u>THE DESIGN PARADIGM : DESIGN ENTITIES</u>

2.1 FUNCTIONS

2.1.1 <u>Categories of requirements or functions</u>

The primary function

In the proposed scheme the overall requirements as initially perceived by the designer are expressed in the form of a primary function.

Secondary functions

The introduction of partial solutions to the overall function in the form of applications of particular processes often give rise to new additional requirements as a by-product of the selected process. Thus the selection of a valve requires sealing as a secondary function and the selection of a rotating shaft requires a support for the running bearings.

Residual functions

As the design progresses parts of the overall function are partially solved by the application of physical processes. Residual functions remain however and these must be determined from the appropriate reduction of the overall function by the appropriate 'removal' of the proposed partial solution. Secondary functions may be conveniently embedded in the residual functions.

2.1.2 <u>Representation of requirements or functions</u>

In the proposed representation there is no distinction in the data-structure for the different categories of functions. The hierarchical relation of different functions is however embedded and this allows the designer to judge the features which should be preferred for retention in a design and those which can be more readily discarded with less impact on the design.

The function is regarded as consisting of the required inputs, outputs and states and the required processes. The use of inputs and outputs expresses the assignment of causalities to energy efforts or flows as required. Associated time delayed states such as accelerations are required to be assigned as effects in accordance with the principle of causality. Particular material forms may be associated with groups of energy flows or efforts. Individual efforts or flows may themselves be associated with particular geometric forms such as axes, lines of action etc.

In material flow the input-output assignment indicates the direction of material flow and the time history of individual particles or components.

Information flow is tramsmitted through the medium either of energy efforts or flows or through material configurations. In a pure information processing system the medium is irrelevent. In real systems however the speed and economy of processing often dictates electronic and digital processes.

In general designs will be dynamical systems. This paper considers only the quasi-static dynamical systems.

2.1.3 Function process representation

Function processes are defined in terms of the inputs, outputs and states. The process descriptions are entered by the designer as rules or operations. The general requirement is to provide a vocabulary compatible with the fundamental process operators of systematic design methods including the set due to Koller Ref.[14]:

change function:		
	change	revert
	forward	reverse
	increase	decrease
connect function:		
	couple	interrupt
	join	separate
	assemble	divide
channel function:		
	transmit	isolate
	collect	scatter
	rectify	oscillate
	conduct	insulate
store function:		
	absorb	emit
	fill	empty

The operands of this set are clearly the material, energy and information components of the system.

In the representation scheme described below some of these operators may be expressed explicitly within the function modules. Others have an implicit meaning and may be expressed in the form of rules describing the function representation.

2.2 PHYSICAL PROCESSES

2.2.1 Physical principles and form features

The concept of a physical process is that of a 'physical principle' and associated 'form design features' without a particular predefined purpose.

Particular physical processes may be used repeatedly even within the same design. It has accordingly been found useful and efficient to store processes in the form of types and to refer to particular uses of these types as instances of these types.

2.2.2 Representation of processes

Processes are considered as interfaces of energy and material ports and the logical or relational rules describing the relation of the elements of these interfaces and any internal states of the process. The description of the process relation includes geometric, shape and form rules.

Physical processes may of course be very complex, they are numerous and they offer many possible forms for implementation. A basic form of process modelling involves the explicit representation of pre-defined feasible processes. An alternative approach is a form of implicit representation in which the representation is constructed by inference from an underlying model of the physical effect.

2.3 APPLICATIONS OF PROCESSES

2.3.1 The use of physical processes

Whereas the concept of a physical process is roughly that of a 'physical principle' and associated 'form design features' the concept of an application of a physical process is that of a 'solution principle' for a sub-function Ref[3]. In other words the concept of an application is that of a 'physical principle' and associated 'form design features' applied to solve a particular sub-function.

Accordingly the application of a process involves a decision about the appropriate assignment of causalities and inputs and output in the chosen process.

3 THE DESIGN PARADIGM : CONCEPTUAL DESIGN OPERATIONS

3.1 CONCEPTUAL SYNTHESIS

3.1.1 Primary requirements

The first stages of the design process start with a preliminary specification of the primary function of the design object.

3.1.2 Explicit function decomposition

In the case that parts of the overall function have clearly defined physical roles such as motor and pump and may be split into different forms of energy then the function may sensibly be decomposed Ref [4] without further reference to the availability of any particular physical processes. Similarly if the function represents a sequential processing system then an 'a priori' structured decomposition is appropriate.

3.1.3 Sub-function requirements

Once a function or sub-function has been defined then the designer can attempt to partially or completely realise the required function by matching it to a process. Since this process may not be too successful the activity is actually a trial 'approximate' matching. The designer may attempt to match many processes to the residual function. To select the 'best fit' or most promising process a measure of the 'nearness of a match' is required. This may be on the basis of the designers judgement or by an automatic process. In the case that an exact fit is achieved the process may be subject to optimisation and comparison with other candidates on economic grounds.

One criterea for comparison is clearly the degree of input-output matching. It is of course frequently necessary in practice to employ strategies which involve the generation of secondary functions in which the number of inputs and outputs of subsequent residual functions increase but in which the associated function processes simplify or become more tractible. In this case assessment of the suitability of candidate processes necessarily must involve a comparison of associated alternative processes. The matching of function and physical-process process-operators may be used to directly select candidate physical-processes. The designer has a significant role in this selection.

Once a physical-process has been chosen the way in which the process is to be used to realise the required function must be established. This will often be determined in the matching process itself. This stage is the generation of the application. At a simple level this may be no more than the recording of which ports of a process are assigned as inputs or outputs.

After the selection of a candidate process and its mode of application it is necessary to determine the remaining components of the overall function in the form of a 'residual function'. The application matching may result in an exact match or a precise decomposition of the system function. In this case determination of the residual function is either not necessary (null residual function) or amounts to a simplification of the initial function.

In the case that the matching and application generation stage results in only an approximate match then a possibly complex residual function must be determined. Inputs matched to outputs and outputs matched to inputs result in cancellation. Operations which are performed in the application may be used to reduce the remaining requirements but secondary functions in the application may add to the resulting residual function.

3.1.4 Combination of applications

After the selection of a particular application of a physical process if previous applications have also been selected and if the set of applications are not interconnected then they may be combined. The combination of physical processes is concerned with ensuring compatibility at the sub-system input-output interfaces. This compatibility obviously has consequences for the internal process rules and may result in rule conflicts which must at some stage be resolved.

If the way in which the processes are combined is new then the result is usefully recorded for the generation of further applications. This may be achieved by recording the underlying physical-process of the combined applications rather than the combined application itself. This approach allows the underlying physical-process to be applied in other ways perhaps with different causality assignments.

3.2 CONCEPTUAL ANALYSIS

3.2.1 Conflict identification

It is suggested that a primary role for an Intelligent CAD system is the discovery of conflicts and constraints.

It is pointed out above that the compatibility requirements of applications and physical processes may result in rule conflicts which must be resolved. In addition at other stages of the design process assignments of values are made to application parameters and these may also cause conflicts. A principle role of analysis is the identification of these constraints. The identification process is required to identify the origin of conflicts to aid the designer determine the means of their resolution. The resolution of these conflicts may be achieved by inference from the process rules and the compatibility requirements. For example the requirements of a single action thrust bearing may require thrust:f_x > 0

whereas the the action of a collar attached with a wedge type process may require

collar:f_x < 0.

If

thrust:f_x equals collar:f_x

then there is a conflict. Happily of course a variant application of the thrust bearing would require

thrust:f_x > 0

instead.

3.2.2 Conflict resolution

One means of resolving such conflicts is by a dependency directed backtracking to alternative process applications, physical processes or functional structures. An alternative is to modify and relax the constraints. This will be feasible in the initial stages of design when only approximate and somewhat arbitrary constraint information has been laid down.

4 THE DESIGN PARADIGM : EMBODIMENT DESIGN

4.1 SYNTHESIS

Once a potential conceptual solution in the form of consistent matched process applications has been found for the overall primary function the layout and form design must be completed. The process applications contain relations based only on the proposed functional surfaces (Koller [14]). The objective of the designer at this stage is to complete the design to the state of a geometric solid whilst invoking the principles of good engineering form design (Leyer [15]).

Since trade-offs and conflicts inevitably may arise in this stage it is necessary for the designer to determine the consequences of form design decisions. Thus if the designer rigidly connects a shaft with a bearing housing it must be understood that that the rotation principle of the bearing is vialated. From a design perspective the general geometric non-metric relations (such as parallelism, normality, tangency etc) are the features of principle interest at this stage. In a conventional drafting approach, however ,decisions about precise co-ordinate and metric data must be taken by the draftsman/engineer.

In the case that the distinction between functions and physical process applications is maintained during the design process an interpreter can then explain the purpose of basic elements. Moreover such an approach offers a useful separation of concept and embodiment tasks.

4.2 Analysis

In addition to checking the interconnection compatibility it is necessary to avoid conflicts between embodiment geometry and conceptual principles. The overall form must conform to the priciples of geometrical constructions and solid models. If embodiment decisions are expressed in non-metric relational form then conflict occurs when the data is geometrically overdetermined (Aldefeld [7]).

The form is incomplete when geometrically underdetermined. As a solid the embodied design object must avoid geometric conflict of intersecting domains of explicitly differing components or material. Effective surfaces must not geometrically 'collide' or overlap unless they themselves can be combined.

5 REPRESENTATION SCHEME

5.1 OVERVIEW

The representation scheme distinguishes between three principal design data structures:

- 'functions' representing design requirements and purpose
- 'processes' representing physical processes and partial or complete characterisation of physical objects
- 'applications' representing a particular application of a physical process or object to achieve a particular function

In the logic programming scheme described these are stored as Prolog clauses.

5.2 FUNCTIONS

The general representation of a function is as a set of Prolog clauses of the type

```
f*(function_name,function_variant,...).
```

stored in the Prolog data-base.

Suppose for example it is required to design a special lathe-centre to accommodate large diameter tubes of varying radius. A means of adapting to the different possible radii is to have for each tube type a detachable ring or collar of the same inside diameter of the tube. A conical wedged ring surface provides a standard means of attachment for each collar. The lathe centre would also be required to rotate freely, to locate the rotation about the lathe axis and to absorb thrust and radial loads.

The statement of a suitable primary function for the tube turning centre named as 'center_f' has the internal representation:

```
/*---------------------PRIMARY FUNCTION--------------------------------*
/*--------------------------MODULE-------------------------------------*
/*                          tube-centre                                *
/*------------------------I_O SECTION----------------------------------*
f*(center_f,1,in(
    [mat,[force_vel(f_x,v_x),axis_x],
         [force_vel(f_y,v_y),norm(axis_x)],
         [torque_speed(t_x,s_x),axis_x]   ])).
f*(center_f,1,out([]).
/*------------------------RULE SECTION---------------------------------*
f*(center_f,1,rule,connect(mat,mat2)).
f*(center_f,1,rule,free(f_x)).
f*(center_f,1,rule,zero(v_x)).
f*(center_f,1,rule,free(f_y)).
f*(center_f,1,rule,zero(v_y)).
f*(center_f,1,rule,free(t_x)).
f*(center_f,1,rule,free(s_x)).
f*(center_f,1,rule,is_axis(axis_x)).
```

The 'I_O' section details the input and the outputs in Prolog lists grouped by material form, energy form and then by associated geometric form. The use of the Prolog null list

    out([])

indicates that in this case it is required to have no overall system output.

The 'rule' section prescribes the required system function. There is to be a means of material connection and disconnection

    connect(mat,mat2)

between the primary input medium and the the subsequent material. There is to be a defined axis of rotation

    is_axis(axis_x)

about which their is to be no constraint on the applied torque

    free(t_x)

and no constraint

    free(s_x)

on the rotational speed. The tube centre is required to absorb the radial an axial forces

    free(f_x)
    free(f_y)

of the turning process but to rigidly locate

    zero(v_x)
    zero(v_y)

the turned component.

5.3 PROCESSES

The general representation of a process is as a set of Prolog clauses of the type

```
               p*(process_type,process_name,...).
```

stored in the Prolog data-base.

Consider the elementary process 'rev_face' of Fig[1] which is simply used to represent a rotational surface interface between some material and some other as yet unknown interface.

```
/*---------------------------MODULE------------------------------------*/
/*                           rev_face                                  */
/*-----------------------PORT SECTION----------------------------------*/
p*(type,rev_face,port,[
p*(type,rev_face,port,[
       [mat,[force_vel(f_x,v_x),axis_x],
            [force_vel(f_y,v_y),norm(axis_x)],
            [torque_speed(t_x,s_x),axis_x]],
       [void,[force_vel(f_n,v_n),norm(face_s)],
             [force_vel(f_s,v_s),norm(face_s)],
             [torque_speed(t_x,s_x),axis_x]]   ]).
/*-----------------------RULE SECTION----------------------------------*/
p*(type,rev_face,rule,is_halfspace(face_s)).
p*(type,rev_face,rule,angle(alpha,axis_x,face_s)).
p*(type,rev_face,rule,is_axis(axis_x)).
p*(type,rev_face,rule,(f_n equals f_x * sin(alpha)
                       plus  f_y * cos(alpha))        ).
p*(type,rev_face,rule,(f_n equals f_x * cos(alpha)
                       minus f_y * sin(alpha))        ).
p*(type,rev_face,rule,(v_n equals v_x * sin(alpha)
                       plus  v_y * cos(alpha))        ).
p*(type,rev_face,rule,(v_n equals v_x * cos(alpha)
                       minus v_y * sin(alpha))        ).
p*(type,rev_face,form,face_s,face).
or
P*(type,rev_face,rule,is_face(face_s)).
```

A process represents a physical process or the features of an object and consequently does not have any intrinsic 'purpose' in its own right other than that assigned by human intention. For example the same gearwheel can be used as a driving wheel an idling wheel or a driven wheel: each with a different function. Accordingly a process does not itself contain input-output causality assignments. These are reserved for the 'applications' of processes of Sec 5.4.

The 'port' section details the ports (assignable as inputs or outputs) in Prolog lists again grouped by material form, energy form and then by associated geometric form. The geometric interface of the process

is_halfspace(face_s)

is a 2D halfspace or directed line. The definition of an angle

angle(alpha,axis_x,face_s)

between the halfspace and the axis allows the relation between the force components and the velocity components to be defined.

Processes may be built up in a hierarchical way. Thus the rotating space 'rev-face' Fig[1] and the bearing surface 'smooth' Fig[2] form building blocks from which the 'brg_face' process Fig[3] is constructed. Other processes so constructed include 'radial' Fig[4], 'collar' Fig[5] and 'thrust' Fig[6].

Further basic processes include 'absorb' Fig[7] (to define encastre forces and torques) and 'joint' (to describe the summation of forces and commonality of velocities at a common point in a solid) .

5.4 APPLICATIONS

The assignment of causalities to the encastre process 'absorb' is stored simply as a list of which ports are to be regarded as inputs and which are to be regarded as outputs.

```
/*--------------------------MODULE-----------------------------------*/
/*                 match primary function with absorb                */
/*----------------------PORT SECTION---------------------------------*/
a*(case,absorb_m,1,center_f,1,absorb).
a*(case,absorb_m,1,port,[ground,[in,in,in,port]]).
```

There can of course be more than one application of any given physical process or object. Thus any one application is simply one case of the possible uses of the process. Moreover although the assignment of causalities must be compatible with the requirements of the system function it is in general possible to assign the ports and their caualities in a variety of ways for a particular function and a particular process. In general it is thus necessary to record the identity of the variant chosen.

This may be done with a 'variant number' to be associated with the application name (the variant number is shown as one above). An example of this is the use of 'joint' to either sum one set of generalised input forces with two sets of generalised output forces or two input sets and one output set. Another example is the possibility of using either the inner race or the outer race of a radial axial bearing as input.

6 <u>LOGIC PROGRAMMING FACILITIES</u>

6.1 OVERVIEW

The process of synthesis is developed by facilities for the matching and combination of inputs, outputs and processes and for firming-up by quantification and addition of new constraint rules which arise as a result of engineering analysis and optimisation.

The process of discovery (of new constraints and requirements) is developed with the aid of facilities for examination of the consequences of the new rules and their combinations (interface rules and input-output unifications). Thus the interaction of rules and the unification of parameter structures may be processed and studied for the identification, explanation and illustration of confluences (deKleer[8] Stallman and Sussman[9]) , conflicts (contradicting rules) and trade-off's (competing and interacting rules) competing for resources from defined costs or objectives. This facility is based on the fundamental characteristic of logic programming (Prolog) to treat data and program as one whole.

6.2 UNIFICATION SYSTEM

6.2.1 <u>The role of unification</u>

The primary role of the unification facilities are the unification of inputs and outputs and the generation of consequential compatibility ('link') rules. Unification facilities are required for:

function decomposition
  sub-function inputs to initial function inputs
  sub-function outputs to initial function outputs

trial matching
  process ports to function inputs
  process ports to function outputs

residual function determination by matching
  application outputs to chosen initial function outputs
  application inputs to chosen initial function inputs

unifying new with existing combined applications by matching
  outputs of existing applications with inputs of new applications
  inputs of existing applications with outputs of new applications

In general such unifications are not unique. Variant designs may accordingly be generated by backtracking the unification.

Function decomposition is in general complex Fig[8]. It is however useful (as described above) when there is a sensible serial or parallel representation, Figs[9a] and [9b].

6.2.2 <u>Explicit unification algorithm</u>

Prolog itself of course has its own underlying unification facility. For the proposed design operations however a unification algorithm is required to perform explicit unification. This algorithm is constructed using meta logical type predicates.

All unification is based on the elements of the input, output and port lists. Matching is performed on elements of similar types of material , then energy type and then geometric form type. The natural variables of Prolog are replaced by system variables identified by their context.

The hierarchical structure of functions, processes and applications is reflected by the use of a hierarchical reference scheme for both parameters, geometric forms and module attributes such as material : so that for example 'thrust:face1:axis' refers to the axis of the instance 'face1' of the bearing face process 'brg_face' included in the thrust bearing 'thrust'. Such objects are identifiable in the appropriate context as 'variable' types capable of instantiation with non-variable elements of compatible type.

Instantiation of this form causes the introduction of 'link' rules in the case of application matching. Non variable elements initiate Prolog equality checks when matched with other non-variables. 'Variable' types matched with other 'variable types cause the assertion of the appropriate equality rule relating the two variables. Equality between complex forms such as axes is achieved by the use of normal homogenised form representations.

Variant designs may be produced by backtracking. A mechanism to record the variant number with the data-base entry has been developed.

6.2 CONSTRAINT INTERPRETER

A constraint interpreter is implemented in the form of an extended expert system type backward chaining backtracking inference system. Module rules are in the form of mixed predicate logic and basic mathematical expressions.

6.2.2 <u>Constraint evaluation</u>

The logical expressions include implication, conjunction, disjunction and negation. The mathematical expressions include equality, the four inequalities , addition (subtraction), multiplication (division). The numerical representation is that underlying Waterloo Prolog and includes integer and floating point. However power (and inverse power) and exponential functions have also been implemented in Prolog using exponential and logarithmic series methods.

The inequality tests are implemented on a numerical test basis. The terms of the inequality are evaluated in terms of their name table entry and compared.

6.3.3 Interpreter mechanism

The logic interpreter is backward chaining and follows conventional expert system practice and builds a proof tree for recording the proof of a goal. An interactive facility allows particular 'askable' parameters and variables to be determined by interaction with the user. The user can defer response. A record is kept of the inference process so that this may be examined with 'why' responses to determine the rules whose conclusion the interpreter is trying to establish.

The mathematical extension determines variables by value in a variable name table. A list of goal variables is kept to avoid infinite looping after the same variable. The value of each variable is pursued in a left to right fashion. When a particular variable is required it is matched against the head of further expressions with backtracking on logical conditions.

6.3.4 Explanation facilities

A novel mechanism for selected explanation in the interpreter allows the designer to filter the rules used in explanation. Constraints may be made members of (possibly many) 'constraint sets'. This allows sets of rules to be hidden or displayed so that particular types of rules eg. geometric, dimensional, functional arithmetic etc. may be selected for exclusion or display. This mechanism is separate from the hierarchical module facility.

6.3.5 Application of interpreter

A principal means of identifying conflicting rules is through backtracking with the interpreter to find if an alternative (contradictory) value exists for a variable. This sytem can be regarded as a backtracking generate and test constraint programming language system where the backtracking is under the control of the designer and the generate mechanism is simply that of explicit designer input.

The processing requirements of functions are key objectives in a design. These checks are a principal application of the interpreter. Thus in the primary tube-centre function 'center_f' above it is for example necessary to establish that the design is rigid in the X and Y directions and thus that

v_x equals zero

and

v-y equals zero

that the radial and axial forces are free to assume any value and thus that

f_x

and

f_y

cannot be infered from the constraint rule data-base etc.

6.3 GEOMETRY AND GRAPHICS FACILITIES

6.3.1 Implementation details

The graphics interface is currently implemented on a Tektronix 4107A graphics system.

The next application of graphics is the facility for the designer manipulation of graphic symbols to express the sequences of the block diagram approach to the conceptual stage. This process may however also be carried out by non-graphical interaction using Prolog goal commands.

6.3.2 Graphics interface

A novel approach to the embodiment stage which is in under test and refinement involves the designer input of 2D non-metric geometric relational data to describe the completion of the form design. The approach is based on a menu driven interface Figs[10]. After the conceptual design stage the derived design is initially displayable in the form of the effective surfaces and geometric forms including points, tracks axes etc. Fig[11] of the suggested process application modules.

The design engineer attempts to complete the form design by specifying the non-metric relational geometric data with the aid of the menu. When the data-base geometric relations no longer form an underdetermined set, as determined by the geometric analyser (Sec 7.3.3 below), then the system can draw a realisable form Fig[12] by means of the geometric concstructor . The result may then be accepted or modified by the designer.

6.3.3 Geometric analyser

The results of the embodiment stages are the assertion of additional geometric rules and relations. These are in the standard format of two-dimensional variational geometry Ref[7]. The key requirement is the determination of underdetermined or overdetermined geometry. The logic programming facility uses Euler's method for a quick check for under-determined conditions.

6.3.3 Geometric constructor

A standard forward chaining interpreter for variational geometry Ref[7] provides for geometry completion and display. This is obviously invoked only after the design has been determined as being no longer geometrically underdetermined by the geometric analyser.

6.3.4 Solid analyser

A 2D polygon cliping algorithm has been implemented in Prolog. The potential of this method is that 2D solid and geometric collision analysis can be conducted within the auspices of a meta-interpreter. It is proposed that this offers the potential for geometric explaination in solid analysis in the manner of traditional expert systems.

Application of this facility indicates that it is reasonably fast for the moderately complex shapes tested.

7 ILLUSTRATIVE EXAMPLE

7.1 Primary function

For illustrative purposes consider the outline development of a conceptual and embodiment solution for the tube-centre with primary function Fig[13] as defined in Sec 5.2 above.

7.2 Design stages

A trial matching of processes against the primary function finds that a close but inexact match is found with the process 'absorb' Fig[14]. A process application 'absorb_a' is accordingly generated with the appropriate application data structure Fig[15] (see Sec 5.4 above). Only three of the process applications are used. The residual function 'rest_1' is now generated by removal of the 'absorb_a' application Fig[16].

```
/*----------------------RESIDUAL FUNCTION----------------------*/
/*                            rest_1                           */
/*-------------------------DEF SECTION-------------------------*/
f*(rest_1,1,center_f,1,absorb_a,1).
/*-------------------------I_O SECTION-------------------------*/
f*(rest_1,1,in(
  [center_f:mat,
     [force_vel(center_f:f_x,center_f:v_x),center_f:axis_x],
     [force_vel(center_f:f_y,center_f:v_y),norm(center_f:axis_x)],
  [torque_speed(center_f:t_x,center_f:s_x),center_f:axis_x]])).
f*(rest_1,1,out(
  [absorb_a:gnd,
     [force_vel(absorb_a:f_x,absorb_a:v_x),absorb_a:axis_x],
     [force_vel(absorb_a:f_y,absorb_a:v_y),norm(absorb_a:axis_x)],
     [torque_speed(absorb_a:t_x,absorb_a:s_x),absorb_a:axis_x]])).
/*------------------------RULE SECTION-------------------------*/
f*(rest_1,1,rule,connect(mat,mat2)).
f*(rest_1,1,rule,free(center_f:f_x)).
f*(rest_1,1,rule,zero(center_f:v_x)).
f*(rest_1,1,rule,free(center_f:f_y)).
f*(rest_1,1,rule,zero(center_f:v_y)).
f*(rest_1,1,rule,free(center_f:t_x)).
f*(rest_1,1,rule,free(center_f:s_x)).
f*(rest_1,1,rule,is_axis(center_f:axis_x)).
f*(rest_1,1,rule,free(absorb_a:f_x)).
f*(rest_1,1,rule,zero(absorb_a:v_x)).
f*(rest_1,1,rule,free(absorb_a:f_y)).
f*(rest_1,1,rule,zero(absorb_a:v_y)).
f*(rest_1,1,rule,free(absorb_a:t_x)).
f*(rest_1,1,rule,zero(absorb_a:s_x)).
f*(rest_1,1,rule,is_axis(absorb_a:axis_x)).
```

Where it should be noticed that the input

```
                free(center_f:s_x)
```

is required and the output

```
                zero(absorb_a:s_x)
```

is required.

A trial matching of processes against the residual function finds that 'collar' is an exact match for the 'connect' sub-function. The next residual function is determined by removal of the application of 'collar'. This simply results in a residual function 'rest_2' wit the

```
                connect(mat,mat2)
```

requirement removed.

A trial matching of processes against the residual function finds that no processes can complete the function. Although both 'radial' and 'thrust' match inputs and outputs the functional requirements are shown not to be met by the interpreter.

The strategy of increasing the number of function paths but decreasing the process tractibility indicates the use of 'joint'. ('Joints' are known to be equally useful for reducing the number of function paths). Matching of 'joint' against the residual function produces an exact input match and an application of the 'joint' process (which has one input set two output sets). The residual function is determined by removal of the application of the 'joint' process.

The collar and joint process having been selected they may now be unified Fig[17]. This process results in the setting up of compatibility or 'link' rules Ref[4] typically of the form:

```
                collar_a:face2:mat equals joint_a:mat
                collar_a:face2:f_x equals joint_a:f_x_1  etc.
```

Trial matching of 'thrust' and then 'radial' with the residual functions produces exact input matches and generate the corresponding applications of the two process. The residual functions are determined.

The matching of 'joint' against the residual function produces an exact match. The first two port sets of 'joint' produce an exact input match with the outputs of the thrust and radial applications in the residual function. The remaining port set produces an exact output match with the remaining inputs of absorb still in the residual function.

The interpreter now shows that requirements for connection between the input medium and the centre, for 'free' forces, 'zero' linear velocities, 'free' torque and 'free' rotational speed are now met.

The primary function is now satisfied by the conceptual solution Fig[18].

Directed backtracking in the matching of the thrust and radial processes to the their appropriate residual functions results in four variant designs Fig[19]. Clearly two of these alternatives require the thrust bearing to be double acting. This may be accepted or we may conclude that we have identified an additional constraint and require that the axial force be positive only.

8 CONCLUSIONS

A systems engineering design philosophy and representation scheme allows the formulation of the primary requirements of an engineering design. Logic programming facilities implemented in Prolog enable the designer to find conceptual physical solutions for these requirements by matching the required function against the representation of physical processes stored in the system data-base. The facility recommends how the physical process should be applied to solve the requirements. Variant solutions may be generated by backtracking.

Constraints may be identified and resolved with the aid of a backtracking backward chaining expert-system type interpreter. All the requirements of the required functions may be validated with the constraint interpreter.

The resulting conceptual solution is stored together with a 2D representation of the effective surfaces and geometric forms essential for the working of the phsical process. The 2D geometric forms may then be completed during the embodiment design stage using an interactive graphics system. Only non-metric geometric relational data need be entered. The system can determine when the data is sufficient to complete the form of the object and can then display the result for acceptance or ammendment.

The effective surfaces and forms are protected by a hierarchical explanation facility which explains the purpose of the geometric features.

9 REFERENCES

1 CLOCKSIN.W.F. and MELLISH.C.S. Programming in PROLOG , Springer Verlag , Berlin , Heidelberg , New York , 1984 .

2 Waterloo Core Prolog User's Manual, Version 1.6, Intralogic Inc., Waterloo, Ontario.

3 PAHL,G. BEITZ,W.,Engineering Design,The Design Council, London, 1984.

4 SHENTON.A.T.Computer integration of geometric modelling and engineering design systems by rule based programming,3rd International Conference on Effective CAD/CAM,November 1987, pp57-68,I.MechE. London.

5 ARBIB.M.A. (Ed) Algebraic theory of machines, languages and semigroups,Academic Press,New York,1968.

6 ZADEH.L. DESOER.C.A.,Linear system theory, McGraw-Hill,1963.

7 ALDEFELD.B. Variation of geometries based on a geometric-reasoning method, Computer-Aided Design. Vol 20, No 3, 1988, pp117-126

8 deKLEER.Choices without backtracking,Proc.Conf.Amer.Assoc.for AI,pp79-85.

9 STALLMAN.R.M. SUSSMAN.G.J.Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis,Artificial Intelligence,Vol 9,pp135-196.

10 BARROW.H.G. Proving the correctness of digital hardware designs, Proc.Nat.Conf.on AI, 1983.

11 POPPLESTONE.R.J, AMBLER.A.P. BELLOS.I. An interpreter for a language describing assemblies,Artificial Intelligence, Vol14 , No 1,pp79-107.

12 AITCHISON,I.E and WILKIE,G.A.R. An Expert System for the Design of Engineering Assemblies.International Conference on Computer-Aided Production Engineering, Edinburgh, IMechE, April 1986 .

13 CHAN.W.T. and PAULSON.B.C. 'Exploratory design using constraints', A.I. for Engeering Design Analaysis and Manufacture, Vol 1 No1 ,pp59-71. 1988.

14 KOLLER,R.Konstruktionslehre fur den Maschinenbau , Springer-Verlag. Berlin, Heidelberg, New York , 1985 .

15 LEYER.A ,Machine Design, Blackie, Glasgow

Physical Process	Rotational Inclined Surface
Name	'rev-face'

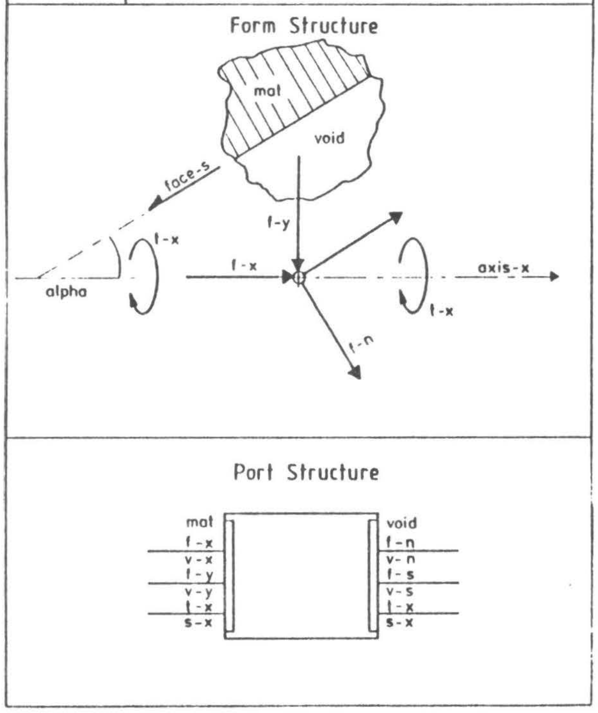


FIG. 1

Physical Process	Rotational Inclined Bearing Surface
Name	'smooth'

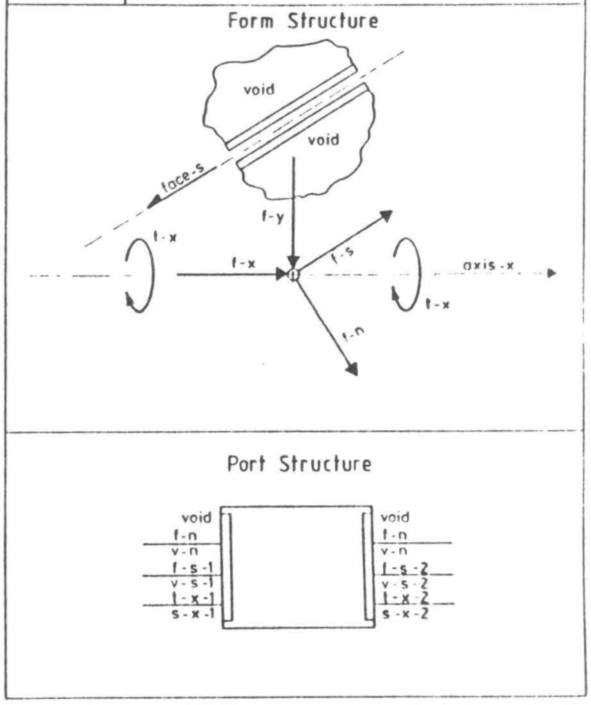


FIG. 2

Physical Process	Radial Bearing
Name	'radial'

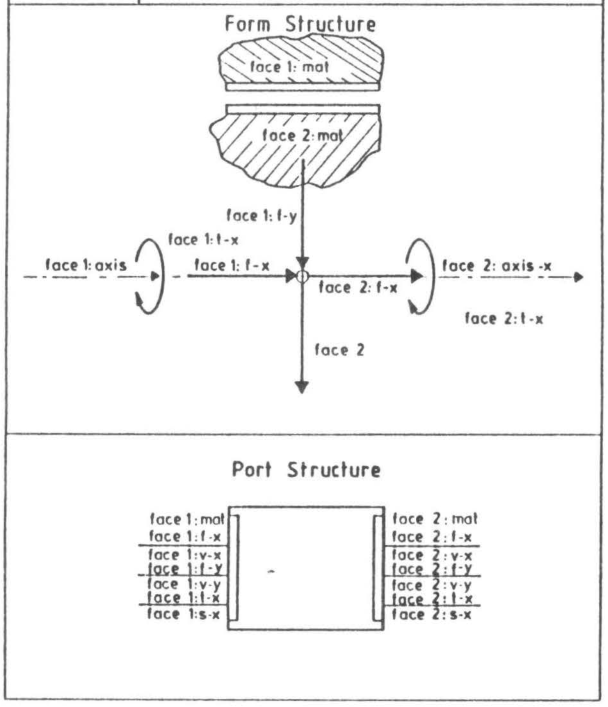


FIG. 4

Physical Process	Rotational Inclined Bearing Component
Name	'brg' face

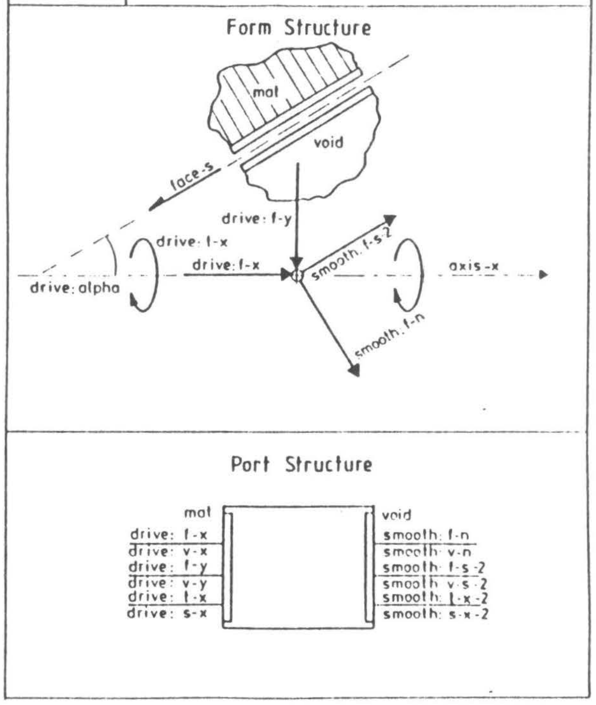


FIG. 3

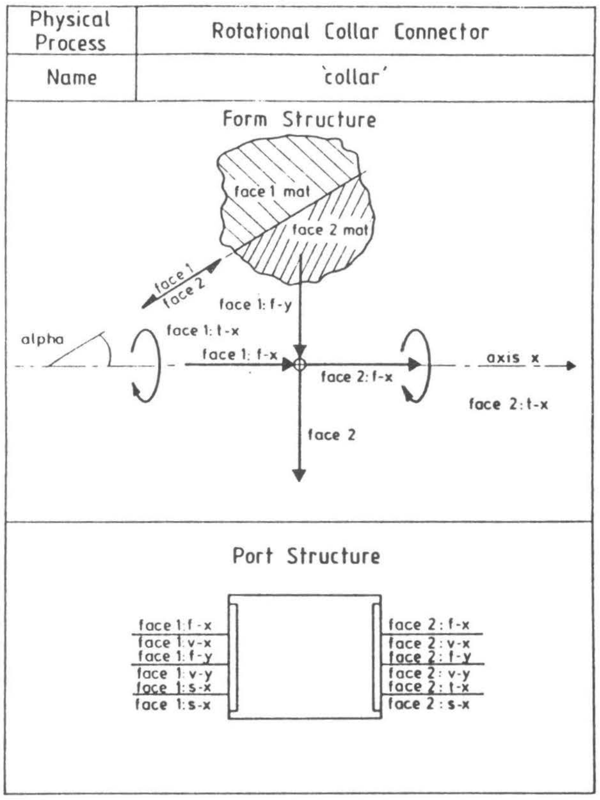


FIG. 5

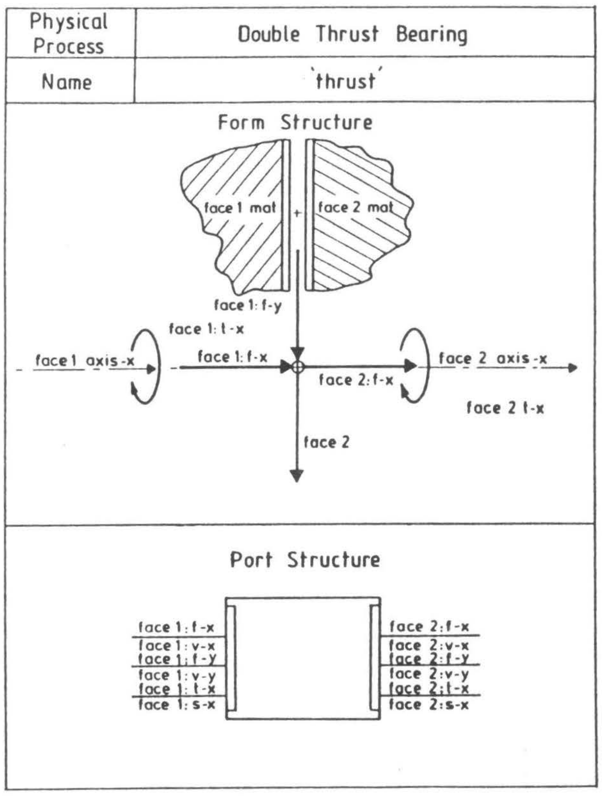


FIG. 6

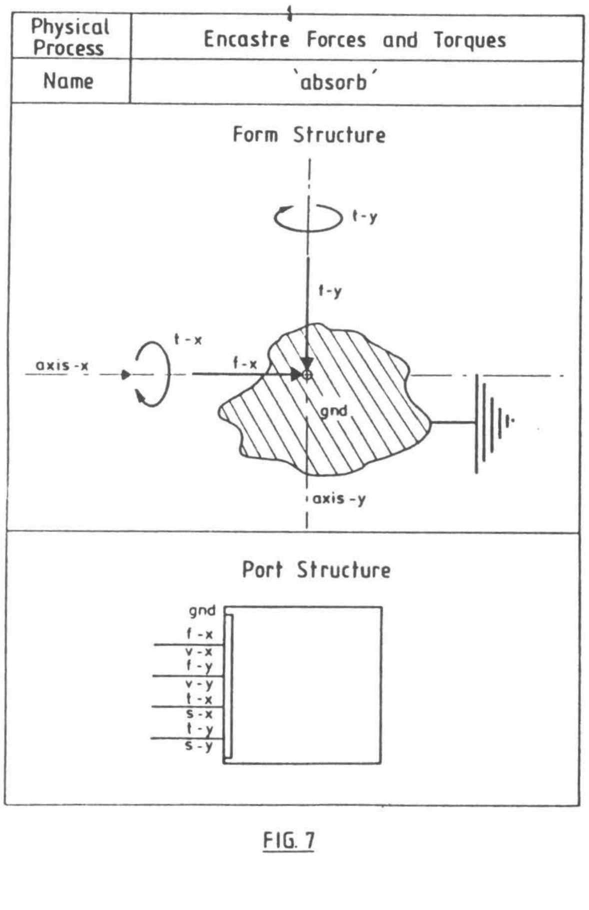


FIG. 7

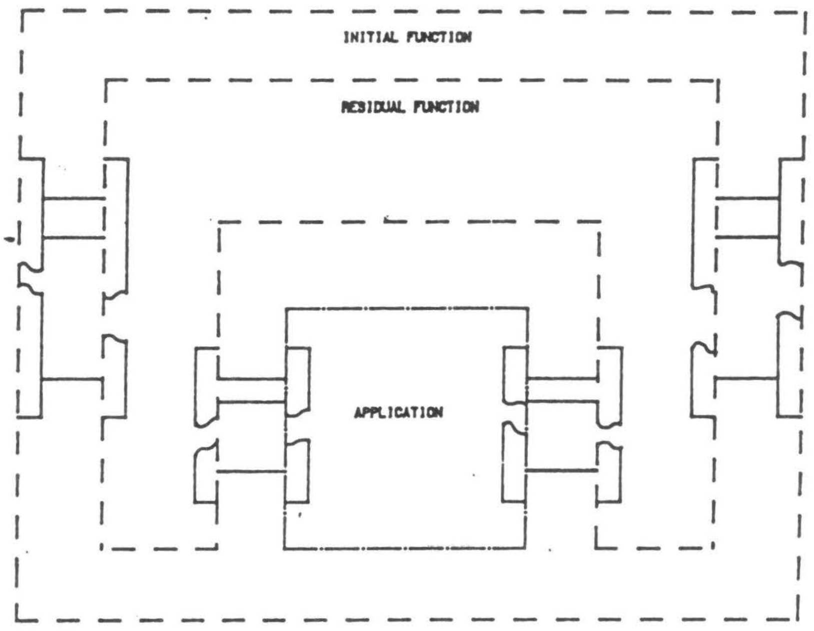


FIG.8 General structure of a residual function

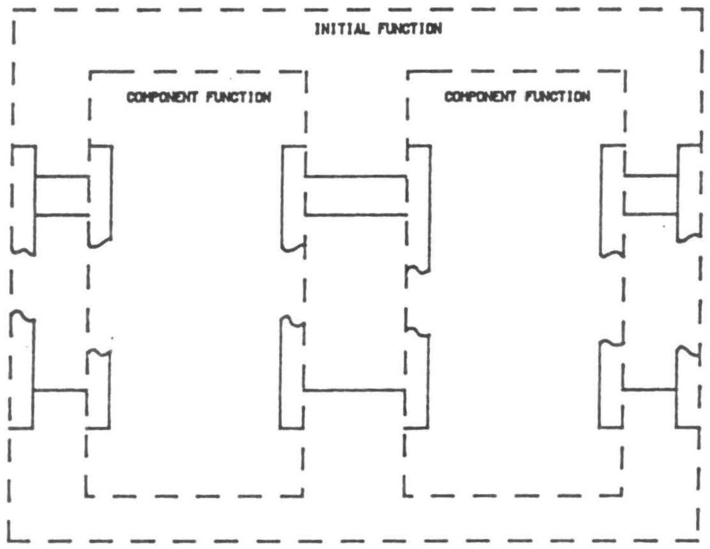
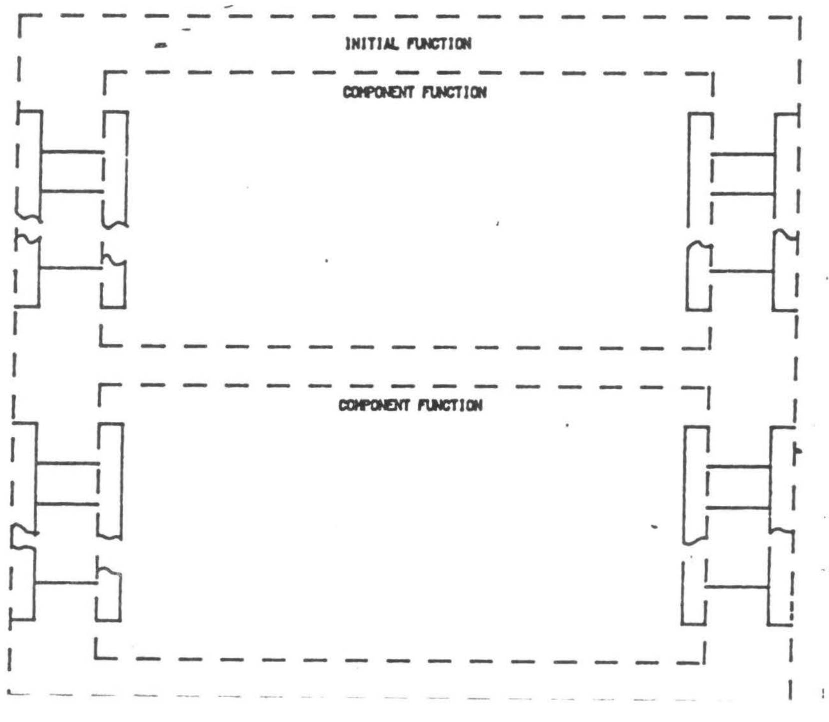
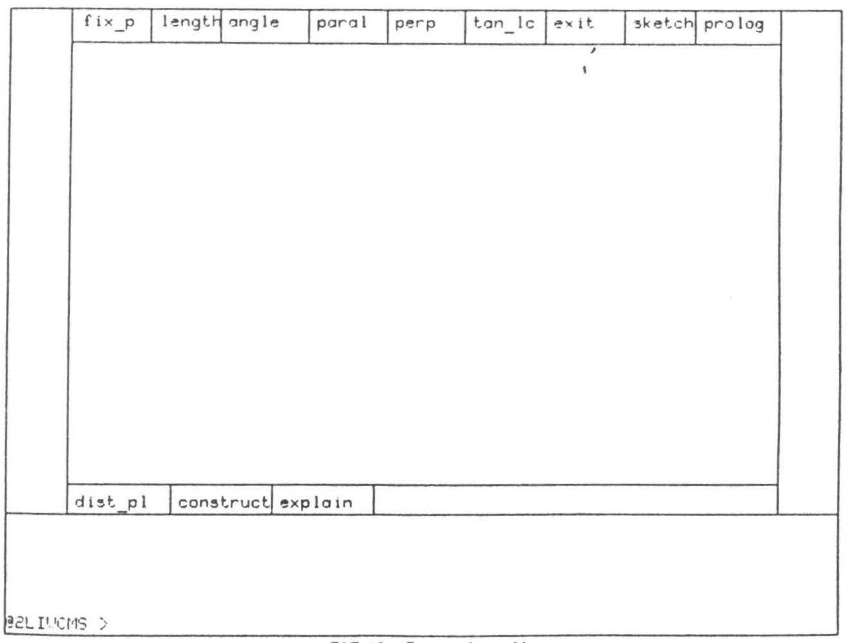


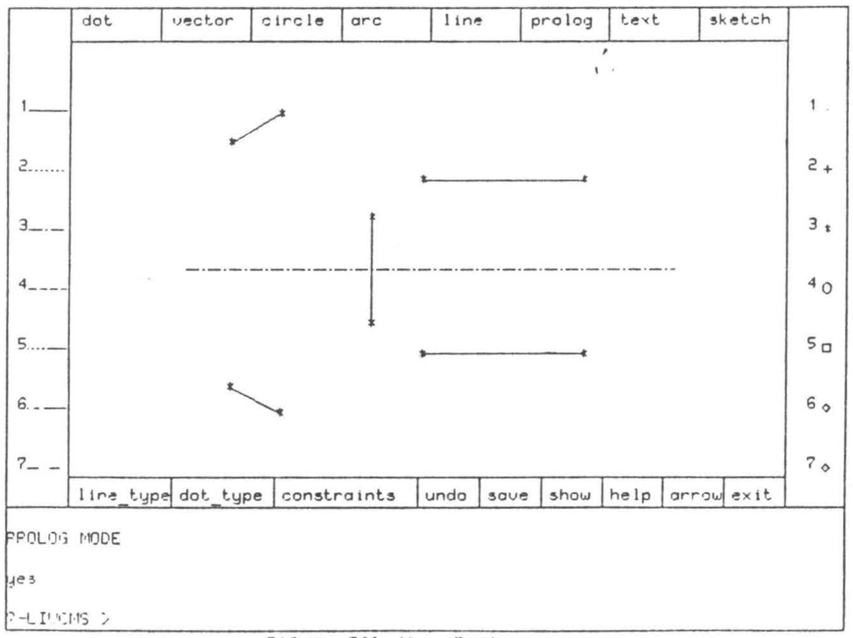
FIG.9a Series decomposition of a function





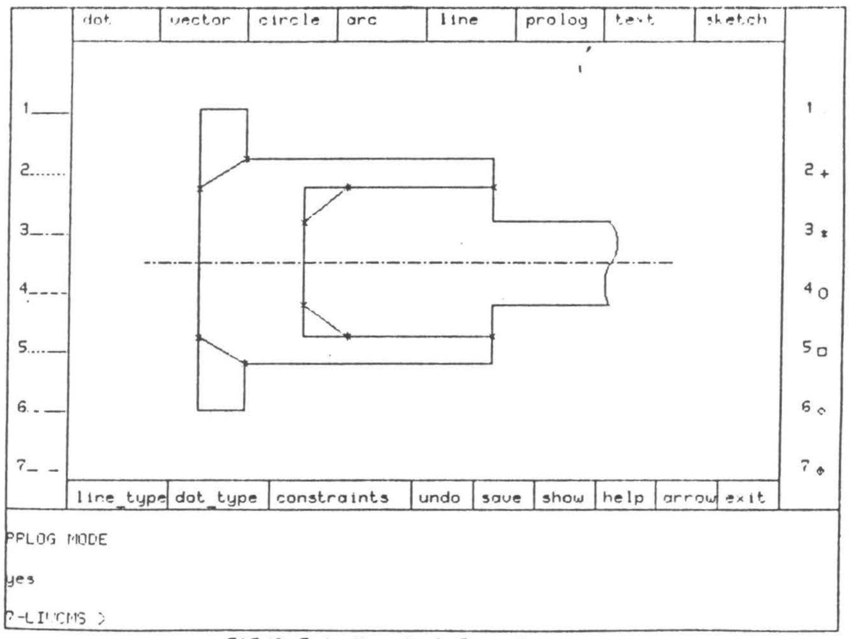
```
@2LIVCMS >
```

FIG. 10 Geometry Menu



```
PROLOG MODE

yes

?-LIVCMS >
```

FIG. 11 Effective Features



```
PPLOG MODE

yes

?-LIVCMS >
```

[illegible]

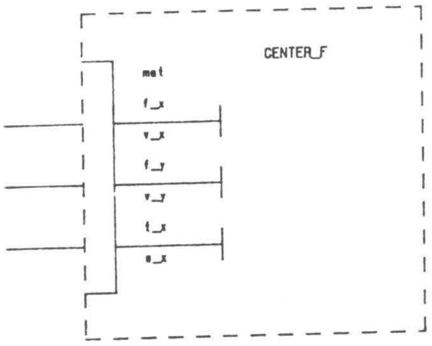


FIG.13 The primary function 'center_f'

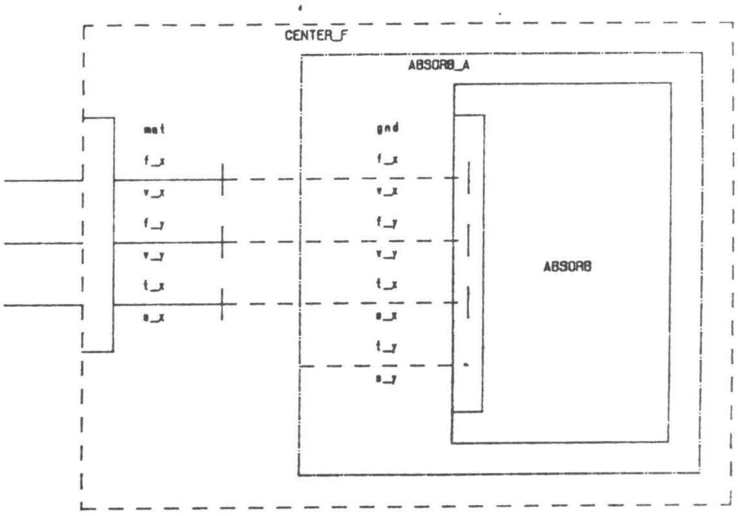


FIG.14 Generation of the first application

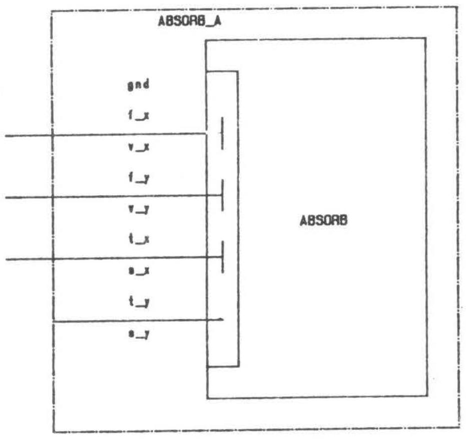


FIG.15 The application data structure

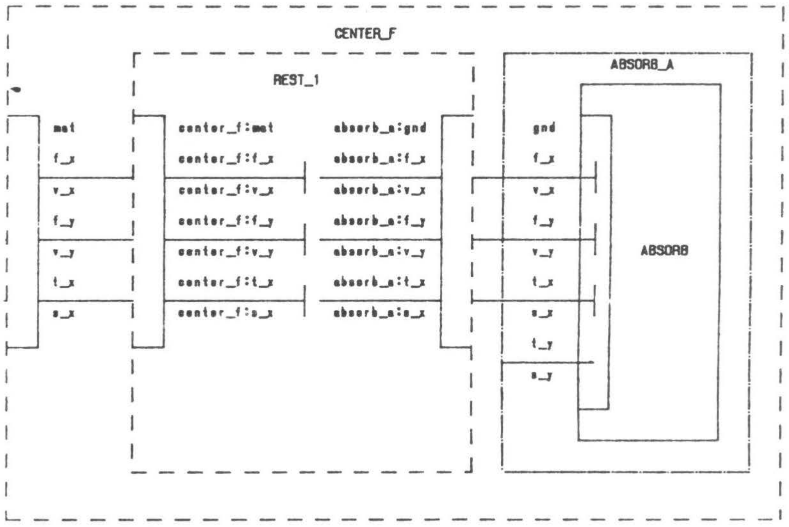


FIG.16 Generation of the first residual sub_function ,

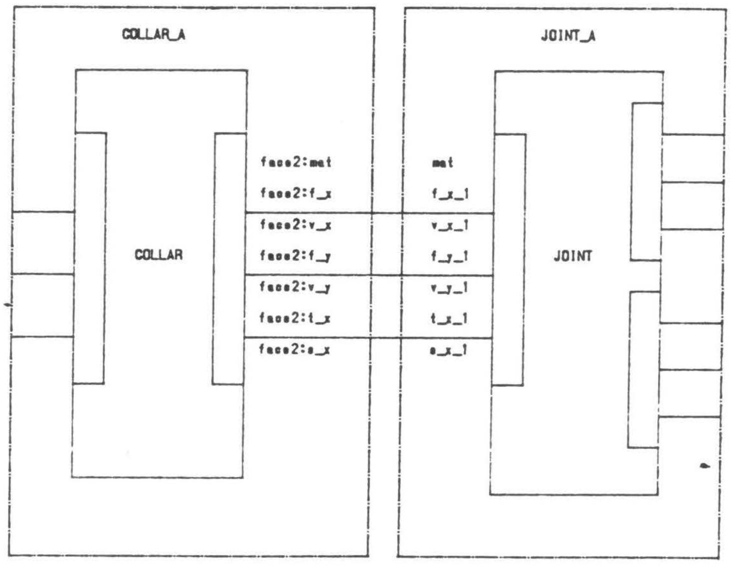


FIG.17 Unification of applications

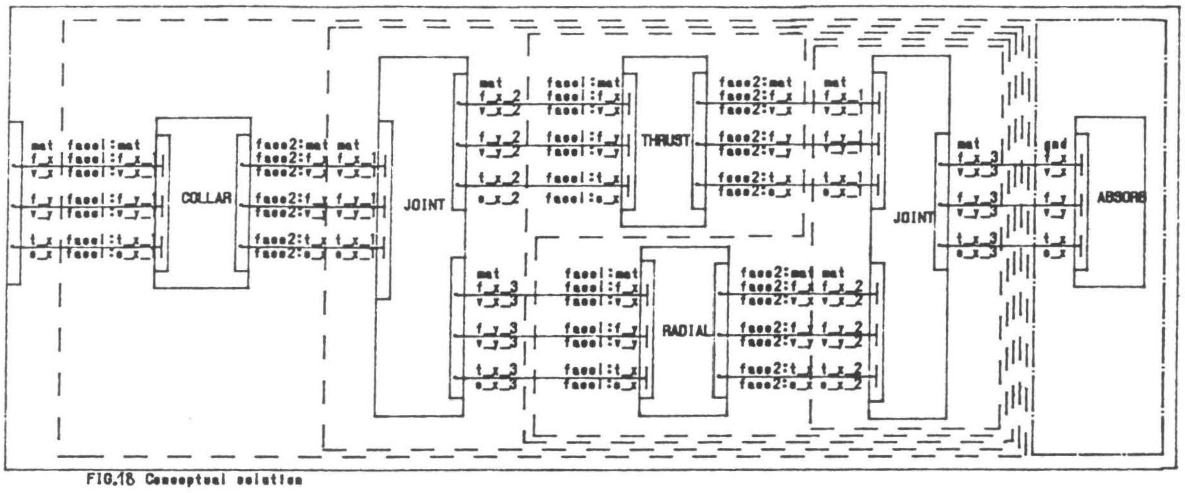


FIG.18 Conceptual solution

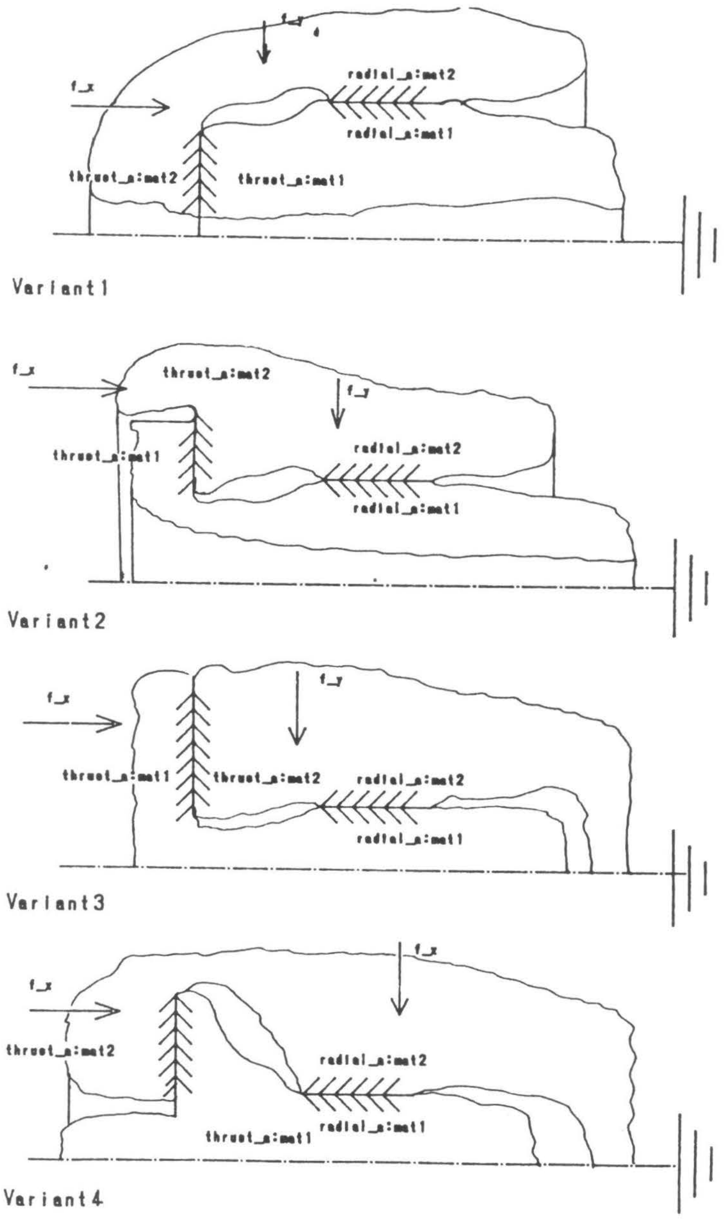


FIG.19 Variant embodiments

Tapio Takala

Application of history mechanism in architectural design

Third Eurographics Workshop on Intelligent CAD Systems,
3-7 April 1989, Texel, The Netherlands

APPLICATION OF HISTORY MECHANISM IN ARCHITECTURAL DESIGN

Tapio Takala
Helsinki University of Technology
Department of Computer Science
SF-02150 Espoo, Finland
phone: +358-0-4513222, e-mail (uucp): tta@hutcs.hut.fi

Pär Silén
Technical Research Centre of Finland
Laboratory of Urban Planning and Building Design
Itätuulenkuja 11, SF-02100 Espoo, Finland
phone: +358-0-4564567, telefax : +358-0-464174

Abstract: An experimental CAD system for conceptual design, called MetaViews, is described. It is intelligent in the sense that knowledge about relations (especially about causal dependencies) between design objects is stored and utilized. Based on its history mechanism, containing all steps of the design process, MetaViews facilitates flexible interactive experimentation and variation of geometric forms. A case study about designing small houses with the system is described. The system's capabilities and limitations in this task are evaluated. Further development is outlined based on a fictive continuation of the same design.

1. Background

With conventional CAD systems it is possible to construct a design product model, which can be analyzed and visualized by computer. As a product document a single model of the final design may be sufficient, but in the conceptual ideation phase designers often return to their previous designs, reusing and modifying earlier ideas. Then it is important to have an explicit model of the design *process* [Takala86], containing such metalevel knowledge about the design as steps through which a design was done and reasons why each decision was made in a particular way. Based on this observation, we have developed a prototype CAD system called MetaViews [Takala88a, Takala88b]. It has so far been applied, but it is not restricting, to artistic industrial and architectural design - in particular to the early, creative sketching and ideation phases, where flexibility is of utmost importance.

A description of how an object is designed can be captured with a *history mechanism*. Perhaps the simplest way is to take sequences of commands from a log-file stored during an interactive design session. When modified and re-executed, such a sequence acts as an automatically formed parametric model for variational design [Takala85]. In order to be really useful in designing, the history should be well-structured. Also we need a good user interface with informative representations and practical tools to manipulate the history.

There are several ways to implement a structured history. The causal dependency relations between phases of design can be understood as logic predicates or parametric transformations, and the history mechanism can be implemented with logic programming [Yamaguchi87] or with special language constructs [Rossignac88], respectively. In our system, the design process is modelled as a directed network consisting of *design transactions*, each of which is a frame describing a discrete design operation. The transactions are partially ordered by links representing dependencies between them.

The MetaViews system can form the history network automatically *bottom-up* by collecting into the history the operations performed by the designer. Alternatively the user can *plan* it directly *top-down* with abstract metadesign operations. *Retrospective planning* combines both methods: a rough network is first formed by example, and is then refined with abstract planning. This way both concrete and abstract intentions of the designer can be captured by the system.

A design often consists of many *subdesigns* (e.g. parts of an assembly), each of which can be separately specified and solved. These are represented with subnetworks or structured transactions, which may be nested hierarchically.

The transaction network is a *metamodel* of the design, describing with relations the process how design objects are formed. The system is *process-centered*, meaning that not only the design objects themselves but also processes describing their construction (*metaobjects*) are handled in an object-oriented way. Variational design can be seen as data flow programming: just put new parameters in, re-executing the process, and collect new design variations from the systm's output! The same approach is also utilized in the system's user interface: from any design and/or analysis operations the user can compose generalized viewing processes, called *monitorsπ*. They automatically perform the specified transformations and conversions of models, reducting them and supplying additional information as is necessary to represent the results and/or parameters of design in the most appropriate form.

In the following (chapter 2) we describe how these problems are technically handled in our system. Then (chapter 3) the system's flexibility in architectural design is demonstrated with examples. And lastly (chapter 4) some potential issues of further development are outlined.

2. Features of the MetaViews System

MetaViews is an experimental CAD system mainly intended for sketching purposes. It is not comprehensive for final product documentation with sophisticated rendered pictures or fully dimensioned drawings. It rather acts as an idea processor, in which a raw model can be quickly produced and then transferred to other systems with better detailing facilities. It provides a new dimension to conceptual designing by making the design process explicitly available to the user.

The system has been implemented with C language in the Apple Macintosh computer, which provided an easy environment for building graphical user interfaces.

The basic design tools of MetaViews are similar to those of any other CAD system. It is possible for example to interactively input and reshape polylines and polygons, to round their corners, to grow or shrink polygons with a given offset, or to perform 2-dimensional set operations (union, intersection, difference) on them. Bezier and B-spline curves can be evaluated from a control polyline as approximating polylines with a precision given by the user. Three-dimensional surfaces and solids can be produced by sweeping (translation or rotation) a two-dimensional object in space, or by trimming (cutting by plane) another 3-dimensional object. Objects can be copied, and pasted repositioned with linear transformations (translation, rotation, scaling). Queries about objects' properties (dimensions, area, volume, etc.) will produce text objects, which can be positioned into the design - a feature yet uncommon in CAD systems.

2.1. The history mechanism

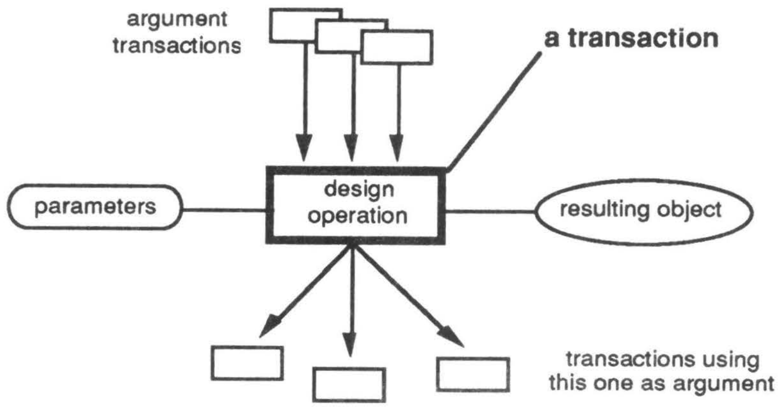


Figure 1: A transaction frame and its associations.

While performing a tool operation, the system automatically stores a transaction frame describing the tool command and its numerical parameters (Fig. 1) into the history structure. Such a frame exists for each past or present design object, thus making a one-to-one correspondence between objects and design operations (they are just different facets of the same entity). Each transaction also has links to all argument transactions whose resulting objects were used by the operation. Also a corresponding inverse link is added every time the transaction is used as argument by a new operation.

All transactions together form the history of a design. It is a data base with network structure formed by the links associating frames to each other. These associations are utilized as access paths to objects within a design project. Following them backwards, it is easy to find either a transaction's (for example, no.4 in Fig. 2) immediate arguments (no.3) or its whole private history, i.e. all antecendant operations upon which the object is directly or indirectly dependent (nos.1 to 4). Following forwards, we can similarly find the operation's (no.4) immediate users (nos.5, 11 and 11a), or all descendant objects dependent on the operation (nos.4 to 7, 11 and 11a).

The history of an object, i.e. the subnetwork formed by the corresponding transaction and its all antecendants, is a parametric metamodel. Changes made to any of its parameters are automatically propagated - through intermediate stages - to the object itself, and further to all its descendants. If a design with the same logical structure but with parametric variations is utilized several times, it can be defined as a macro operator. The object's history is extracted and a separate copy of the subnetwork is instantiated for each variation. For example in Figure 2, history of the chair's seat part (nos.8 to 10) is copied and some form parameters changed (nos.8a to 10a), yielding another chair with a different seat but legs similar to the original (no.11a).

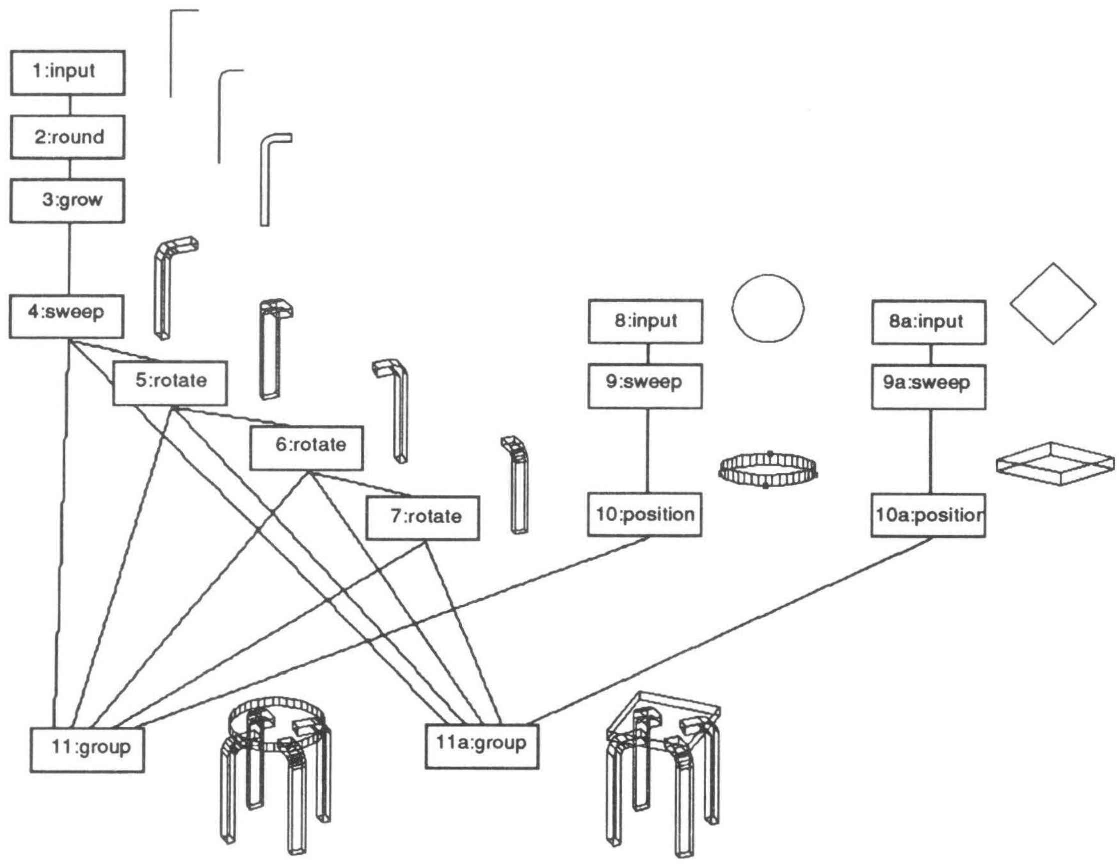


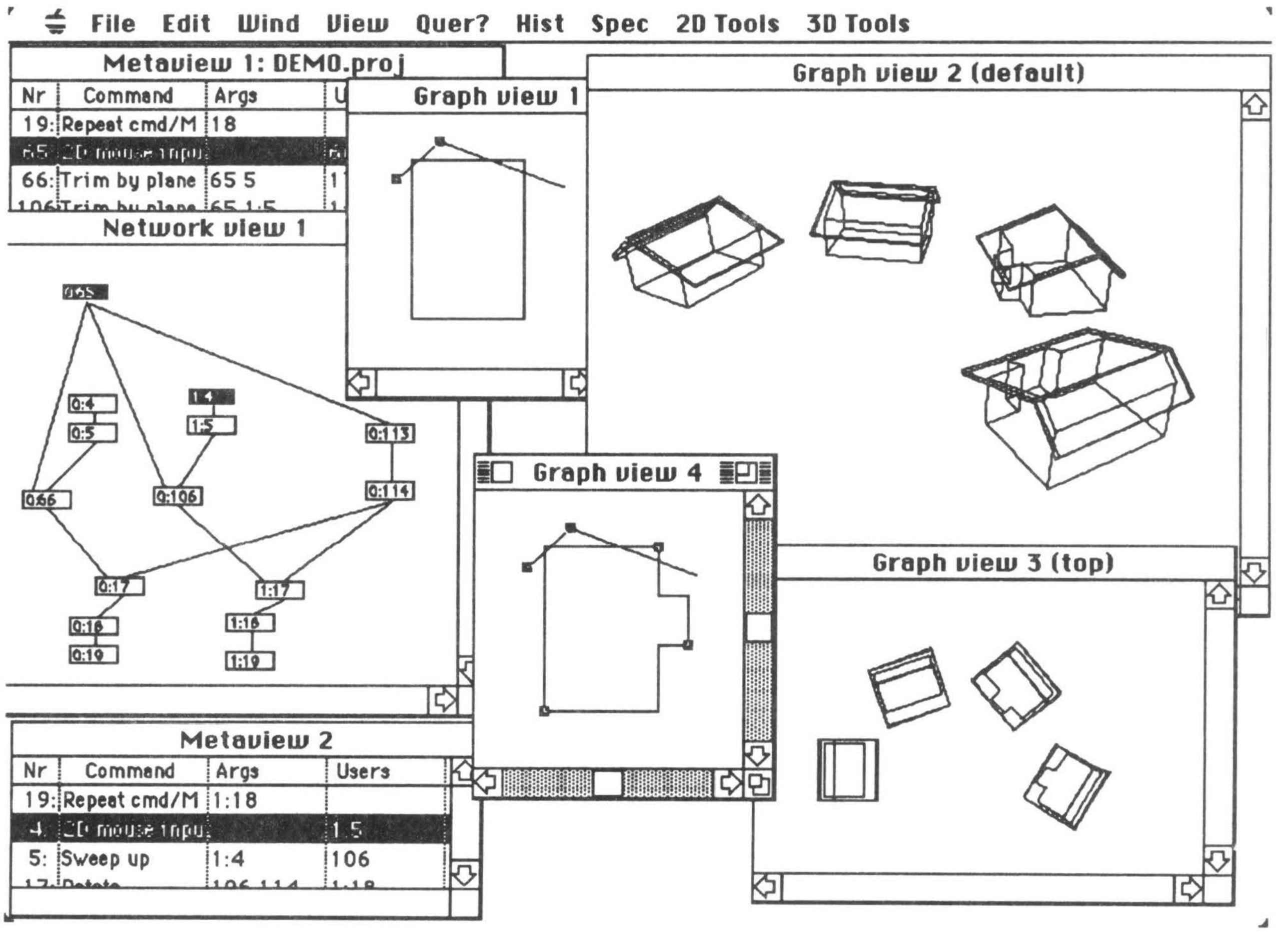
Figure 2: Design process of a chair with partial variations.

The history can be manipulated in MetaViews with metadesign commands. Like usual design commands, these are executed by pointing graphically an object (in any view) and then selecting the command from a menu. However, they don't usually cause new entities to be added in the history, but rather affect the network's structure or the contents of its individual transactions. Metadesign commands for manipulating a design's history network are among others:

- *ExtractDerivation*: find from the history all antescendants of a selected object (or operation).
- *Collapse*: collect the selected nodes of a network into a group, which will be shown as one macro operator in the original network. Its contents can be looked at in another view.
- *Duplicate*: create new transactions in the network, repeating the same operation with the same arguments as had those selected for copying.
- *VariateParams*: modify some parameters of a transaction, thus variating the resulting object.
- *ChangeArguments*: make an operation to act on different arguments, by redirecting the links pointing to argument objects. The network structure will thus be changed.
- *Re-evaluate*: redo a command whose argument objects or input parameters have been changed. Re-evaluation is done automatically for the decendants of a modified transaction, which are affected by the changed one.

2.2. Interactive manipulation of designs

Entities can be seen on the screen in different views. A *view* means a process consisting of a window on the screen, and methods with which objects are *projected* into the window or are manipulated by input actions (projection refers here to any conversion or transformation of objects, not only to geometric projections). The number of simultaneous views is unlimited, and each may have a different projection (Fig.3). A group of different views to the same objects (a multiview) is useful for example in monitoring interactive manipulation of three-dimensional objects.



Metaview 1: DEMO.proj

Nr	Command	Args	U
19:	Repeat cmd/M	18	
65:	2D mouse inpu		6
66:	Trim by plane	65 5	1
106:	Trim by plane	65 1:5	1

Metaview 2

Nr	Command	Args	Users
19:	Repeat cmd/M	1:18	
4:	2D mouse inpu		1 5
5:	Sweep up	1:4	106
17:	Rotate	106 114	1:18

Figure 3: A typical screen lay-out of MetaViews, showing different views (about how the houses are designed, see chapter 3.2).

Selected objects from a history can be looked geometrically in a graphical view. A metamodel (the whole project's history or an extracted part of it) can be seen through a specific *metaview* either as a network emphasizing graphically the dependency links, or in textual spreadsheet format showing more information about each transaction. At any time, the set of active entities is shown by highlighting corresponding objects and transactions in all views.

Manipulation of entities of a history network is based on the *hypermedia* concept [Conklin87]. Hypermedia facilitates metaphorical communication, allowing the user to refer to design objects via symbolic representations. An entity is selected by pointing to its representation within a view, and its associations to other entities can be followed simply by clicking a button. This kind of interactive navigation within a CAD data base, utilizing active iconic symbols with attached associative knowledge, is termed *HyperCAD* [Vanier88].

Double-clicking a transaction in metaview or a geometric object in graphical view will open it, representing either its internal structure in a metaview (if it is a collapsed macro operator) or its numerical parameters in a panel where they can be edited. The dependencies between transactions (the links to and from arguments) can be modified by moving them graphically in a network view.

Traditional bottom-up *construction* operations of CAD, performed by selecting argument objects in views and tool operations from menus, will automatically produce new transaction sequences in the history. In order not to loose information of the whole design process, every step is stored separately. When an object is changed, its previous form is not destroyed but a new modified copy is added in the history. Despite of that, the user may have a feeling of directly manipulating the object, as its graphical representation on screen is immediately replaced with the new version.

Alternatively, we can *plan* a design process top-down by pasting so far unspecified operations on a network in metaview. Their exact connections to other operations and the values of their parameters can be specified later. An operation may even be an anonymous macro, which is later refined with an additional subnetwork.

The associations between entities are important hints when trying to capture the designer's intentions and way of thought [Makkuni87]. For this purpose not only the design objects but also constituents of the design environment (windows etc.) are considered entities in the history. When parameters of an operation are given graphically, they become dependent on the view in which they were given. If the view's projection is changed, the parameters may become different. (For example, a plane in 3-dimensional space is most easily defined by extruding a line, drawn in the projection plane, into the viewing direction). An additional benefit of storing the whole working environment together with the design process is the possibility to suspend an uncompleted design session and later resume exactly the same situation from a history file.

3. Experiences in Architectural Design

MetaViews is experimentally applied to architectural design as part of the project "Aesthetically Qualified Environment and New Design Technology", where new CAD techniques are surveyed and their effects on the quality of built environment is researched.

3.1. The nature of architectural design

The behaviour of an architect doing design is often described with the terms generate and test. According to this description of the process, the architect generates a set of solutions to the problem at hand and tests their properties in terms of functional and aesthetic qualities. The testing of different proposed solutions can either be immediate subsequent to generation, or can be postponed until a set of proposed solutions have been created.

In the above mentioned research project another important process in the task of architectural design has been identified. This process could best be described with the terms focusing, subdivision and substitution.

In any moment of the design process, the designer is focusing his attention on one problem or a limited set of problems to be solved. The term focusing is important in so much as the designer, while doing this, does not omit the context of the problem(s) at hands, but keeps it in his mind as a "fuzzy" set of criteria that affect the problem(s).

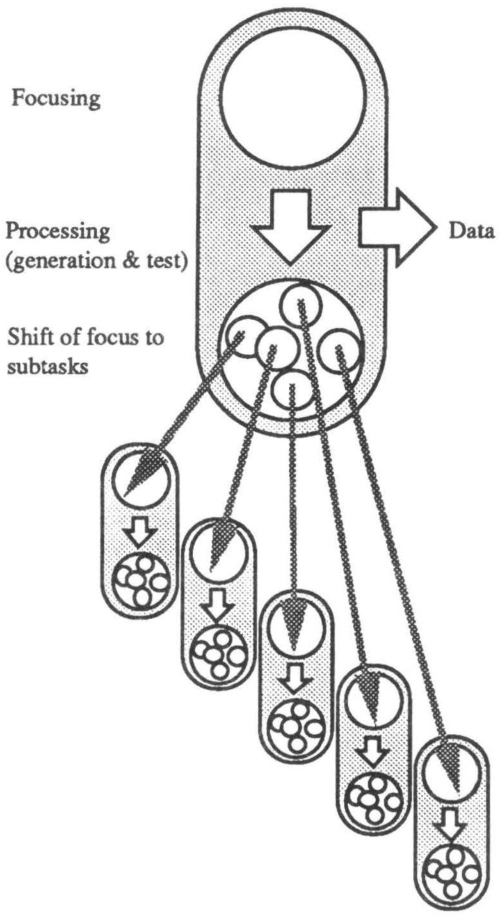


Figure 4: The "atoms of architectural design"

In solving the problem(s) that the designer is focusing his attention on, he usually performs two kinds of tasks. Firstly, he subdivides the problem(s) further into a set of smaller problems. Secondly, while doing this he defines the physical and functional limits of the solutions to these subproblems which, when solved and taken together, will form the solution of the actual problem he is tackling. Usually this definition of limits is done by sketching some lines on tracing paper and perhaps by making annotations in a notebook about special properties of these subproblems and proposed solutions.

The more experience the designer has, the rougher the sketches can be. An experienced designer knows how much space he must allow on the critical places to be able to later develop a functioning detail that solves the problem as a whole, while an inexperienced designer must take a bottom-up approach in every subtask. He must explore the subproblems in greater detail before he can determine whether the solution will fit into the limits of the context.

When the solving of a set of problems has reached a sufficiently detailed stage, the designer uses the solutions on all corresponding places. He substitutes the problems with the solutions.

It is, in our opinion, in this process of focusing, subdivision and substitution, that the generate and test pattern of behaviour is practiced over and over again.

We must emphasize, however, that this process by no means is a technical or rigid process of hierarchical subdivision. Mostly it is more like a game of associations and movements between different kinds of problems of different scales. We are convinced, that no optimal path can be scheduled, that would fit all kinds of design tasks, even if we would limit the scope of tasks to a minimum, e.g. the design of one-room log cabins. The actual order in which the designer solves the problems inherent in the design project will always be dependent firstly on the designer and his experience, secondly on the kind of artifact that is to be designed and thirdly on the context into which the artifact is to fit.

It is against this underlying view of the design process that the MetaViews system is tested. The main goal is not as much to test how well the tools implemented so far can be used to design, as how the underlying philosophy of the program could serve the design process. The actual test is described below.

3.2. The objective of the experiment

The system is used in the three-dimensional composition of building volumes. This early sketching phase of building design requires flexible modelling and easy modification of design proposals. It may start from scratch, or from a program of spaces made separately with an expert system. After having completed a sketch with MetaViews, the resulting model can be transferred into a drafting system (in this case MacDraw II or VersaCad), where the final detailed and annotated drawings can be produced.

The objective of the experiment is to design a group of small houses with a common basic form, but allowing variation in some features like dimensions of the plan, number of rooms, types of roof and windows, etc. Experiences are collected and used to evaluate the system's limits and usefulness of its tools.

Of special interest are the possibilities of using the history mechanism in the development of alternative solutions to the design problem. Assessing whether the use of the history mechanism is more or less efficient than the tools included in standard CAD packages was difficult and could not be fully performed.

3.3. Description of the design process

The design process started with the definition of the outlines of the plan of one building, by means of an *input* tool for 2D polygons (Figure 5).

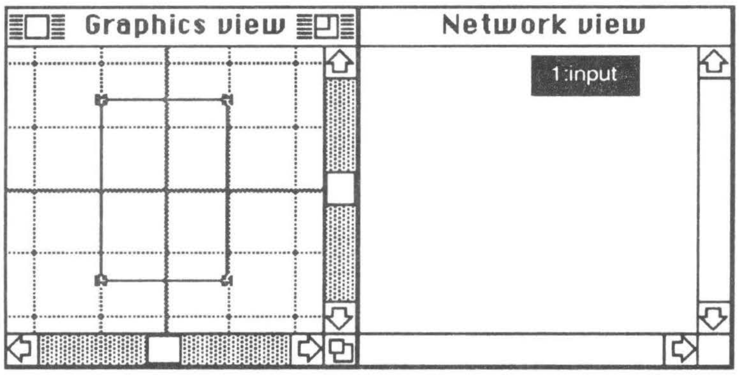


Figure 5: The initial plan form of a house and the corresponding design history network.

This polygonal form was extruded to a prism of a certain height by the *sweep* command, and the form of the roof was defined by two lines (Figure 6).

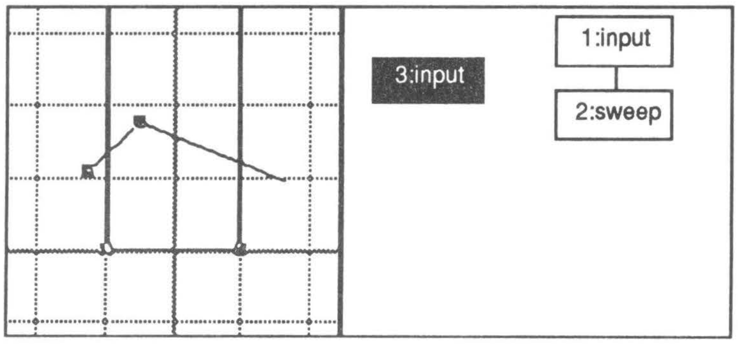


Figure 6: The building volume and the profile line of its roof, before the creation of roof surfaces.

The two roof lines are extended into the viewing direction, resulting in planes. These are used to *trim* the prism at the roof (Figure 7).

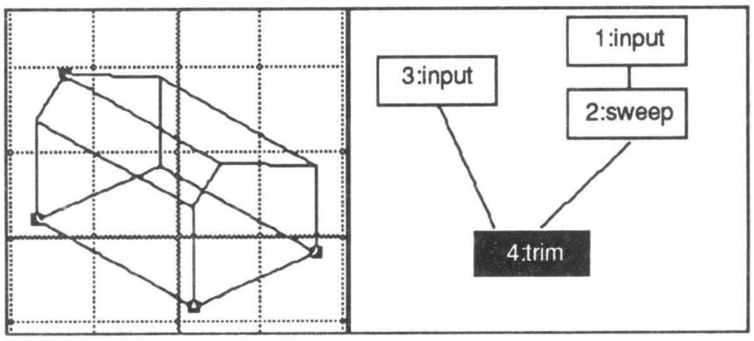


Figure 7: Trimming the building volume by the roof surfaces.

Thereafter thickness was added to the roof profile by the *grow* command, and the profile is swept in space for a given length, resulting in a fairly realistic wireframe view of a small building. This building was *copied, moved* and *rotated* a few times to create a group of identical houses.

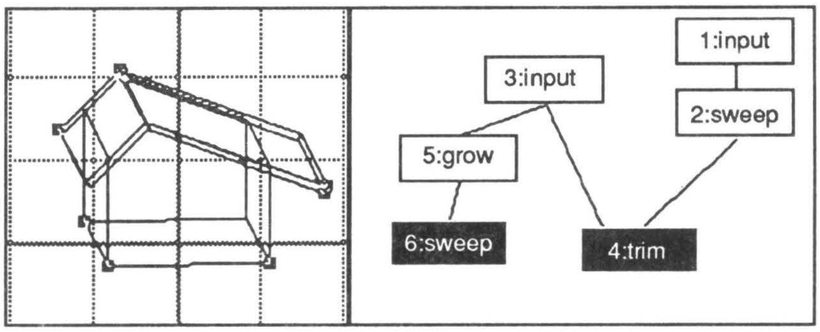


Figure 8: A roofed house and its design history (visible parts of the network highlighted).

After this, variation of houses begun with the aid of the history mechanism. The designer activated one of the buildings and *extracted* its design history in the metaview, and copied all the relevant stages of it. Among the commands in the history he activated the 2D-polygon input transaction, issued the *VariateParams* command and edited a rectangular bulge at one side of the building, depicting the adding of one room. After accepting the new form, all its descendants in the design process were *re-evaluated* using this new plan form. In a moment all the buildings, regardless of their placement, were updated on the screen, showing the same building volume (with the new room added) accurately trimmed by the old roof form.

This update did not, however, delete the old versions of the houses that existed in the history window, even though they were not shown on the graphic window. The designer activated and deleted (from view) a couple of the new designs, and activated the corresponding stages of the history of the original houses. By *copying* the right step in the history, and *pasting* it into the graphics view, he could instantaneously show the old versions between the new ones. In fact, all the versions were preserved and existed as alternative solutions on the same physical locations. The designer was able to choose, which of them he wanted to include in a view. The results are shown earlier in Figure 3 (note that in the 3-dimensional graphic views two buildings are shown from the original history, and two others from the modified branch).

In the next step of the process a new roof form was developed and applied to the two different versions of the buildings with the same success. The designer was able to freely swap between the four different kinds of buildings, retaining their physical locations.

3.4. A fictive continuation

In the following a simulated process is presented, in order to give an idea of the possibilities of MetaViews and problems that might rise due to an increased complexity in the design history network. The example makes use of two routines, *CreateWindow* and *CreatePane* that are not implemented in this version of the system. The idea is, that these routines would create windows and attach them to the wall they "belong to".

The fictive example would start exactly as the real example shown above in the Figures 5-8. After creating the intersection of the building volume with the roof planes, the resulting surfaces are selected individually, and the situation would be as in Figure 9.

Now the designer starts including windows in the design. In reality he would most probably locate the placement of the first window in the graphical wiew. After activating the right wall he would issue the commands *CreateWindow*, and twice the command *CreatePane*, resulting in the situation shown in Figure 10.

At this stage the normal procedure in traditional CAD systems would be to copy the window and paste it into the next place where an identical window is wanted. In MetaViews the designer can choose between doing exactly the same or duplicating the design sequence and applying it in a new context, as shown below in Figure 11.

The designer could also issue another command sequence, *Copy* and *Paste* (here for the sake of clarity shown as one box). These commands create a new copy of a group of objects (the window and the panes) and position the copy in a specified place, resulting in Figure 12.

Creating two more windows on the same wall could be accomplished by applying another *Copy&Paste* command on both the previous *Copy&Paste* command and the previous window creation commands, as is shown in Figure 13.

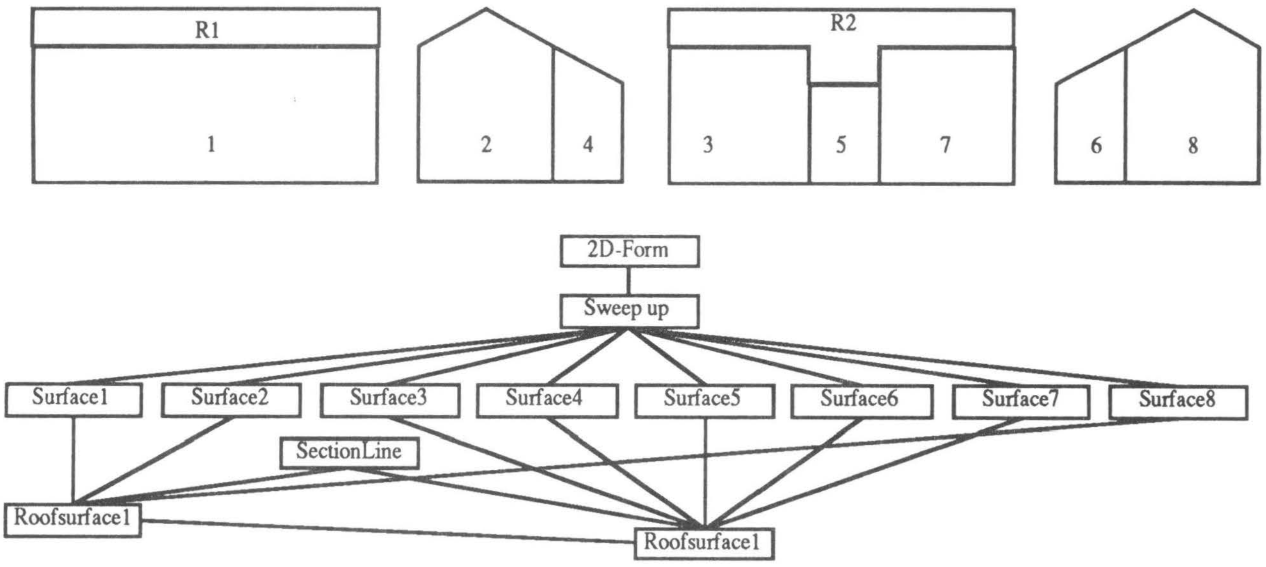


Figure 9: The facades of a small building and the corresponding design history.

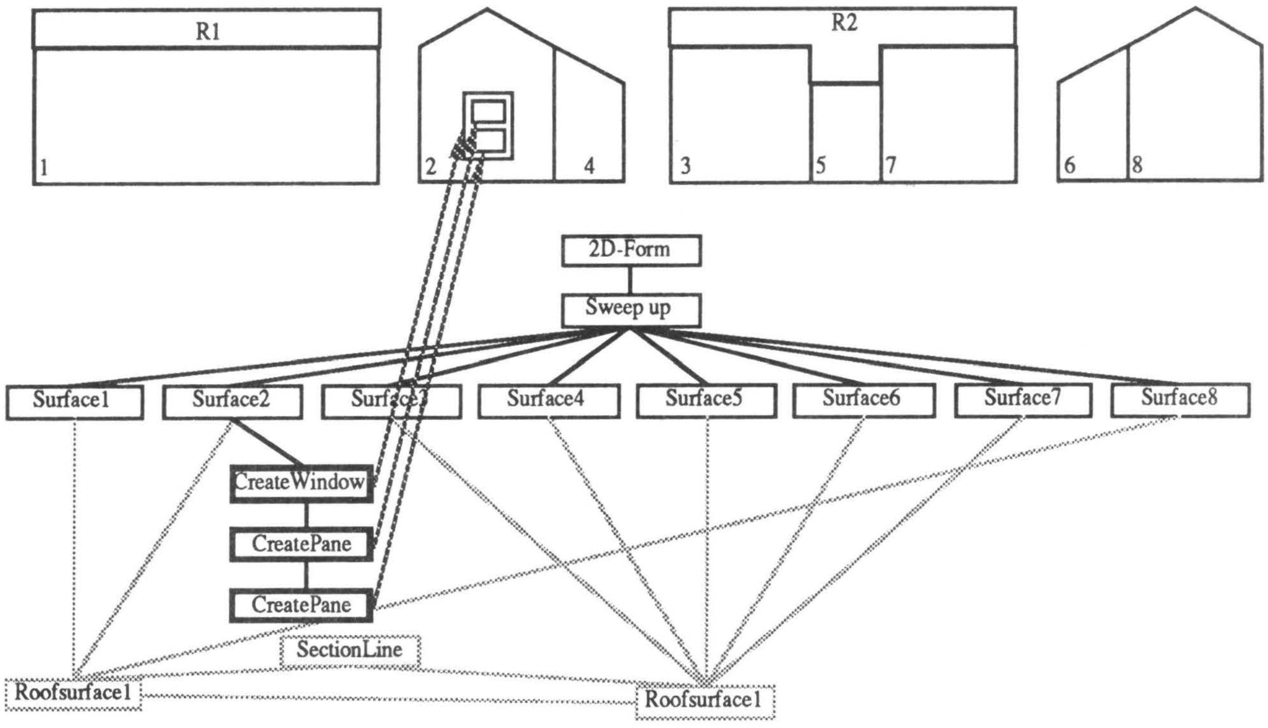


Figure 10: The object and the design history after creation of the first window. The irrelevant parts of the history are dimmed for the sake of clarity.

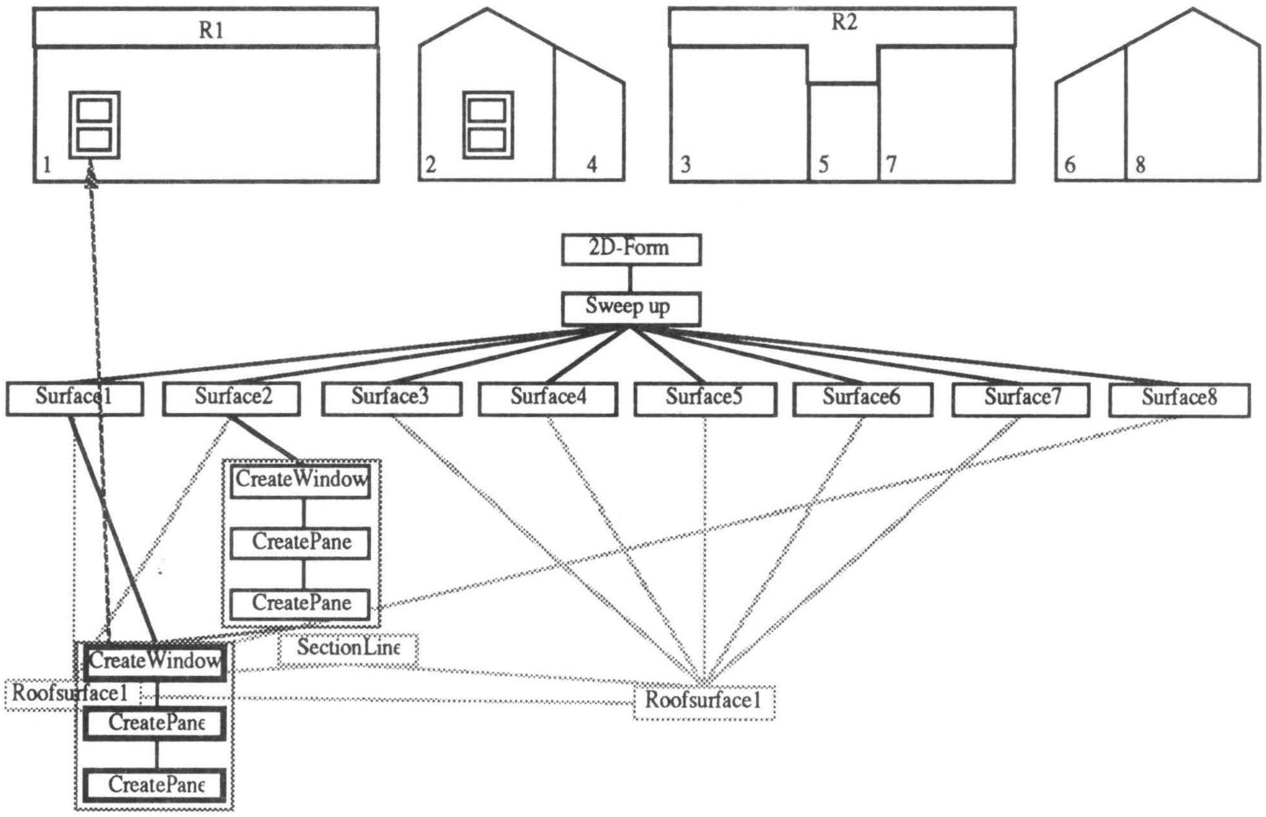


Figure 11: The resulting model and design history after applying the window creating commands at wall 1.

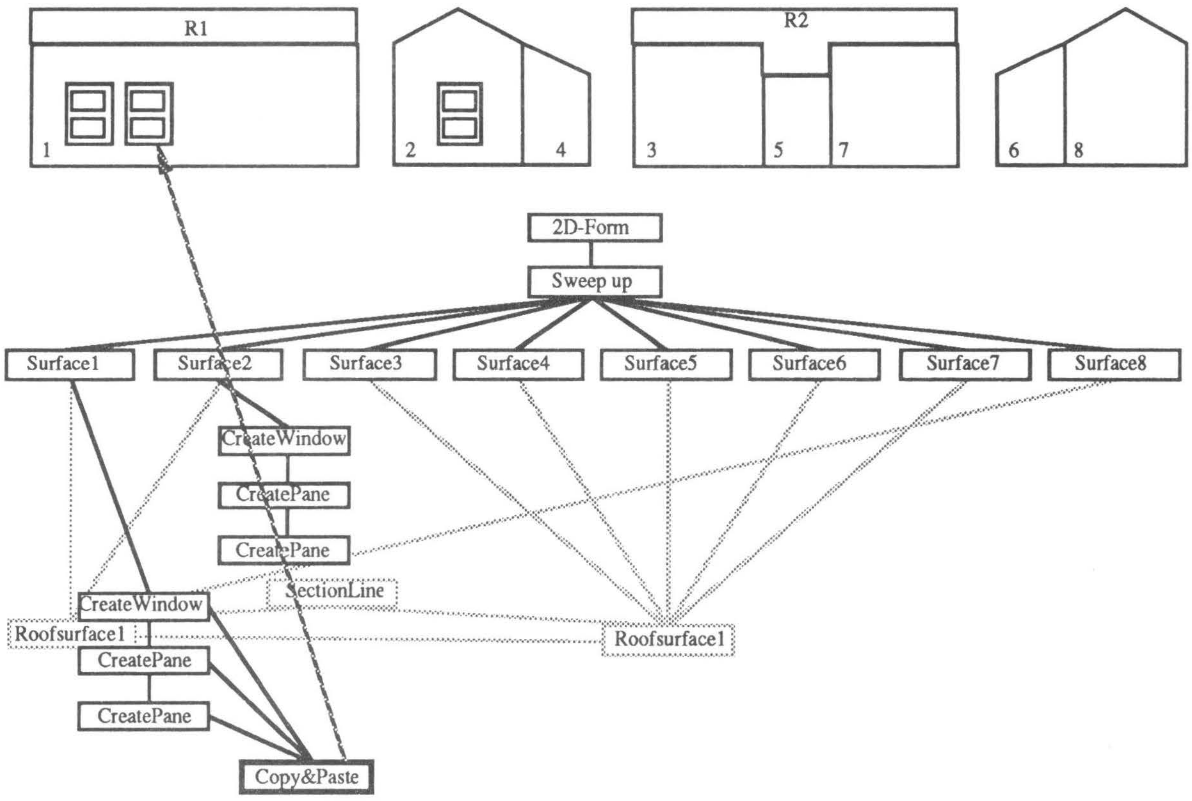


Figure 12: The Copy&Paste command applied to the window creating commands in the history network.

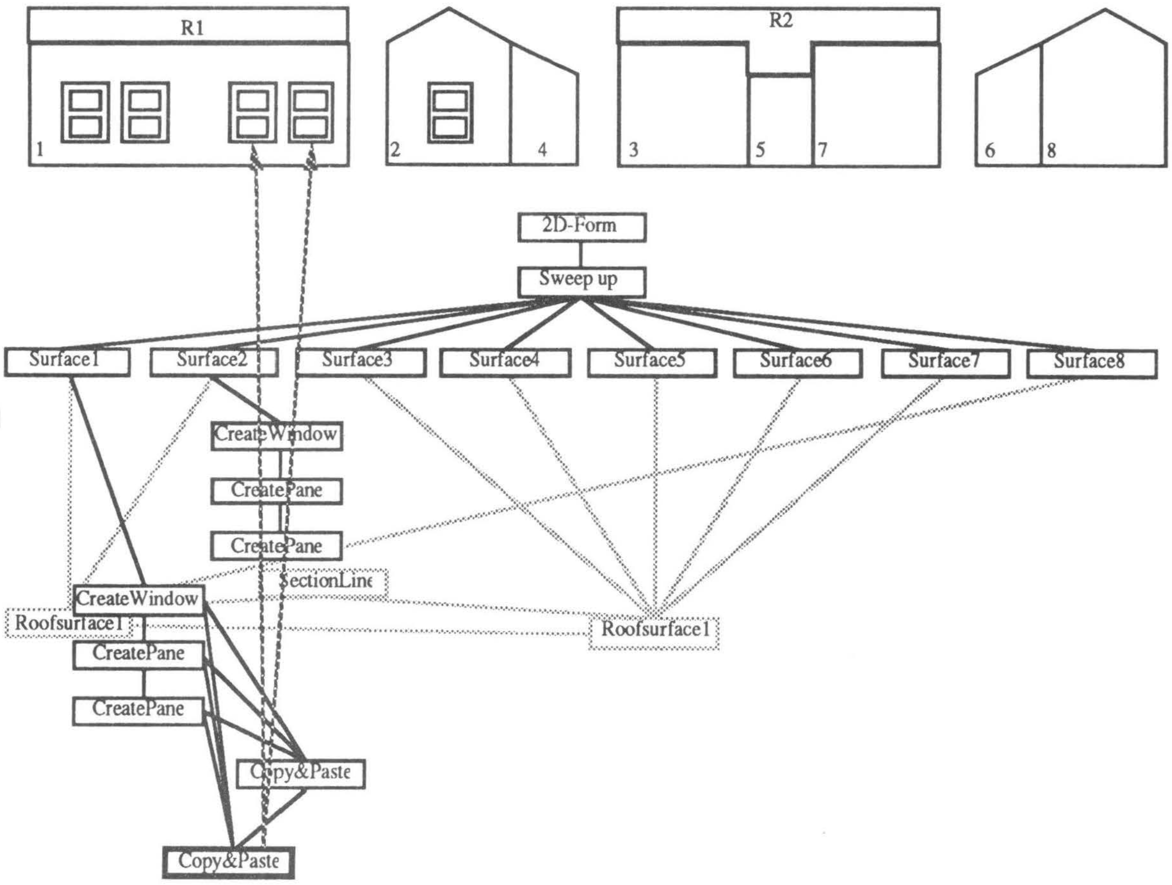


Figure 13: One method of creating another group of two identical windows on the same wall.

If the designer wants to create another pair of windows, but of a different size on another wall of the building, he could "open" the *CreateWindow* command and change e.g. the width parameter, as shown in Figure 14.

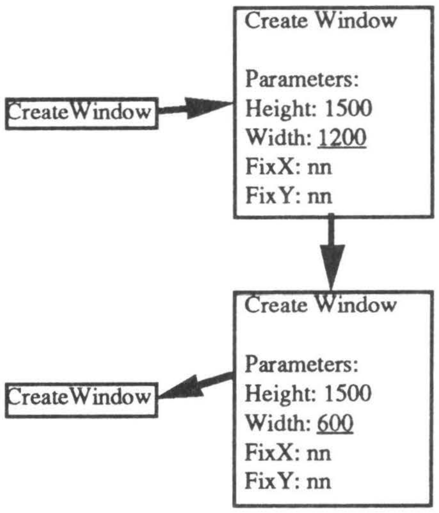


Figure 14 "Opening" of a node in the design history in order to change its parameters.

After this the designer could preferably apply the corrected sequence of commands on the wall where he wants the new pair of windows, as shown in Figure 15.

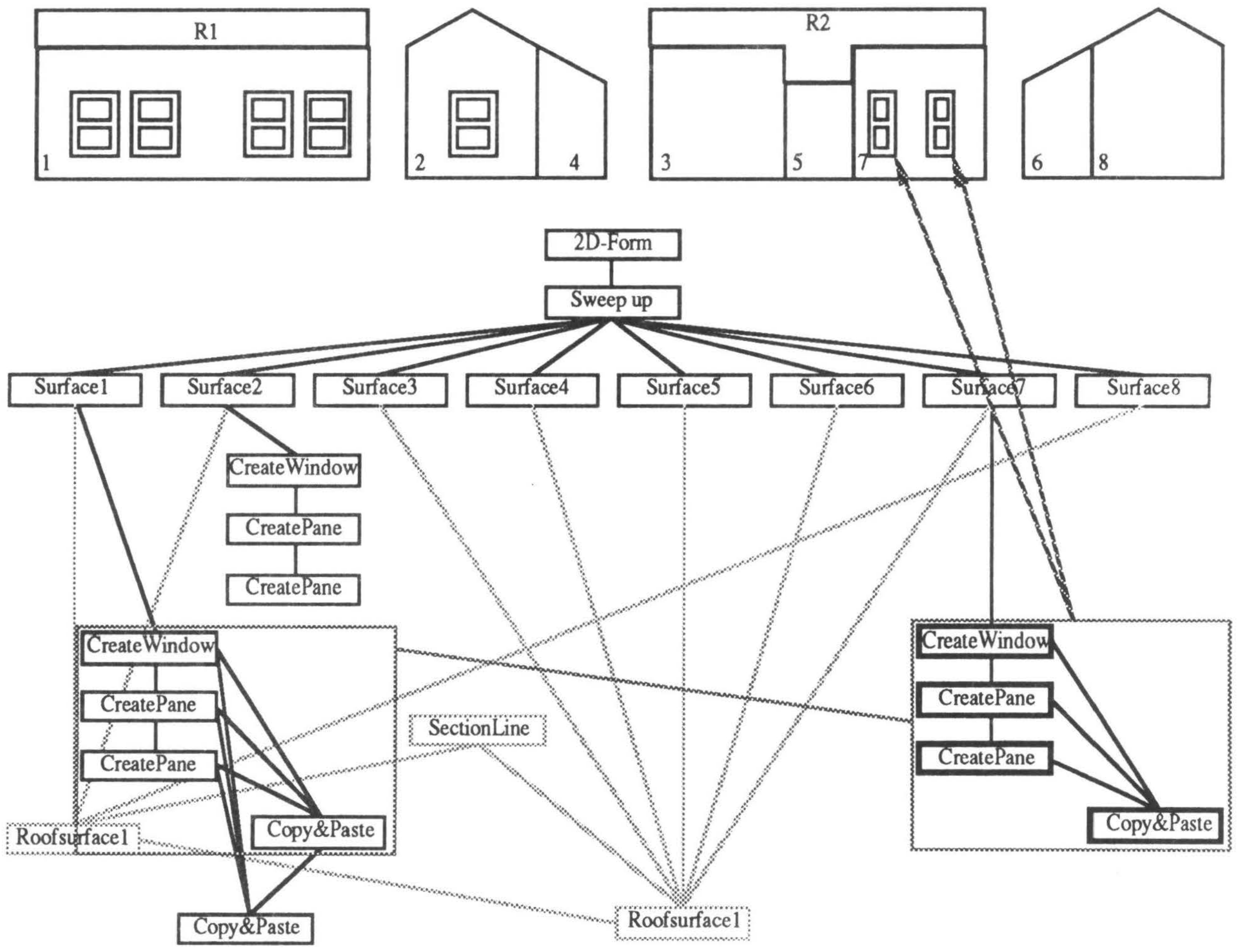


Figure 15: The creation of a pair of windows of a size different than the previously created windows by means of a modified copy of applicable nodes from the design history.

3.5. Results of the experiment

The work with the prototype so far has yielded some results, but a lot of questions are still lacking answers. A very natural obstacle, considering that the program in question is a prototype under continuing development, is the frequent interruptions caused by errors made by the user or sometimes by bugs in the code. These interruptions make it quite difficult to assess the efficiency of the program when compared to traditional modelling programs.

The most important result is in our opinion the convincing function of the history mechanism. It is quite fascinating to see how changes made to one part of the model can be pipelined to change all or just parts of the model. It could be compared to parametric design or powerful macros, with the exception that the user at no stage has to explicitely create the macros or parametric symbols.

When the user has developed different versions of his project, the strengths of the history approach become more obvious. The possibility of doing endless "what-if" studies is very powerful in the initial stages of housing design. In practice the way of doing this is the following:

> The user activates the stage of the model he wants to use as the starting point of a new version. By the command *ExtractDerivation* the program activates all the stages of design that have led to this result. Then the designer duplicates all the activated nodes in the history network window. At this stage he has two identical sets of designs related to each other by links. (See Figure 16). In the future the program will automatically create a consistent branch of such a copied history, at this stage the user must activate the nodes one by one and extract the arguments to be able to change them. After the formation of the new branch the user can change parameters, delete commands or insert commands or command sequences, which all affect the features of the model.

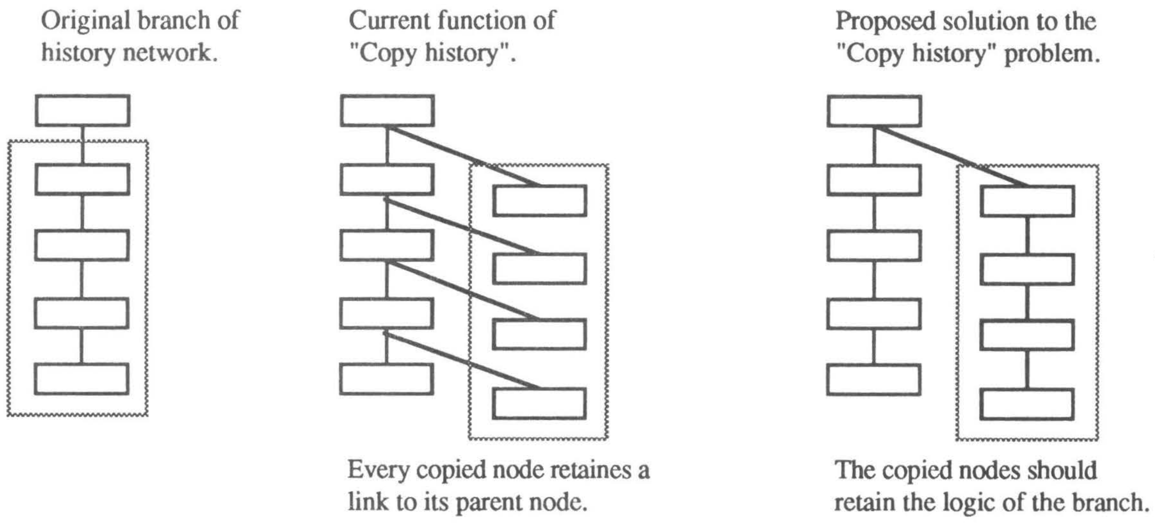

Original branch of history network.
Current function of "Copy history".
Proposed solution to the "Copy history" problem.
Every copied node retaines a link to its parent node.
The copied nodes should retain the logic of the branch.

Figure 16: The schemata representing current implementation of the CopyHistory command and a proposed solution that would keep the inherent logic of the copied sequence of nodes.

It is very difficult to decide what is more efficient, activating an object in the graphical view to perform actions on it or doing the same thing with the history mechanism. Perhaps the most important thing is, that the two ways to look at the model function together, so that when activating an object in the graphical view the corresponding node in the history is highlighted and vice versa. Sometimes it is more convenient to locate an object in the history network than among a lot of lines in the graphical view, sometimes the natural method in the graphic window is more appropriate.

Another important aspect of the history mechanism is the possibility of selectively showing or hiding any stage of the design process. It is even possible to manipulate in one view parts of the model that have been created in another view, without loosing the logical relations to the other parts of the model. Thus it is possible to develop the main body of a building in one view and the roof in another, and view the resulting combination of them in a third view, or perhaps in multiple views from different directions.

The program, as it is at the moment of writing this, has a set of two- and three-dimensional tools that are barely enough to do fundamental modelling of buildings. It is naturally necessive to develop many more tools before the program could replace conventional three-dimensional modellers. Some of the methods applied in the Schema program [Norman86] could be very good to implement also in MetaViews in order to enhance the user-friendliness of the program. It is however clear, that the history mechanism is a very powerful fundamental philosophy, that greatly widens the possibilities of viewing and manipulating models in architectural design.

4. Conclusions

The experiment conducted in this study has demonstrated the potential of the history mechanism of MetaViews in architectural design applications. Designers find it very convenient to build parametric models without explicitly programming them. Also the ability to handle different phases of design in simultaneous views gives better insight to the design parameters and their effects. Quick variational design experiments can easily be produced with these tools.

However, as the system is still an experimental prototype, a lot of questions considering its practical efficiency are still lacking answers. With our small scale example it is not possible to investigate, how one should navigate in a network database without getting "lost in hyperspace".

Some directions for further development can readily be pointed out. A more comprehensive set of geometric modelling tools is evidently needed. Geometric sketching also requires tools beyond traditional solid/surface modelling. For example, techniques for handling infinite or partially trimmed objects are essential. Also other relations than direct dependencies between entities, like constraints on design parameters and resulting objects, or associations to painted freehand sketches and verbal annotations would be very useful.

Acknowledgements

The original work leading to MetaViews was done at Helsinki University of Technology and has been supported by the Technology Development Centre. Practical application experiments and later development, continuing at Technical Research Centre of Finland with the "ESSU" (Aesthetically Qualified Environment and New Design Technology) project, are funded by the Emil Aaltonen Foundation.

References

[1] T.Takala: **Theoretical Framework for Innovative Computer Aided Design**, in H.Yoshikawa, E.Warman (eds.): *Design Theory for CAD*, Proc. of IFIP WG 5.2 Workimg Conference on Design Theory for CAD (Tokyo, Japan, November 1985), North-Holland 1986.

[2] T.Takala: **Design Transactions and Retrospective Planning - Tools for Conceptual Design**, Proc. of Second Eurographics Workshop on Intelligent CAD Systems (Veldhoven, The Netherlands, April 1988), to be published by Springer in 1989.

[3] T.Takala: **METAVIEWS - A Process-Oriented Approach to CAD and Graphics,** Proc. of Eurographics'88 (Nice, France, September 1988), North-Holland 1988, p.171-182.

[4] T.Takala: **User Interface Mangement System with Geometric Modelling Capability - A CAD System's Framework**, IEEE Computer Graphics and Applications, Vol.5, No.4, April 1985, p.42-50.

[5] Y.Yamaguchi, F.Kimura, P.J.W. ten Hagen: **Interaction Management in CAD Systems with History Mechanism**, Proc. Eurographics'87 (Amsterdam, The Netherlands, August 1987), North-Holland 1987. p.543-554.

[6] J.Rossignac, P.Borrel, L.Nackman: **Interactive Design with Sequences of Parametrized Transformations**, Proc. of Second Eurographics Workshop on Intelligent CAD Systems (Veldhoven, The Netherlands, April 1988), to be published by Springer in 1989.

[7] R.Makkuni: **A Gestural Representation of the Process of Composing Chinese Temples**, IEEE Computer Graphics and Applications, Vol.7, No.12, December 1987, p.45-61.

[8] J.Conklin: **Hypertext: An Introduction and Survey**, IEEE Computer, September 1987, p.17-41.

[9] D.Vanier: **Hypertext - a Computer Tool to Assist Building Design,** in *Conceptual Modelling of Buildings*, CIB W74 + W78 Seminar (Lund Institute of Technology, Lund, Sweden, October 1988).

[10] M. van Norman: **A digital Modelshop: The Role of Metaphor in a CAAD User Interface**, Design Computing, Vol.1, 1986, p.95-122.

Paper Session *Languages*

Paul Veerkamp, Ravic Pieters Kwiers,

and Paul ten Hagen

Design Process Representation in IDDL

Design Process Representation in IDDL

Paul Veerkamp, Ravic Pieters Kwiers, and Paul ten Hagen

Department of Interactive Systems
Centre for Mathematics and Computer Science (CWI)
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract: This paper presents the use of IDDL (Integrated Data Description Language) for design process representation in an Intelligent CAD system. IDDL is a special purpose programming language which originated from the Centre for Mathematics and Computer Science (CWI). It is now further being developed at both the University of Tokyo and CWI. IDDL is designed for implementing systems which assist a designer in performing a design task. It allows for a dynamic design object description and a flexible design process representation. In this paper we briefly introduce the design object representation in IDDL and elaborate on the design process representation. In particular, we pay attention to the multi-world mechanism and scenario interpreter. In the last section the actual implementation of IDDL on top of Smalltalk-80[†] is discussed.
Keywords: Knowledge Representation, Object-Oriented Programming, Logic Programming, Intelligent CAD, Multiple Worlds

1. Introduction

Conventional CAD systems, seen from a historical point of view, are centered around a graphical representation of the object to be designed. These are merely drafting systems used in combination with independently operating calculation applications. Designing however, is more than generating drawings of the artifact and testing these through calculation programs. Designing is a stepwise refinement process which takes a rough model of the design object and generates a more detailed model of the design object through a series of design actions [1, 13]. A drawing is more an interpretation of the design object model in a certain context than a description of the design object model. An intelligent CAD system has to provide a mechanism to generate several of such interpretations at the same time: a geometric representation, finite element method analyses as well as cost analyses etc..

An Intelligent CAD system which has the above mentioned properties contains a central design object description which we call the *meta-model*[8]. A meta-model is a context free description of the design object. During the course of the design process several models are derived from this meta-model. These models are evaluated in a certain

[†]Smalltalk-80 is a registered Trade Mark of Xerox Corp.

context (a *world)* and new information is obtained through the evaluation. The meta-model is extended with the newly derived information resulting in a more precise design object description. We call such a cycle of deriving a model from the meta-model, interpreting it and updating the meta-model, a design step. The whole design process consists of a number of design steps in a certain order.

Another important property of an Intelligent CAD system is the possibility to create multiple views on the design object in parallel. Such a system behaves like several expert systems working on the design object concurrently. We call these expert systems *application modules.* We make a distinction between application modules that work independently from each other and dependent application modules. The former may result in design object descriptions that differentiate. The latter will always amount in one unique design object description [10].

To implement an Intelligent CAD system which inhabits a meta-model and a design process model based on stepwise refinement, a special programming language is needed. This language allows for a flexible design object description and some means to describe the different design steps. It should have a multi-world mechanism to represent dependent and independent design object representations as well. A language fulfilling these requirements is IDDL (Integrated Data Description Language). IDDL, developed at CWI, is designed for the implementation of the IIICAD (Intelligent Integrated Interactive Computer Aided Design) system [9, 12]. The IIICAD system is a generic system, to be used in any domain where designing is involved. We focus our attention however, on the field of both architecture and mechanical engineering. In these areas we have the expertise to actually build a prototype system which possesses the functionality described above.

In the sequel we present IDDL, giving a short introduction to the constructs used for a design object description, and elaborating on the mechanisms for design process representation. In the last section we introduce the IDDL compiler and scenario interpreter. An example of the use of IDDL is given as well.

2. Design Object Representation in IDDL

Constructs for describing a design object in IDDL combine features of object-oriented and logic programming systems. In IDDL we are able to describe both declarative and procedural properties of an artifact. Functions denote procedural aspects and predicates and rules denote the more declarative aspects of a design object. The basis of the design object representation in IDDL is a deductive database which we call a *facts-base.* A facts-base involves objects and relationships between objects. In this chapter we shall show the essential elements of IDDL concerning the facts-base, but without becoming diverted by details and exceptions.

Before we are able to describe the facts-base we first have to introduce the notion of objects in IDDL. All data structures in IDDL, like in other object-oriented languages are objects. We however, make a distinction between *primitive* objects and *composite* objects. Primitive and composite objects are treated equally in IDDL, the only difference exists in their internal structure. Primitive objects are the basic data types of IDDL, they form the building blocks. They have a type and a value. Examples of primitive object types are: Integer, Real, Character, String, etc.. Objects which have more than a type and value own an internal structure; they are called composite objects. The internal structure of a composite object is made of *attributes, functions,* and *constraints.* Composite objects are referred to by their (unique) name. The set of all composite objects is found in the *objects-base.*

A facts-base is used to describe relationships between objects and consists of *facts.* A fact is a predicate symbol followed by a list of *constants,* separated by commas, e.g. supports(leg, table). A predicate symbol defines a relationship between its constants. A constant is either a symbol, starting with a lowercase character, in which case it is a composite object's name and it, hence, refers to a composite object or it is a primitive object denoted by its value, e.g. numberOfLegs(table, 4). In this example the symbol table is a composite object's name and 4 is a primitive object of type Integer and value 4.

In the current implementation of IDDL predicates are allowed only to have constants for their arguments. The next implementation, however, allows predicates to have functions for their arguments as well. These functions behave like messages, sent to an object. A function call is depicted as follows: an object, for which the function is called, a function name, i.e. a colon followed by a symbol starting with a lowercase character, and an (optional) list of constants separated by commas in square brackets, i.e. anObject :aFunction[aConstant]. An example of a predicate containing a function is: greater(table :numberOfLegs, 4). The object table, in this example, has an attribute named numberOfLegs which can be accessed through a function :numberOfLegs. The entire predicate expresses the fact that there exists an object, table, which has an attribute numberOfLegs, and the value of that attribute should be greater than four, i.e. it can be seen as a constraint. Functions do not only denote attribute values of objects. They may occur as a function declaration in the object's internal structure as well. A function returns a primitive object's value, either an attribute value or a computed value. Hence, there is no difference in behaviour between a function call which returns an attribute value and a call which returns a computed value, when you regard the object's internal structure as a black box.

In IDDL there exists a number of predicate symbols which have a special meaning. These built-in predicates describe the structure of the objects-base. First of all, a composite object is instantiated by asserting a one-placed predicate to a facts-base. The predicate symbol denotes the type of an object and its argument the name. An example of an instantiation is the assertion of table(kitchenTable), where table denotes an object's type and kitchenTable an object's name. In IDDL there exists a catalogue of *prototype* definitions. When a composite object is instantiated, the prototype definition which corresponds to the object's type is looked up. A copy of the prototype is made and is further used for the object's internal structure. The notion of prototypes in IDDL is similar to that of classes in Smalltalk-80[†] However, in IDDL the internal structure of an object may change during its life-time without having effect on the prototype definition, i.e.

[†]Smalltalk-80 is a registered Trade Mark of Xerox Corp.

attributes, functions, or constraints may be added or removed to/from an object. This is contrary to Smalltalk-80 where the methods and instance variables belonging to a certain instance cannot be extended without affecting all other instances of the same class.

So far, we have introduced built-in predicates for the instantiation of individual composite objects. An artifact representation consists of a set of composite objects which together form a detailed description of the design object. In IDDL, the decomposition of a design object into its parts is achieved through a built-in predicate hasPart. It is a two-placed predicate symbol denoting that the second argument is a part of the first, e.g. hasPart(table,leg). The part-of hierarchy of a design object may be several levels deep. If some information is requested from an object, (we shall describe how this is done in the next section), and the object is unable to provide it, then the request is delegated to its parts, as described by the part-of hierarchy, in order to obtain an answer.

In IDDL there exists an is-a hierarchy as well. It defines a prototype hierarchy similar to the class hierarchy in Smalltalk-80. It is realised by a two-placed built-in predicate symbol, isA. The isA predicate specifies that all objects which belong to the type of its first argument, are a specialisation of the type of the second argument. Objects of the first type inherit properties of the second type. An example is: isA(table, pieceOfFurniture); objects of type table inherit properties that are defined for objects of type pieceOfFurniture. A more detailed description of the delegation and inheritance mechanisms can be found in [11]. In the same article a more elaborated description of IDDL concerning the design object representation can be found.

Another built-in predicate is used to structure the facts-base. It behaves like a composite object definition predicate we have introduced above. It defines a special type of composite object, namely a *world*. A world is a facts-base in itself. It contains predicates and it has an objects-base associated with it. A world may contain other world definitions and so on. Worlds partition the facts-base in a straightforward manner. So if we assert world(table) to the facts-base (which itself is a world), we create a new facts-base with the name table to which we can assert new facts. The use of worlds will be discussed further in the next section.

3. Scenarios

In the previous section we have introduced the constructs in IDDL which concern the representation of a design object, the so called meta-model. In order to construct a meta-model IDDL provides mechanisms to reason about the current state of the design object and to generate new information on it. The construct to achieve this is called a *scenario*. A scenario contains both procedural and declarative knowledge in order to evaluate the design object model in a certain context and to derive new information. A scenario has a world associated with it, a set of functions (optional) and a number of IF-THEN rules [2]. A sequence of scenario calls corresponds to a series of design steps. A scenario is invoked with a world (facts-base) associated with it. Execution of a scenario involves that rules are being fired and that functions are called. A scenario remains active until a certain stop condition has been reached. Finally, control is given back to the caller of the scenario. The results of the call are registered in the called world and these are merged into the world of the caller, if and only if these two worlds are consistent with each other. Let us look now briefly at the syntax of scenarios and then concentrate on how a scenario interacts with its world when it is executed.

3.1. Syntax of scenarios

A scenario has a name, a rule selection mechanism in parentheses (optional), a set of function declarations (optional) starting with the keyword FUNCTIONS, and a number of IF-THEN rules starting with the keyword RULES. Thus, a scenario may look like:

```
scenarioName ( ruleSelectionMethod )
FUNCTIONS
    :f = { functionBody }
    :h[ argumentList ] = { functionBody }
RULES
    IF condition THEN action
    .
    .
```

A scenario is just another composite object and it, therefore has a unique name. A rule selection method determines in which order the rules are fired. We shall come to this later when we deal with the execution of scenarios. A function has a name, i.e. a colon followed by a symbol starting with a lowercase character, and a number (may be zero) of arguments. An argument list is a sequence of variables, symbols starting with a capital, separated by commas. A function body consists of a number (may be zero) of temporary assignments separated by semicolons, followed by a return expression. All variables that occur in a function body are automatically defined as local variables within that function. An example of a function declaration is:

```
:area[ R ] = { RR := R * R ; 3.14 * RR }
```

Function calls may occur within a function body as well, however, we do not allow recursion. Note that such a function call already appeared in the previous example. The function "*" is called upon R with R as its argument. Strictly speaking the expression should have read something like: R :multiplyWith[R], but for convenience we adopted a more readable notation.[1]

An IF-THEN rule describes a design action. The condition part is an evaluation of a certain expression in which some variables may be bound against for instance a facts-base (an alternative way to bound a variable is via a user-interface call). A condition, however, does never change the state of a facts-base, it has a so called read-only permission. It is justified to compare a condition with a query to a database, it examines the current state of the design object description. The action part adds new information to a design object description either by asserting a new fact or by assigning a value to an attribute.

[1] A similar strategy is used in Smalltalk-80 for the arithmetic and boolean operators.

Both a condition and an action are called *formulae.* In order to define what a formula is we have to define *terms* and *predicates* first. A term is defined as follows:

1. A variable is a term, e.g. X, Y, ATable.
2. A constant is a term, e.g. a, b, aTable, 4, 3.14, 'hello world', etc..
3. A function call is a term, e.g. aTable :area[X], X :f.[2]
4. If t is a term, then (t) is a term.

A *ground term* is a term not containing variables. If p is a n-placed predicate symbol, a symbol starting with a lowercase character, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a predicate. A *predicate constant* is a zero-placed predicate. In IDDL only the predicate constants, T (true), F (false), succeed, and fail are allowed. Now we have defined terms and predicates, a formula is defined as follows:

1. A predicate is a formula.
2. If p is a predicate, then %p is a formula.
3. If p and q are formulae and $\otimes$ is a binary connective, then p $\otimes$ q is a formula.
4. If p is a formula, then ~p is a formula.
5. If p is a formula, then (p) is a formula.

So far, we defined the unary operators % (the *unknown* operator) and the unary connective ~ (negation). The binary connectives are & (logical and) and | (logical or). Hence, a valid formula is for instance: p(a) & ~ (%q(a) | r(b)).

Basically, IDDL is equipped with two-valued logic. Therefore, if a fact is queried to a facts-base, the result is either 'true' or 'false'. A fact may appear either positively or negatively, e.g. p or ~p. Similar, we can pose the query p or ~p which evaluates to 'true' if respectively p or ~p occurs in the facts-base. Furthermore, a facts-base in IDDL behaves according to the "closed world assumption" [7]. Thus, the query of a fact fails either since the negation of that fact appears in a facts-base, or it was not present at all (unknown). In some situations, however, we want to know explicitly whether a fact is unknown or not. For that case IDDL provides the unknown operator, %. Its truth table is built up as follows; let p be a predicate, then evaluate

$$\%p \text{ to } \begin{cases} \text{true} & \text{if neither p nor} \sim\text{p can be found in the facts-base.} \\ \text{false} & \text{if either p or} \sim\text{p appears in the facts-base.} \end{cases}$$

With the use of % we have implicitly introduced a third truth value, the unknown. The % operator can only be used in condition formulae. It may be used with both constants and variables. The former checks whether a certain predicate is absent in a facts-base. The latter checks whether a whole set of predicates is absent. For instance, %p(a) succeeds if neither p(a) nor ~p(a) appears in the facts-base. The query %p(X) checks whether there exists at all a one-placed predicate p or its negation. Note that a binding of X cannot

[2] We do not allow a nesting of function calls, thus only constants and variables may appear in a function call.

possibly occur, since X will only be bound when there exists a unary predicate p. However, in that case the query fails and it, therefore, produces no effect.

3.2. Evaluation of a condition formula

As said before, the predicates occurring in the condition formulae evaluate the current state of a design object description. For condition formulae we distinguish the predicate constants, T and F, and three kinds of predicates: (1) queries to a facts-base, (2) built-in predicates, and (3) user interface queries. The predicate constants obviously evaluate to the value they represent. When a predicate from the first category occurs in a condition formula, the facts-base is searched for a predicate with the same predicate symbol and the same arity. If such a predicate is found, its arguments are matched against those of the predicate appearing in the condition predicate (the actual unification mechanism is explained below). Such a query evaluates to true if the match succeeds. Predicates that do not belong to the second and third category automatically belong to the first.

Built-in predicates perform a test on their arguments. Examples of these are: equal, greater, smaller, known etc.. The terms of such a predicate are evaluated and compared with each other according to the type of predicate symbol. For instance, the built-in predicate equals evaluates to true in case of equals(a, a), and it evaluates to false in case of equals(a, b). These are of course trivial examples. The evaluation is less straightforward, if the terms are variables or function calls, e.g. equals(table :numberOfLegs, 4). The predicate is evaluated with the return value of the invoked function. However, when a function returns 'nil', for instance when an attribute value is not (yet) defined, the entire predicate evaluates to 'unknown', and hence the predicate fails. Furthermore, if we pose the query %equals(table :numberOfLegs, 4), the predicate will return true if the function returns nil, otherwise it will fail. Moreover, the query known(table :numberOfLegs) succeeds if the attribute numberOfLegs is already determined.

Predicates from the third category behave similar to those of the first category, but with the difference that the user is queried instead of a facts-base.

A formula is evaluated from left to right. The operator % has the highest precedence, followed by ~, while & and | have the lowest precedence. Variables that occur in a rule are local to that rule. They are bound during the evaluation of the condition. If there are still unbound variables after an evaluation, the rule fails. Presently, IDDL uses a very simple unification algorithm [6] to bind variables. The predicates occurring in a formula are processed one by one, gradually reducing the number of possible bindings. In short the algorithm behaves as follows. The first predicate is unified with a facts-base resulting in an *instantiation pair list* containing all possible bindings. The same happens with the second predicate resulting in a other instantiation pair list. The two lists are compared, conflicting bindings are removed and the two lists are merged into one. Then the following predicate is evaluated, its instantiation pair list is merged with the result list, and so on. The algorithm is best described with an example. Suppose a facts-base contains the following facts:

```
p(a). p(b). p(c).
q(a, b). q(a, c). q(b, c).
r(b, c, b).
```

Now, the formula, p(X) & q(X, Y) & r(X, Y, Z) is posed to this facts-base. For the first predicate an instantiation pair list, {X → a;b;c} is found. Unification of the second predicate generates {X,Y → a,b;a,c;b,c}. A merge of the two lists results in: {X → a;b. X,Y → a,b;a,c;b,c}. Note that c is removed from the first list. The third predicate produces {X,Y,Z → b,c,b}. The resulting instantiation pair list is then {X → b. X,Y → b,c. X,Y,Z → b,c,b}. The final result of the unification is that X is bound to b, Y to c, and Z to b. Furthermore, the condition succeeds.

3.3. Evaluation of an action formula

An action formula adds new information to a design object description. All variables that occur in an action formula must be bound at the time the action is evaluated. If there still exists an unbound variable, the rule will fail. Similar to condition formulae, there are several categories of predicates available for an action formula. We distinguish four kinds of predicates: (1) assertions to a facts-base, (2) assignments to attributes, (3) scenario calls, and (4) control predicates.

A predicate symbol which is not an assignment, a scenario call, or a control predicate is automatically regarded as an assertion to the scenario's world. The terms of a predicate are evaluated before assertion, i.e. the functions are evaluated prior to the assertion. So, the assertion of greaterThan(table :numberOfLegs, 4) will result in a predicate with two constant terms.

Assignments in IDDL are basically allocations of attribute values. An assignment is realised with a built-in predicate gets. It is a two-placed predicate, whose first term is a function call to an object's attribute. The second term produces the value to be assigned, e.g. gets(table :numberOfLegs, 4). Only uninstantiated attributes may be assigned. Therefore, an assignment fails if an attribute is already determined. Besides, an assignment predicate fails when its first terms does not contain a function call, or when a function call is not a reference to an attribute.

IDDL provides a mechanism which allows the user to invoke a scenario from a rule, and hence, to create several levels of scenarios. We call a scenario from which a scenario is invoked, the *parent* scenario. A built-in predicate use is employed in IDDL to call a scenario. Its first argument is a scenario name. Further arguments of the predicate are references to worlds. These worlds are associated with the scenario during its activity. A scenario remains active until a certain stop condition is encountered (represented by control predicates, which we will introduce below). An example of a scenario call is: use(designTable, tableWorld). When there are no world references in a scenario call, the invoked scenario will use the worlds of the parent scenario. The results of a scenario are checked for consistency with the worlds of the parent scenario after successful termination. These results, i.e. assignments and assertions, are consistent if the assertions do not violate constraints in the parent scenario's worlds and if the negations of the assertions are absent in the parent scenario's worlds.

The following control predicates can be found in action formulae: succeed, fail, and directive. The first is a zero-placed predicate which causes a scenario to succeed immediately, and returns control to the parent scenario. The second, also zero-placed, behaves the opposite way. It causes a scenario to fail with loss of all results. The third one-placed built-in predicate is used to change the current rule selection mechanism dynamically. For instance, when directive(recency) is encountered, from now on the rule selection method recency will be applied by the scenario interpreter until an other directive is encountered.

3.4. Multiple worlds.

The multi-world mechanism in IDDL allows for invoking multiple scenarios concurrently. This implies that there are more than one scenarios active at the same time. Moreover, it means that multiple worlds are accessed concurrently. Two additional binary connectives are employed to invoke the multi-world mechanism. These are: && (parallel and) and | | (parallel or). The former is used to create dependent worlds. This dependency implies that the scenarios' worlds have to be consistent with each other, at the time all dependent scenarios have terminated. The following action formula is an example of the invocation of two dependent scenarios, use(scenario1, world1) && use(scenario2, world2). Dependent worlds in IDDL allow the user to generate different interpretations of the meta-model concurrently.

The | | connective is used to create independent worlds. Independent worlds, in contrast with dependent worlds, do not necessarily have to be consistent with each other. Inconsistent independent worlds result in distinct design object descriptions. Independent worlds provide the user with a mechanism to model distinct design solutions. An example of the invocation of independent worlds is: use(scenario1, world1) || use(scenario2, world1).

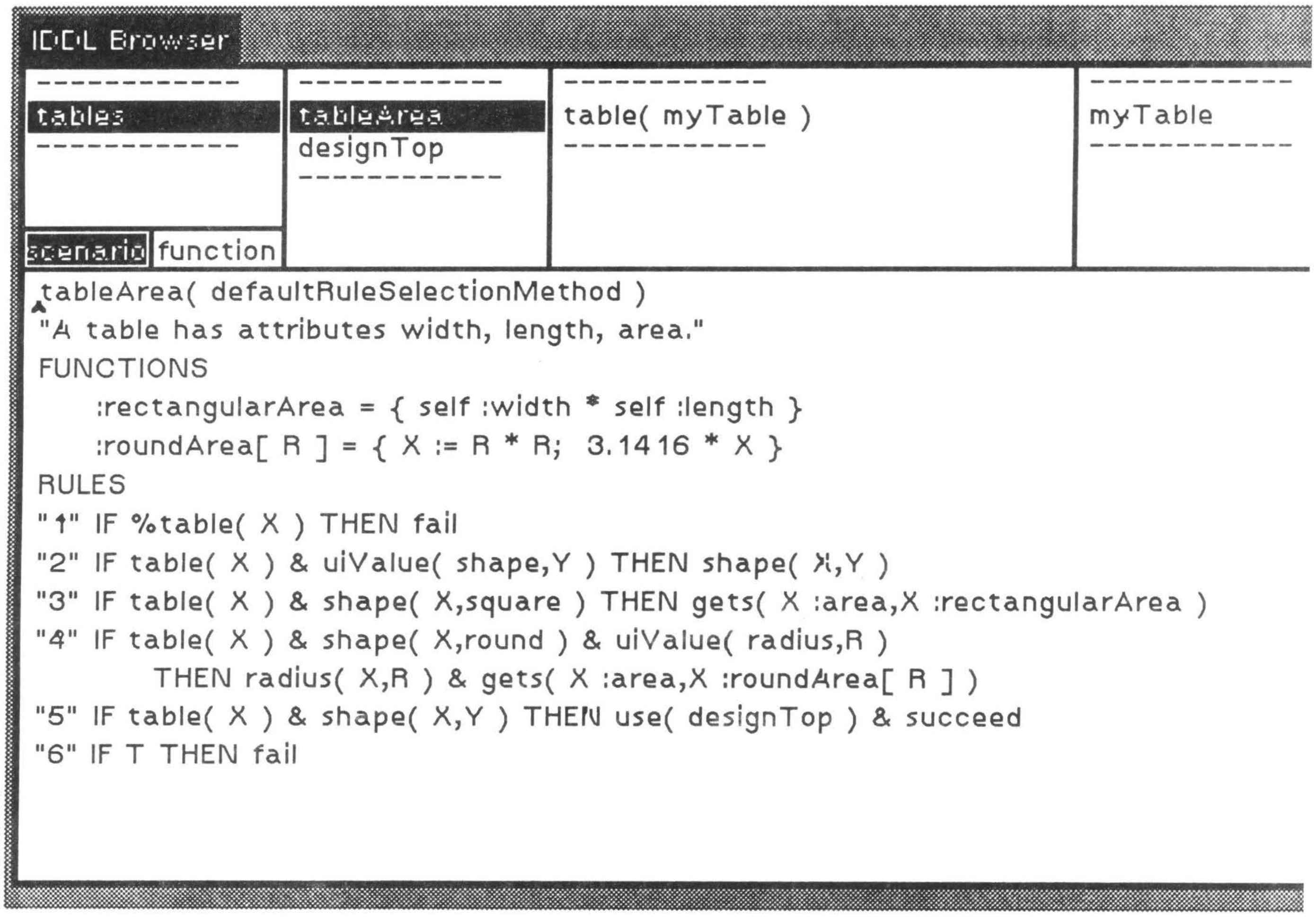
3.5. The modal operator

Designers are quite often used to reason about uncertain facts. IDDL is therefore equipped with a modal operator P. The operator P is a unary operator. Syntactically it is used similar to the operator %. However, P only appears in action formulae. The assertion of a predicate which is preceded by P results in an *assumed* fact [3]. Normal assertions to a facts-base are permanent. It may be convenient, however, to assert a fact which is not absolutely certain, as an assumed fact. Such a fact may in a later stage of the design process be withdrawn, or become certain. Such an assertion is, for instance, P shape(table, round).

A fact which is derived from a assumed fact is an assumed fact as well. A fact is derived from all facts that appeared in the condition of the IF-THEN which asserted it. A derived fact has a list of dependencies associated with it. This list contains the assumed facts from which it is derived (only one level deep). At the time an assumed fact becomes certain, it is removed form the dependency lists in which it occurred. Facts, that depend only on this fact, become certain as well. An assumed fact cannot cause an inconsistency. If it does, it is removed from the fact-base, and all facts depending on it as well.

4. The scenario interpreter

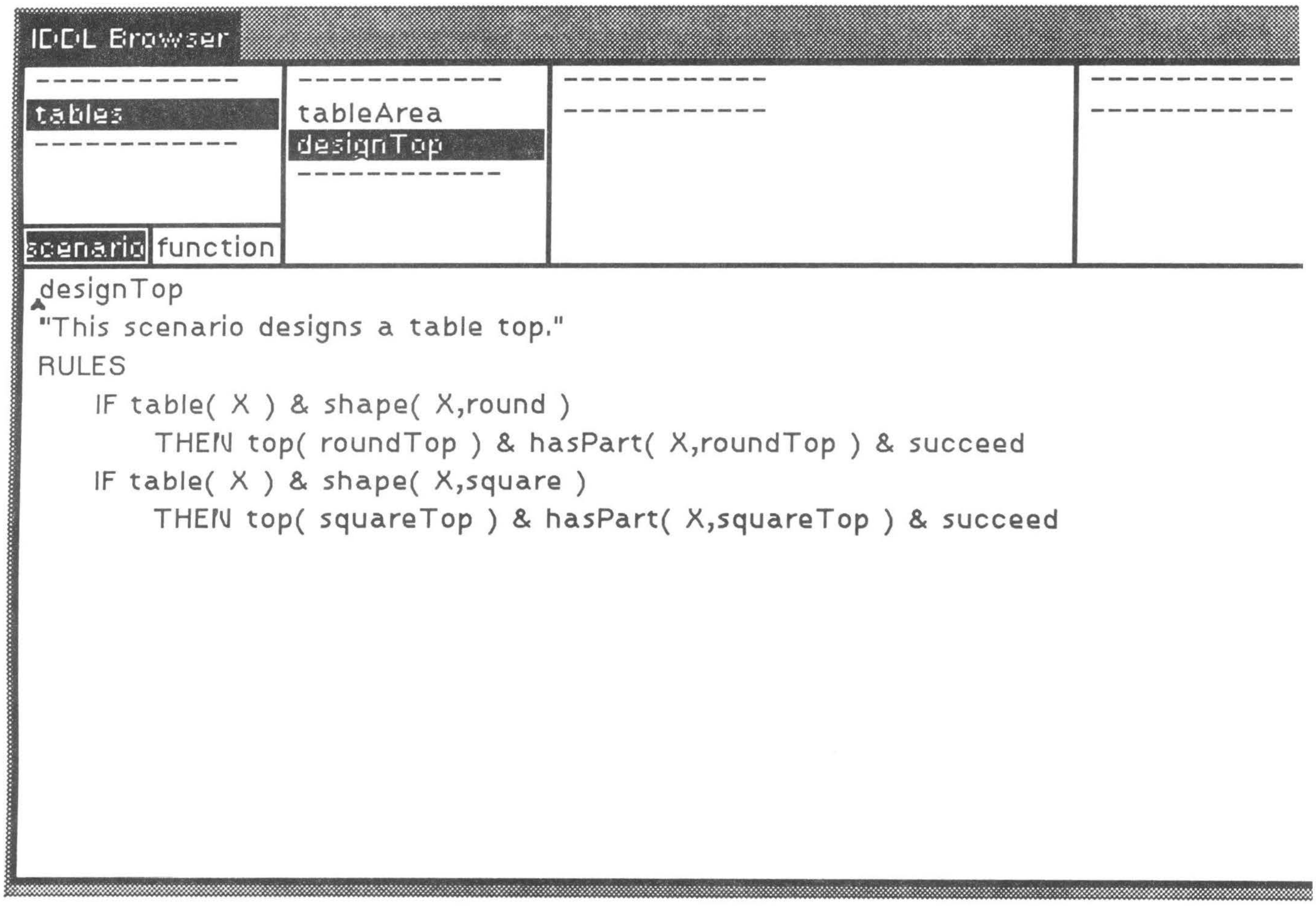
In this section we illustrate the execution of IDDL scenarios with an example of the design of a part of a table. The encoded scenarios are shown in Fig. 1. to Fig. 3..



```
tableArea( defaultRuleSelectionMethod )
"A table has attributes width, length, area."
FUNCTIONS
    :rectangularArea = { self :width * self :length }
    :roundArea[ R ] = { X := R * R;  3.1416 * X }
RULES
"1" IF %table( X ) THEN fail
"2" IF table( X ) & uiValue( shape,Y ) THEN shape( X,Y )
"3" IF table( X ) & shape( X,square ) THEN gets( X :area,X :rectangularArea )
"4" IF table( X ) & shape( X,round ) & uiValue( radius,R )
        THEN radius( X,R ) & gets( X :area,X :roundArea[ R ] )
"5" IF table( X ) & shape( X,Y ) THEN use( designTop ) & succeed
"6" IF T THEN fail
```

Fig. 1. Scenario tableArea before execution

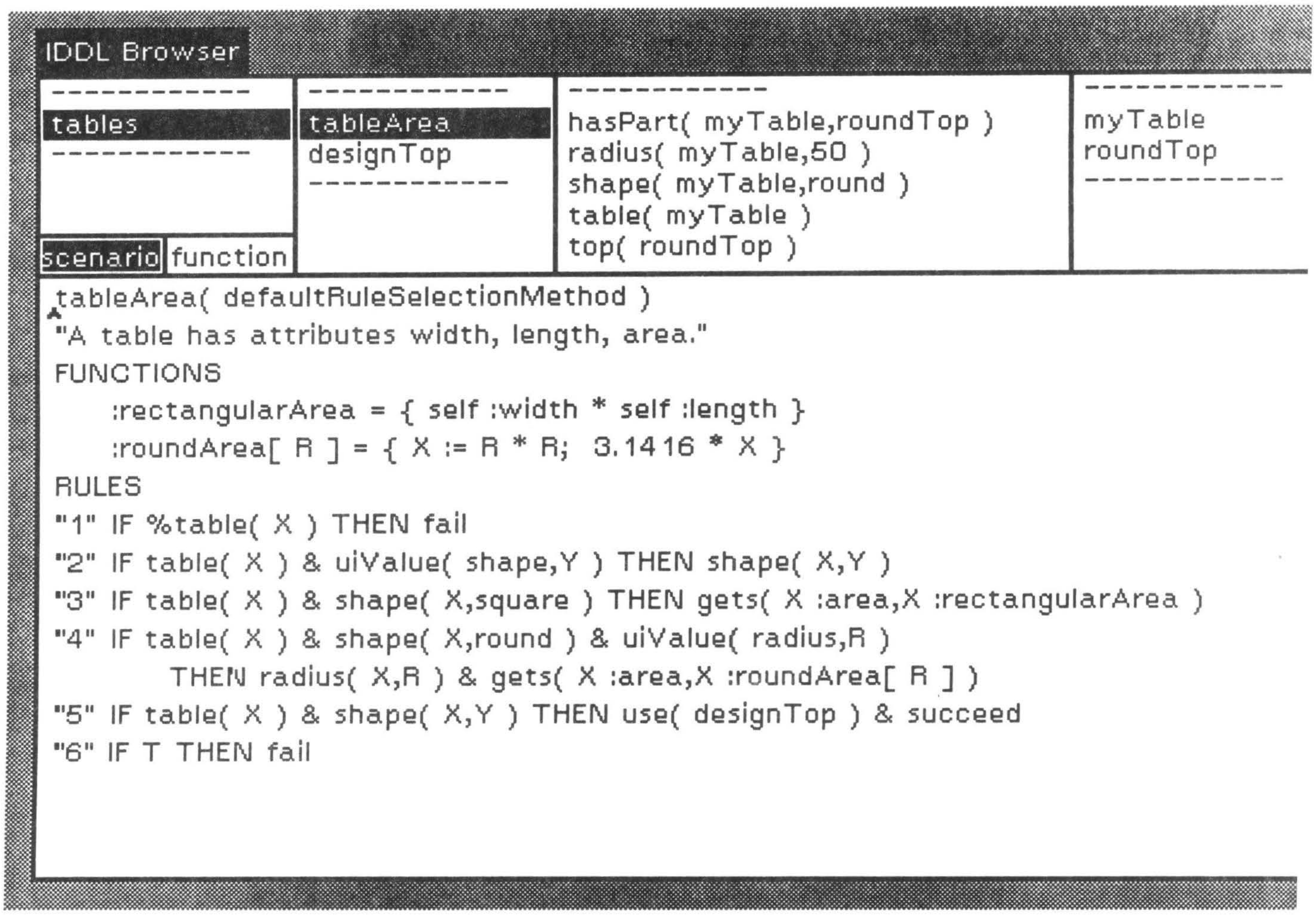
If the scenario from Fig. 1. is executed, the first rule is selected according to the specified rule selection method. In this case it is the default rule selection method, i.e. straightforward top-down selection, and rule #1 will be selected. If there does not exist a predicate table in the world associated with tableArea, the query %table(X) will evaluate to true and the predicate fail will cause the execution of the scenario to terminate unsuccessfully. The existence of any one-placed predicate table, for instance table(myTable), will cause rule #1 to fail and the next rule to be selected. The condition of rule #2 consists of a conjunction formed by the conditional and. The first part of the condition will evaluate to true with X bounded to myTable. The second part is a user-interface request, i.e. the user is asked to enter the shape of the table. If the user has entered a non-empty string, say round, this query will succeed with Y bounded to round. As a result the predicate shape(myTable, round) will be asserted to a temporal facts-base. Rule #3 and #4 compute the table area, depending on the table shape. In the case of a round table, rule #3 will not match but rule #4 will. Being a round table the user is asked for the radius, say fifty, the predicate radius(myTable, 50) is asserted and the result of the function call myTable :roundArea[50] is assigned to the attribute area of myTable. Rule #5 will match, resulting in a call to the scenario designTop. If designTop terminates successfully, its results will be added to the facts-base of tableArea and the predicate succeed will cause the execution of tableArea to terminate successfully. Likewise, the results of tableArea will be added to the facts-base of its parent scenario. Rule #6 will only be fired if rule #5 did not succeed, i.e. it did not match because one of the queries failed or the scenario call failed. Rule #6 will succeed always because T evaluates to true and fail will cause the execution of tableArea to terminate unsuccessfully.



```
designTop
"This scenario designs a table top."
RULES
    IF table( X ) & shape( X,round )
        THEN top( roundTop ) & hasPart( X,roundTop ) & succeed
    IF table( X ) & shape( X,square )
        THEN top( squareTop ) & hasPart( X,squareTop ) & succeed
```

Fig. 2. Scenario designTop

The scenario designTop (Fig. 2.) has access to the facts-base of its parent scenario. Therefore, its first rule will match, the predicates top(roundTop) and hasPart(myTable,roundTop) will be asserted and control will be returned to tableArea. As a result of the assertion of top(roundTop), the object roundTop will be added to the objects-base because top is a prototype. The results of designTop will be added to the facts-base of tableArea.



```
hasPart( myTable,roundTop )
radius( myTable,50 )
shape( myTable,round )
table( myTable )
top( roundTop )
```

```
myTable
roundTop
```

```
tableArea( defaultRuleSelectionMethod )
"A table has attributes width, length, area."
FUNCTIONS
    :rectangularArea = { self :width * self :length }
    :roundArea[ R ] = { X := R * R;  3.1416 * X }
RULES
"1" IF %table( X ) THEN fail
"2" IF table( X ) & uiValue( shape,Y ) THEN shape( X,Y )
"3" IF table( X ) & shape( X,square ) THEN gets( X :area,X :rectangularArea )
"4" IF table( X ) & shape( X,round ) & uiValue( radius,R )
        THEN radius( X,R ) & gets( X :area,X :roundArea[ R ] )
"5" IF table( X ) & shape( X,Y ) THEN use( designTop ) & succeed
"6" IF T THEN fail
```

Fig. 3. Scenario tableArea after execution

In Fig. 3. the results of the execution of tableArea are visible. The facts-base contains five facts and the objects-base contains two objects.

5. Conclusions

In this paper, we have showed the current state of the implementation of IDDL. At the moment we have at CWI a prototype version of IDDL operational. This version provides the functionality, as described by this paper, except for the multi-world mechanism and the modal operator. The implementation of these concepts is still in an experimental phase. Other members of the project team are working on application modules which can easily be integrated in the system. The interface between these modules and the IDDL kernel is encoded in scenarios. Examples of such application modules are an intelligent user-interface and a geometric modeller. Also, we are working on an underlying database model to represent the objects-base efficiently.

The next phase of the implementation of IDDL will pertain to incorporate the multi-world mechanism and the modal operators into the prototype system. Furthermore, an extensive example in the domain of architectural design (floor planning of a house) will be encoded in IDDL.

Acknowledgements

The authors wish to thank Tetsuo Tomiyama, Varol Akman, Farhad Arbab, Jan Rogier and Sabine Eisses for the many fruitful discussions, which have contributed a lot to this paper. Furthermore, we would like to thank Monique Megens whose pioneering work on the implementation of IDDL is invaluable.

References

1. V. Akman, P.J.W. ten Hagen, J. Rogier, and P.J. Veerkamp, "Knowledge engineering in design," *Knowledge-Based Systems*, vol. 1, no. 2, pp. 67-77, 1988.
2. R. Davis and J. King, "An Overview of Production Systems," in *Machine Intelligence*, ed. E.W. Elcock and Donald Michie, pp. 300-332, Ellis Horwood Ltd., Chichester, 1977.
3. J. De Kleer, "An Assumption Based TMS," *Artificial Intelligence*, vol. 28, pp. 127-162, 1986.
4. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its implementation,* Addison-Wesley, Reading, MA, 1983.
5. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley, Reading, Mass., 1984.
6. J.W. Lloyd, *Foundations of Logic Programming,* Second, Extended Edition. Springer-Verlag, Berlin, 1987.
7. R. Reiter, "Towards a Logical Reconstruction of Relational Database Theory," in *On Conceptual Modelling*, ed. M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, pp. 191-233, Springer-Verlag, New York, 1984.
8. T. Tomiyama and P.J.W. ten Hagen, "Representing Knowledge in Two Distinct Descriptions: Extensional vs. Intensional," Report CS-R8728, Centre for Mathematics and Computer Science, Amsterdam, 1987.
9. T. Tomiyama and P.J.W. ten Hagen, "Organization of Design Knowledge in an Intelligent CAD Environment," in *Expert Systems in Computer Aided Design*, ed. J. Gero, Amsterdam, 1987.
10. P.J. Veerkamp, "Multiple Worlds in an Intelligent CAD system," in *Intelligent CAD: Record of IFIP 5.2 Workshop on Intelligent CAD*, ed. H. Yoshikawa and D. Gossard, North Holland, Amsterdam, 1988.
11. P.J. Veerkamp, V. Akman, P. Bernus, and P.J.W. ten Hagen, "IDDL: A Language for Intelligent Interactive Integrated CAD Systems," in *Intelligent CAD Systems 2: Implementational Issues*, ed. Varol Akman, P.J.W ten Hagen and P.J. Veerkamp, Springer-Verlag, Berlin, 1989 (to appear).

12. Bart Veth, "An Integrated Data Description Language for Coding Design Knowledge," in *Intelligent CAD Systems 1 — Theoretical and Methodological Aspects*, ed. P.J.W ten Hagen and T. Tomiyama, pp. 295-313, Springer-Verlag, Berlin, 1987.

13. P.J.W. ten Hagen, J. Rogier, and P.J. Veerkamp, "An Environment for Knowledge Represenation for Design," in *Proceedings of Civil Engineering Expert Systems*, Madrid, 1989.

Luis A. Pineda, Ewan Klein

On the Integration of Graphical and Linguistic Knowledge for CAD Systems

On the Integration of Graphical and Linguistic Knowledge for CAD Systems.

Luis A. Pineda and Ewan Klein
Centre for Cognitive Science & EdCAAD
University of Edinburgh

1. Introduction.

In this paper a logical and graphical language for representing CAD knowledge is presented. The language of first order logic (FOL) is augmented with a set of geometrical and topological functions. Logical constant and predicate symbols are used for representing linguistic knowledge. This language is a theoretical presentation of the representational system used in a program called GRAFLOG [9, 10], in which the relationships between natural language expressions and graphical representations in computer graphic interaction are explored. In Section 2, an interaction with the system is illustrated. In Section 3 the syntax and semantics of the representational system are presented. The representational language determines an structure in which graphical symbols and relations are systematically related. In Section 4, a notion of graphical concept is introduced and the way graphical concepts emerge from the graphical structure is shown. In Section 5, a procedure by which the semantic representation of drawings is produced through the interactive session is presented. For this purpose, a set of identification rules acting upon the graphical and linguistic input is defined. In architectural and other kinds of drawings there are basic symbols that are explicitly drawn, as well as other context dependent space partitions that receive an interpretation in terms of the graphical context in which they emerge. For instance, *walls* of an architectural drawing are explicitly drawn, but the *rooms* that those walls determine emerge from the graphical context. Nevertheless, both kind of symbols recieve an interpretation and are named by a similar linguistic device, namely, common nouns. The way basic and context dependent graphical symbols are represented and referred to by the language is illustrated in Section 6.

1.1. Some related Work.

The application of logical representations to CAD is a current research field. Though the features of the so-called Intelligent CAD systems are subject of considerable debate, some requirements of these systems have been vaguely agreed, as for instance, a practical and theoretically sound knowledge representation framework, a flexible acquisition and explanation mechanism, an intelligent interface, a powerful graphics package and engine, and all of them supporting CAD applications themselves [14]. Within these topics, the need for intelligent interfaces is highly emphasised [7, 12, 13], and the need for natural languages facilities in ICAD interfaces has been advanced too [20]. In our approach, the role of intelligent interfaces based on clear semantic notions is highlighted as the core of functionality in an environment for supporting ICAD applications. Throughout the interface, knowledge is acquired as required by the knowledge representation structure in which conceptual and graphical knowledge have an interrelated and active role.

The representational system presented here is oriented to a particular design domain: geometric reasoning for geometric modelling. This particular topic has been subject of considerable research [1,2,15-17]. Though in this paper just a representational system is described, its applications to this design area have been explored. However, the use of this representational system in design is described in other sources [11].

2. An Interactive Session with GRAFLOG.

The system supports a graphic interface for editing lines, as well as a facility for expressing natural language and logical expressions. Through the graphic interaction the user is able to type the following ostensive definition,

These are walls.

at the time a set of lines are drawn, as shown in Figure 1. The graphical symbols are taken from a graphical menu.

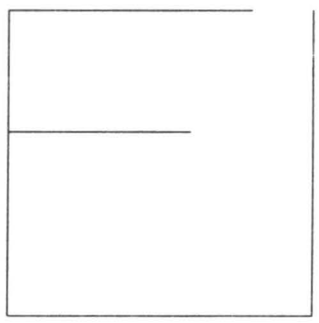


FIGURE 1.

The user can question the meaning of the representation. If at the time either of Figure 2.a, 2.b or 2.c is pointed out in the screen the user types,

What is this?

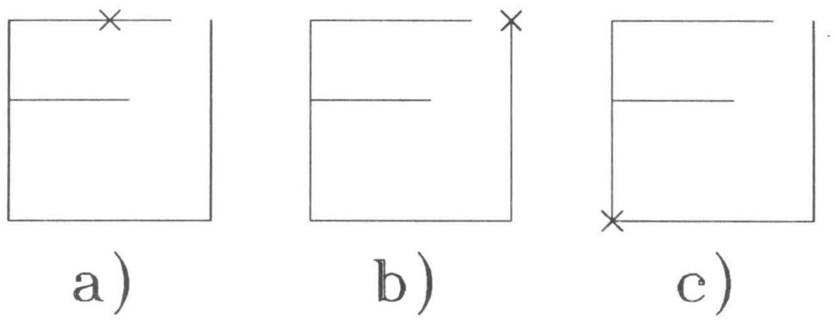


FIGURE 2.

The answers produced by the system are functions of the drawing and the position of the pointing device, and they are *a wall, the origin of this wall* and *the join of these walls* respectively. These linguistic answers are supported by a graphical feedback. The graphical symbols referred to by the words *this* and *these* are highlighted as shown in Figures 3.a, 3.b and 3.c respectively.

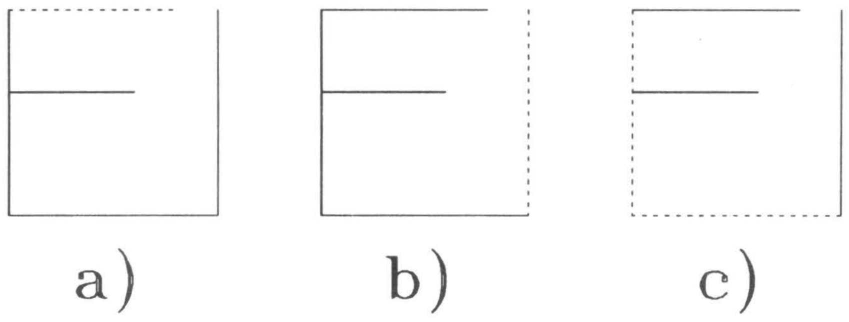


FIGURE 3.

Then the user can type

This is a house

at the time a region of the space is described by identifying a polygon whose vertexes are dots defined in terms of the basic lines, as shown in Figure 4.

Some of the GRAFLOG's features presented here are currently being implemented. We regard this representational system as a theoretical specification.

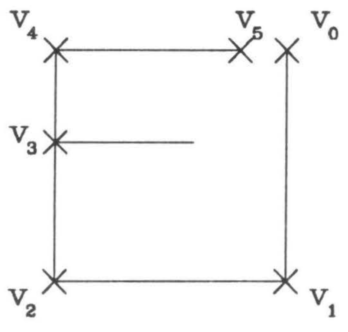


FIGURE 4.

The lines in the representation can be updated by direct manipulation as shown in Figure 5.

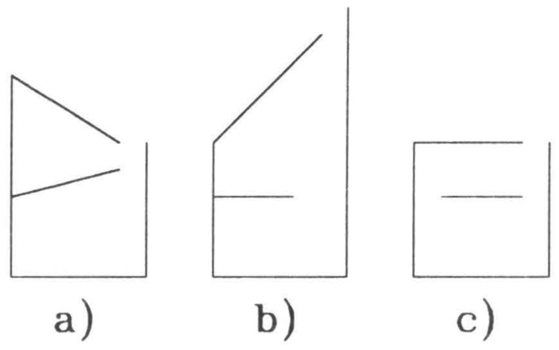


FIGURE 5.

From the point of view of the representational system, Figures 5.a and 5.b are variations of the house, because the topological relations of the original definition are still preserved. However, the drawing of Figure 5.c is no longer considered a house, because one topological relation of the original definition does not hold for this drawing.

Through the interaction, other space partitions can be identified. A room can be identified by typing

This is room.

at the time the marks in Figure 6.a are identified.

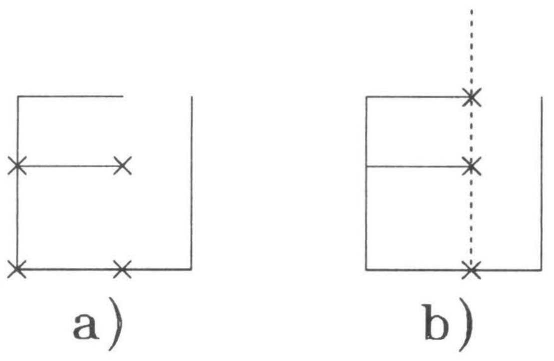


FIGURE 6.

Three dots are identified as the origin or the end of the lines representing the walls; however the fourth vertex is not fully determined by the basic symbols, and it has to be identified with the help of a construction line. If the system finds no reference for one such vertex, the user is prompted for two additional referents by which the construction line is identified as shown in Figure 6.b.

The walls are represented by basic symbols explicitly drawn, but the house and the room are symbols which emerge from the basic ones in terms of the graphical context. The user can ask for both basic and emergent graphical symbols. If at the time Figure 7.a is pointed out the question

What is this?

is asked, the answer is *a house*.

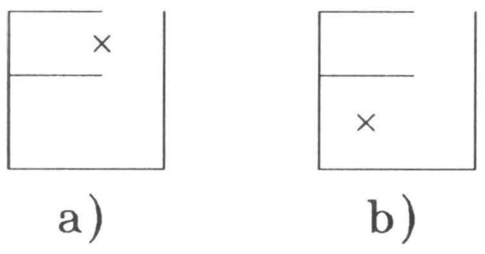


FIGURE 7.

However, if the same question is asked when the mark of the pointing device is placed as shown in Figure 7.b, there is an ambiguity, because the referent might be either the house or the room. For solving this kind of ambiguity the user **intention** has to be considered. If the referent of the pointing act is within a graphical context this kind of ambiguity will always arise. In standard graphic interaction ambiguity does not arise because symbols on the screen are always considered, as it were, independent to the context. Graphical contexts are integrated in human minds, but traditional graphical editors do not have knowledge of such an integration. Through GRAFLOG, the user is able to express his or her pointing intention by typing a logical expression (which is the semantic representation of some natural language expression). If in a context which the drawing in Figure 7.b, the user refers to the room rather than the house consistently, he or she can express the following rule,

$$\begin{aligned}\forall x,y,this: [this = x \wedge room(x)] \vee \\ [this = y \wedge house(y)] \\ \rightarrow this = x.\end{aligned}$$

which means, roughly speaking, *If this is a house or a room then the this selects a room.* When this interpretation has been given, the user is able to ask *What is this?* again, at the time Figure 6.b is pointed out, and the answer provided by the system will be *a room.*

Through this representational system, graphical and linguistic information can be incrementally constructed throughout the interactive session. The graphical and linguistic knowledge is supported in an integrated fashion. GRAFLOG suggests a mode of interaction as natural as human conversation in which natural language and drawings are used for communication and representational purposes throughout design tasks.

3. A Graphical Language for Architectural Drawings: L_{gla}.

In this Section a language, namely L_{gla}, for representing the semantics of drawings is represented. The syntax of L_{gla} is stated by augmenting FOL following the conventions of Enderton [5].

3.1. Syntactic definition of L_{gla}.

A. Logical symbols of L_{gla}

0. Parentheses (,).
1. Sentential connective symbols: $\wedge$, $\vee$, $\rightarrow$, $\equiv$, $\neg$.
2. Variables: **x, y, z, v_0, v_1,...etc.**
3. Equality symbol: =.

B. Parameters of L_{gla}

0. Quantifier symbols: $\forall$, $\exists$.
1. Constant symbols: a set of symbols.
2. Numeral symbols for denoting real numbers.
3. Predicate symbols: For each positive integer *n* a set of symbols called *n-place* predicate symbols.
4. *1*-place geometrical predicate symbols: *vertical, horizontal.*

5. 2-place geometrical predicate symbols: *perpendicular, parallel, collineal, on, in.*

6. *1*-place function symbols: *position_of, length-of, area_of, angle_of, angle_between, origin_of, end_of.*

7. 2-place function symbols: *cross_at, joins_at, leaves_at, joins_origins_at, joins_ends_at, precedes_at, follows_at, would_join_at, would_leave_at, would_join_origins_at, would_join_ends_at, would_precede_at, woud_follow_at, would_be_a_line, union_of, intersection_of, difference_between.*

8. A set of graphical identifying function symbols: *dot, line, polygon*

If ϕ and ψ are a well formed formulas *(wff)* then $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\phi \equiv \psi$, $\neg\phi$, $\forall x\phi$, and $\exists x\phi$ are *wff* as well. The set of *terms* is the set of expressions generated from constant symbols and variables by the syntactic operation of functions. Constant symbols and variables are terms themselves. If α and β are terms then $\alpha = \beta$ is a *wff.*

The arguments of geometrical predicates and functions are terms referring to objects that are graphically representated by symbols of types *dots, lines* and *polygons.* The functions and their arguments are typed terms. There are four types in graphical representations: *values, dots, lines* and *polygons.* The model Π of $\mathbf{L_{gla}}$ is a function mapping the language parameters to the set $|\Pi|$, where $|\Pi|$ is the union of the set of individuals referred to by natural language and the real numbers $\mathbf{R}$. *Graphical values* refer to real numbers, *dots* of the graphical representation denote pairs of real numbers, *lines* denote ordered pairs of dots and *polygons* denote ordered sets of lines.

For clarity, we specify the types of the terms related by geometrical predicates. It is worth realling that these types are relevant for computing the geometrical knowledge involved in the relations, and from the point of view of $\mathbf{L_{gla}}$ they are useful for keeping in mind the kind of object that the predicate arguments refer to in the model.

If α is a term of type *line* then *P.1* and *P.2* are *geometric predicates.*

P.1. *vertical(*α*).*

P.2. *horizontal(*α*).*

If α and β are term of type *line* then *P.3* to *P.5* are *geometric predicates.*

P.3 *perpendicular(*α, β*).*

P.4 *parallel(*α, β*).*

P.5 *collineal(*α, β*).*

P.6 If α is a term of type *dot* and β is a term of type *line* then *on(*α, β*)* is a *geometric predicate.*

P.7. If α is a term of type *dot* or *polygon* and β is a term of type *polygon* then *in(*α, β*)* is a *geometric predicate.*

For clarity, we give the definition of functions using the equality relation. The types of the terms involved in the relation are specified as well. In the definition of 2-place functions we take the convention that the individual denoted by α is always taken as the domain of the function and the one denoted by β is the range of the function.

F.1. If α is a term of type dot *dot* and β is an ordered pair of real numbers then β = *position_of(*α*)* is a *wff.*

F.2. If α is a term of type *line* and β is a real number then β = *length_of(*α*)* is a *wff.*

F.3. If α is a term of type *polygon* and β is a real number then β = *area_of(*α*)* is a *wff.*

F.4. If α is a term of type *line* and β is a real number then β = *angle_of(*α*)* is a *wff.*

F.5. If α and β are terms of type *line* and γ is a real number then γ = *angle_between(*α, β*)* is a *wff.*

F.6. If α is a term of type *line* and β is a term of type *dot* then β = *origin_of(*α*)* is a *wff.*

F.7. If α is a term of type *line* and β is a term of type *dot* then β = *end_of(*α*)* is a *wff.*

If α and β are terms of type *line* and γ is a term of type *dot* then *F.8* to *F.20* are *wff.*

F.8. γ = *cross_at(*α, β*).*

F.9. γ = *joins_at(*α, β*).*

F.10. γ = *leaves_at(*α, β*).*

F.11. γ = *joins_origins_at(*α, β*).*

F.12. γ = *joins_ends_at(*α, β*).*

F.13. γ = *precedes_at(*α, β*).*

F.14. γ = *follows_at(*α, β*).*

F.15. γ = *would_join_at(*α, β*).*

F.16. γ = *would_leave_at(*α, β*).*

F.17. γ = *would_join_origins_at(*α, β*).*

F.18. γ = *would_join_ends_at(*α, β*).*

F.19. γ = *would_precede_at(*α, β*).*

F.20. γ = *would_follow_at(*α, β*).*

F.21. If α and β are terms of type *dot* and γ is a term of type *line* then γ = *would_be_a_line(*α, β*)* is a *wff.*

If α, β and γ are terms of type *polygon* then *F.22* to *F.24* are *wff.*

F.22. γ = *union_of(*α, β*).*

F.23. γ = *intersection_of(*α, β*).*

F.24. γ = *difference_between*(α, β).

We introduce a set of function for explicitly naming graphical objects within $\mathbf{L_{gla}}$.

E.1. If α denotes a term of type dot and *a* a constant symbol then *a* = *dot*(α) is a *wff*.

E.2. If α is a term of type line and *a* a constant symbol then *a* = *line*(α) is a *wff*.

E.3. If α is a term of type polygon and *a* a constant symbol then *a* = *polygon*(α) is a *wff*.

3.1.1. Semantics of $\mathbf{L_{gla}}$.

Geometric Predicates *P.1* to *P.7* assert relations between graphical entities and some primitive geometric property that can be known from geometrical knowledge. Predicates take as arguments names of individuals which are graphically referred to by a graphical symbol. Geometrical predicates take geometric functions as arguments as well. For any geometric function taken as a predicate's argument, the type of the function has to agree with the type of the predicate's argument. Predicates assert geometrical conditions useful for analogical reasoning in design.

Geometrical knowledge represented through geometrical predicates and functions is captured by standard geometrical analysis. The knowledge of functions and relations involving two individuals denoted by symbols of type line is captured throughout the parametric equations of two arbitrarily defined vectors in the space:

$$\mathbf{p = u_0 + t_1(u_1 - u_0).}$$
$$\mathbf{p = v_0 + t_2(v_1 - v_0).}$$

Where vector *A* is $\mathbf{<u_1 - u_0>}$ and vector *B* is $\mathbf{<v_1 - v_0>}$ and **p** is the intersection point, as shown in Figure 8.

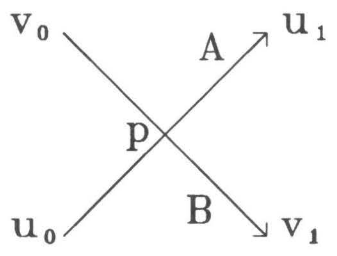


FIGURE 8.

The parameters' value are classified in five interesting cases: **$t < 0$, $t = 0$, $0 < t < 1$, $t = 1$** and **$t > 1$.** Within this classification there are 25 cases for parameter pairs. For example, the pair **($0 < t_1 < 1$, $0 < t_2 < 1$)** represents the *cross_at* function, the pair **($t_1 = 1$, $0 < t_2 < 1$)** represents the *joins_at* function. Combinations for the discrete parameters value **$t = 0$** and **$t = 1$** for both parameters determine functions *F.11, F.12, F.13, F.14.* When one or both of the parameters take a value in either **$t < 0$** or **$t > 1$** there is no current intersection and one of the *would F.15* to *F.20* functions is true of the given vector's relationship. Cases not explicitly listed can be found by interchanging the domain and range vectors in the relations. This analysis besides being simple and robust, leads to efficient implementations [8]. In Figure 9 these relations are graphically shown in terms of the parameters' pair cases. For clarity, the α individual is horizontally oriented, and the β individual is vertically oriented.

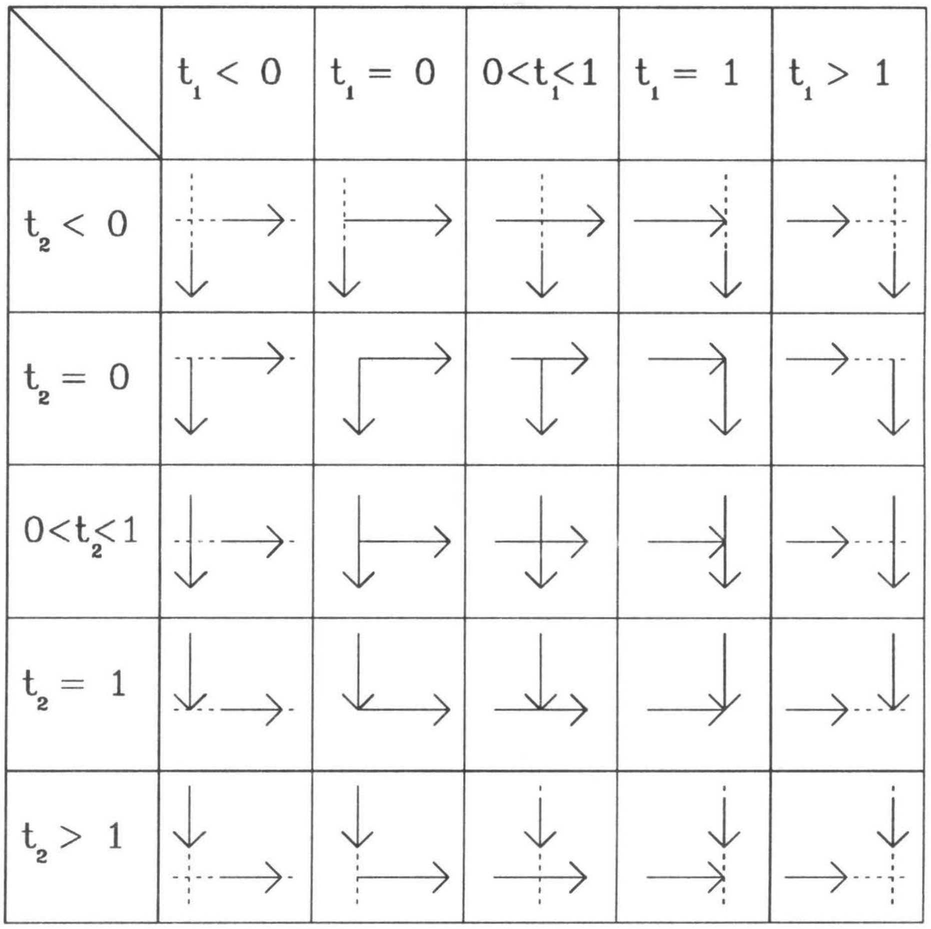


FIGURE 9

Functions *F.8* to *F.14* assert relations between two intersecting oriented lines.

Relations between oriented lines are instances of the more general cases of *t_join_at* and *extreme_join_at* for which orientation does not matter, and geometric theorems of $\mathbf{L_{gla}}$ are:

$\forall$x,y,z: [z = joins_at(x, y) $\vee$

$$\begin{aligned} & z = \mathbf{joins\_at}(y, x) \lor \\ & z = \mathbf{leaves\_at}(x, y) \lor \\ & z = \mathbf{leaves\_at}(y, x)] \\ & \rightarrow z = \mathbf{t\_join\_at}(x, y). \end{aligned}$$

and

$$\begin{aligned} \forall x,y,z{:}\ [& z = \mathbf{joins\_origins\_at}(x, y) \lor \\ & z = \mathbf{joins\_ends\_at}(x, y) \lor \\ & z = \mathbf{precedes\_at}(x, y) \lor \\ & z = \mathbf{follows\_at}(x, y)] \\ & \rightarrow z = \mathbf{extreme\_join\_at}(x, y). \end{aligned}$$

and the more general case from a linguistic point of view

$$\forall x,y,z{:}\ [z = \mathbf{t\_join\_at}(x, y) \lor z = \mathbf{extreme\_join\_at}(x, y)] \rightarrow z = \mathbf{join\_at}(x, y).$$

Functions *F.15* to *F.20* assert relations between two lines which *would* intersect if one or both of them were projected in their own directions. These are conditional intersections, and they represent the principle underlying the definition of construction lines in architectural and other kinds of drawings. As will be shown, these functions will prove to be very useful for the process of analogical reasoning in design. these functions are instances of the more general cases of *would_be_t_join_at* and *extreme_join_at* for which orientation does not matter, and geometric theorems of $\mathbf{L}_{\mathbf{gla}}$ are:

$$\begin{aligned} \forall x,y,z{:}\ [& z = \mathbf{would\_join\_at}(x, y) \lor \\ & z = \mathbf{would\_join\_at}(y, x) \lor \\ & z = \mathbf{would\_leave\_at}(x, y) \lor \\ & z = \mathbf{would\_leave\_at}(y, x)] \\ & \rightarrow z = \mathbf{would\_be\_t\_join\_at}(x, y). \end{aligned}$$

and

$$\begin{aligned} \forall x,y,z{:}\ [& z = \mathbf{would\_join\_origins\_at}(x, y) \lor \\ & z = \mathbf{would\_join\_ends\_at}(x, y) \lor \\ & z = \mathbf{would\_precede\_at}(x, y) \lor \\ & z = \mathbf{would\_follow\_at}(x, y)] \\ & \rightarrow z = \mathbf{would\_be\_extreme\_join\_at}(x, y). \end{aligned}$$

and the more general case from a linguistic point of view

$$\begin{aligned} \forall x,y,z{:}\ [& z = \mathbf{would\_be\_t\_join\_at}(x, y) \lor (z = \mathbf{would\_be\_extreme\_join\_at}(x, y)] \\ & \rightarrow z = \mathbf{would\_join\_at}(x, y). \end{aligned}$$

Function *F.21* takes as arguments two dots and produce a line. This function is also useful for defining construction lines for architectural drawings.

Functions *F.22* to *F.24* define topological operation between polygons. The inclusion of graphical symbols of type polygons allows the definition of complex regions of the space by composition of simple polygons. In particular, the *union_of, intersection_of* and *difference_between* set operations among two arbitrary polygons in the space define a closed algebra for polygons defined as regular sets [18]. The output of these operations is shown in Figure 10.

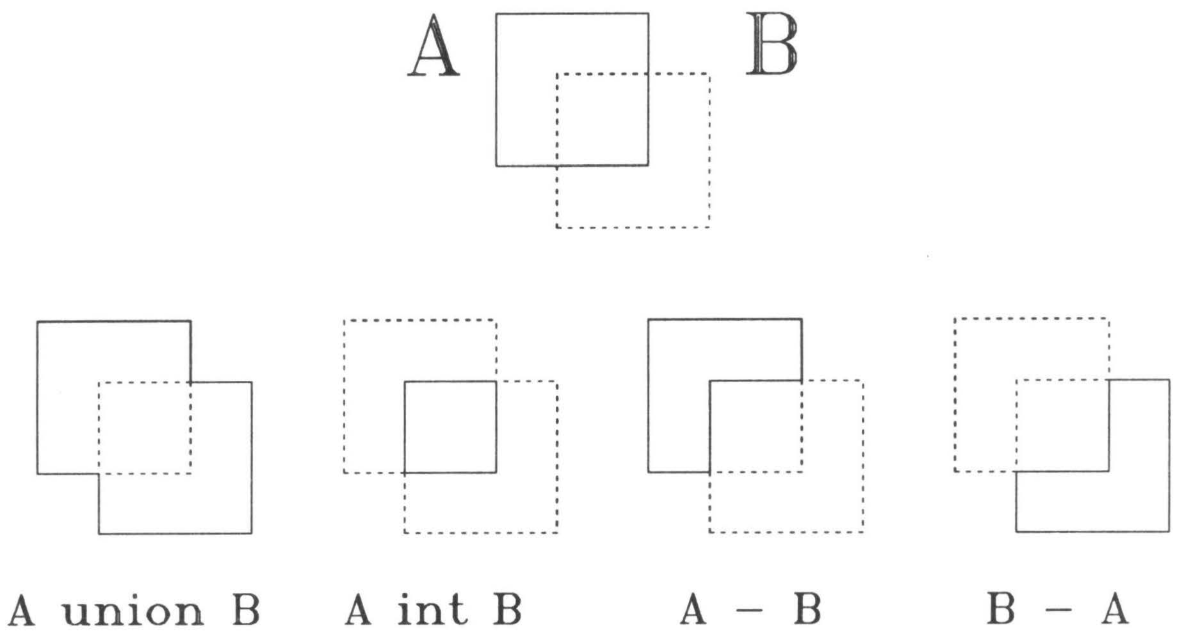


FIGURE 10.

Algorithms for comparing two arbitrary polygons can be efficiently implemented; for example [8, 19].

The geometrical identifying functions *E.1* and *E.2* for naming dots and lines are just type checking functions: they are restricted to verify that their arguments are terms referring to dots or lines. It is worth noticing that such terms can either be given directly or produced by the syntactic operation of functions. The identifying function *E.3* verifies that the graphical object determines a well formed polygon. A theoretical definition of this kind of object can be given [8].

These relations define a structure for graphics. Through this structure two kind of entities can be defined: basic and emergent. Basic entities are introduced in a context independent manner. They are, for example, basic symbols taken from a graphic menu. Basic graphical symbols can be given a name when they are introduced by means of an explicit ostensive definition . [9] On the other hand, there are some entities that emerge from graphical context, as *rooms* emerge from *walls* in architectural drawings. In particular, space partitions are always context dependent. Context dependent entities have properties that might be relevant for the interpretation of drawings; for instance, *a room* in an architectural drawing has *an area.* These kind of properties are capture by $\mathbf{L_{gla}}$. Context dependent graphical entities emerging from basic symbols might not have a name, and the place for the linguistic argument of this function should be filled by a variable name. But of course, when a graphical context has been specified graphical entities emerging from the structure might be given an explicit name by ostension, by means of the identifying functions *E.1* to *E.3*.

We define *lines* as the only basic type for symbols. Symbols of type *dot* and *polygon* will be defined always in terms of the context from which they emerge.

Suppose that there is a line whose name is *wall_1*. The geometrical information of that line is kept in a data-base of geometrical objects. The name *line_1* is the index of the object in the data-base. We define a meta-theoretical **translation function** that relates the linguistic name of an object with the index to a data-base where its geometrical information is stored. The translation function for that line is:

translation(wall_1, line_1).

4. Primitive and Emerging Concepts in $\mathbf{L_{gla}}$.

In Figure 11 a plan of a house is shown. The lines are chosen from an icons menu, and a basic interpretation for each line is introduced by means of an ostensive definition as was shown in Section 1.

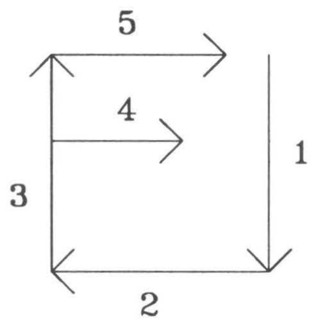


FIGURE 11.

A name of $\mathbf{L_{gla}}$ and a graphical identifier for each graphical symbol are shown in Figure 11. The translation function for this drawing is:

translates(right_wall, 1).
translates(back_wall, 2).
translates(left_wall, 3).
translates(interior_wall, 4).
translates(front_wall, 5).

By means of $\mathbf{L_{gla}}$ we can say, for example,

∃x,y: y = t_joins(x, left_wall)

which will be true if the variable **x** and *interior_wall* denote the same entity. The language $\mathbf{L_{gla}}$ allows the expression of simple concepts that are directly know from the structure of $\mathbf{L_{gla}}$ for any particular drawing defined within the structure.

We can think of the graphical structure as a kind of conceptual lexicon for graphical representations. The set of geometrical predicates and functions gives an intentional account of all possible graphical entities that have the properties and stand in the relations asserted by predicates and functions of $\mathbf{L_{gla}}$. For instance, by predicate *P.3* we can know that

perpendicular(back_wall, left_wall)

is true of the drawing in Figure 11.

4.1.1. Ambiguity of Graphical Concepts.

Dealing with individuals in abstraction from their substantial realisation can produce some ambiguity within the representation. In fact, every individual represented by a graphical symbol of type dot will have a position and that can be known from function F.1. This geometrical property might have the same value for different expressions of $\mathbf{L_{gla}}$ and a single drawing. For instance, a dot in a representation might involve several different concepts, or emerge from the structure in the context of different functions. In Figure 12 there is a mark referring to an object graphically represented by a symbol of type dot, but what entity is it?

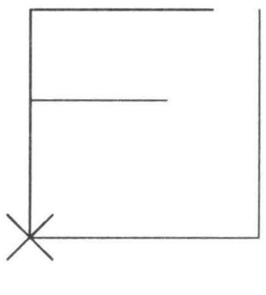


FIGURE 12.

In terms of the structure it might be:

by function F.6: ∃x: x = origin_of(left_wall).
by function F.7: ∃x: x = end_of(back_wall).
by function F.13: ∃x: x = precedes_at(back_wall, left_wall).
by function f.14: ∃x: x = follows_at(left_wall, back_wall).

And also, it can be in terms of the graphical theorems:

$\exists$x: x = extreme_join_at(back_wall, left_wall).

$\exists$x: x = extreme_join_at(left_wall, back_wall).

$\exists$x: x = join_at(back_wall, left_wall).

$\exists$x: x = join_at(left_wall, back_wall).

4.2. Ambiguity Resolution of Graphical Concepts.

For purposes of identification, to chose one or another individual makes a difference, and we can classify possible answers for the graphical referents in terms of three notions: levels of abstraction, information content and possible world.

We can consider two levels of abstraction: the geometrical and the linguistic. The referent realised directly from the graphical structure (the geometrical level) implies the direction of the vectors in the graphical representation. This level is useful for geometric modelling purposes. On the other hand, the linguistic concept hides the notion of orientation, because it might not be relevant for a linguistic discourse. This distinction is important for graphic interaction, because some users interested in specification tasks might think of the graphical representation in terms of high level concepts, and others might be more concerned explicitly with the design task, and then the orientation of vectors might be very useful information for the definition of transformation functions in modelling design tasks.

The information content criteria might be explained considering the following example: if the question *what is this?* is asked when the mark is placed as shown in Figure 12 then *It is where the back wall precedes the left wall* would be a better answer than than *It is the end of back wall.* The former answer is richer in its **information content** than the latter. Each function has an information content value, and most likely answers can be given in terms of the fundamental principle of information theory, that is, the amount of information in a message is an inverse proportion of the probability of occurrence of the message. The formulas α = *origin_of(wall_1)* and α = *end_of(wall_1)* are necessarily true for some α if *wall_1* exists, and their information content is then nil. On the other hand, the formula α = *follows_at(back_wall, precede_wall)* is true for the drawing in Figure 12, but it might be false for other dispositions of the house's walls. If this relation is true for some set of possible contexts and false for others, it has to have a larger information content than other relations that are necessarily true for every graphical context.

Is worth noticing the fact that expressions at the linguistic level of abstraction, as for instance *extreme_join_at, join_at* are less informative than basic geometric predicates and functions, except for those with no information at all. This is consistent with a notion of expressiveness in natural language: the more comprehensive the less specific [6]. In natural language the greater the abstraction the larger the intention and the lower the abstraction the greater the extension.

This might be a paradox, but the power of language is due, at least in some degree, to the abstractions that can be expressed, and in the abstraction process, irrelevant information must be filtered out. To be expressive is not to be informative. And this is also true for graphics: if the arrows representing the vectors' orientation are dropped, information is lost, but expressivity is increased.

The third mentioned criteria that can be used for solving graphical ambiguities is the notion of possible world. It is illustrated by the conditional functions whose names are prefixed by *would.* If the question *What is this?* is asked when the mark is put as shown in Figure 13.a the answer has to be *It is the origin of the right wall.*

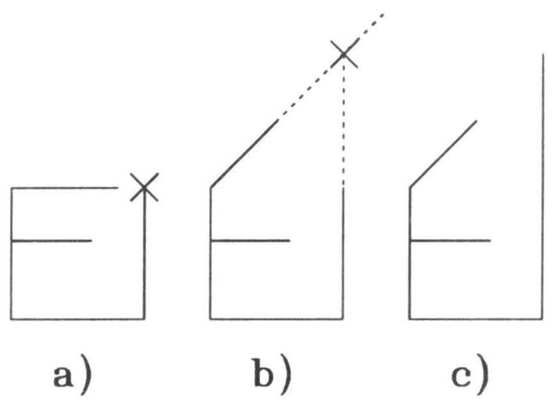


FIGURE 13.

However, the answer can also be *It is where the front wall would precede the right wall if the former wall were projected in its own direction.* If at the time the question were asked the latter rather than the former were the desired answer, then the *would* rule would have indeed a very large information content value: it is highly unlikely that someone were interested in such an eccentricity. However, in other circumstances this answer might be highly desirable. For example, if looking at Figure 13.b the same question were asked, we would have to answer *It would be where the front and the right wall join if they were projected in their orientations.* The dotted lines are construction lines commonly used in architectural drawings and they are useful for the production of the drawing but not within the representation of the object itself. This situation might arise if the house in Figure 13.a were modified in the course of the design process to the one shown in Figure 13.b. But suppose that the designer has the intention to modify the right wall as well, as shown in Figure 13.c. The designer would have to draw the needed construction lines before extending the right wall to the yet unknown location. The *would function* produces the new origin of the right wall from the current graphically represented individuals.

Though individuals identified by conditional rules have a larger information content than the individuals produced by the other rules of the graphical structure, we can assume the convention that conditional rules only apply for identification purposes if there is no individual graphically referred to by a deictic question in the current context or state of affairs. The criterion of possible world is not useful for talking about what is true now, but for talking about what would be true if the situation were different.

5. Relation between Deictic Functions and Graphical Structure.

In this Section a formal notion for the representation and evaluation of deictic questions such as *What is this?* or *Who is this?* in terms of the graphical structure and $\mathbf{L}_{\mathbf{gla}}$ is presented.

5.1. Purpose of Deictic Functions.

The representation and evaluation of deictic expressions are considered here as meta-linguistic notions. Modelling deictic expressions requieres the definition of an interfacing device for relating the graphical and linguistic domains. This interfacing component receives as inputs a linguistic expression (for example, an ostensive definition or a deictic question) and a referent provided by means of a pointing device: a point in the space which is to be considered the spatial referent. The output of this module is a formula of $\mathbf{L}_{\mathbf{gla}}$.

In the interpretation of deictic functions we can identify two kinds of things: if a basic symbol of type line is chosen, then the purpose of the function is to identify the individual that is represented by such a graphical symbol. On the other hand, if the chosen individual emerges from some basic individuals in terms of the graphical structure, the purpose of the deictic function will be to identify the concept producing the graphically referenced individual. Rules for defining more complex deictic functions involving the deictic pronouns *who, which,* etc., have to be defined in terms of additional linguistic information represented through $\mathbf{L}_{\mathbf{gla}}$ as will be shown below.

5.2. Definition of Deictic Identification Rules.

In this section the interfacing component between the graphical and linguistic domain by which deictic questions are answered is defined. This module is stated by defining a set of deictic identification rules. The purpose of these rules is, as was mentioned, to identify through its linguistic name an entity that is graphically referred to when a deictic question is asked.

For identification purposes, we take the convention that lines are considered open intervals (they do not include their extreme points) and polygons are considered regular sets [18]. If only individuals represented by symbols of type line are denoted in a context independent manner, as is in fact the case, this convention establishes a mutual exclusion between basic and emergent symbols within the graphical structure.

We define a deictic meta-linguistic function, namely *deictic_functor* as a function of the form

$$\mathbf{deictic\_functor(\Psi, mark) \rightarrow \phi}$$

where ϕ is a *wff* of $\mathbf{L_{gla}}$ and Ψ is a function symbol of $\mathbf{L_{gla}}$ and *mark* is the position of the pointing device. We take the convention that within ϕ there is an existentially quantified identifying variable, namely *this* which has as a denotation either the individual represented by the symbol which is pointed out in the graphical domain, or the configuration that is referred to by a term produced by the syntactic operation of a function of $\mathbf{L_{gla}}$. We also take the convention that the only meta-linguistic parameter considered for the definition of the identification rules is the position of pointing mark. The identification rules are presented as an implication in which the antecedent is a *wff* of $\mathbf{L_{gla}}$ augmented with the graphical parameter *mark*, and the consequent is the formula of $\mathbf{L_{gla}}$ produced by the the deictic functor. The identification rules are defined as,

D.1. If $\exists x\text{: } on(x, \alpha) \wedge mark = position(x)$ then $\exists this\text{: } this = \alpha$.

D.2. If $\exists x\text{: } x = \Psi(\alpha) \wedge mark = position(x)$ then $\exists this\text{: } this = \Psi(\alpha)$. Where Ψ is variable standing for a function symbol in *F.6* or *F.7* of $\mathbf{L_{gla}}$ and,

D.3. If $\exists x\text{: } x = \Phi(\alpha, \beta) \wedge mark = position(x)$ then $\exists this\text{: } this = \Phi(\alpha, \beta)$ where Φ is a variable standing for a function symbol defined in any of *F.8* to *F.20* of $\mathbf{L_{gla}}$.

D.4. If $\exists x\text{: } in(x, \alpha) \wedge mark = position(x)$ then $\exists this\text{: } this = \alpha$ where α is graphical represented by a symbol of type polygon.

Identification rule *D.1* produces an individual represented by a symbol of type line. Rules *D.2* and *D.3* produce graphical concepts. The question *what is this* is translated to $\mathbf{L_{gla}}$ as

$$\mathbf{\exists this, what\text{: } this = what.}$$

where *this* and *what* are variable names. If the question is asked when looking at Figure 14.a -assuming the names given in Figure 11- then the answer would have to be *what = front_wall*, because by identification rule *D.1* the meta-theoretical formula

$$\mathbf{\exists what\text{: } on(what, front\_wall) \wedge mark = position(what) \rightarrow \exists this\text{: } this = front\_wall.}$$

is true. The variable *what* denotes the same object that the constant symbol *front_wall*.

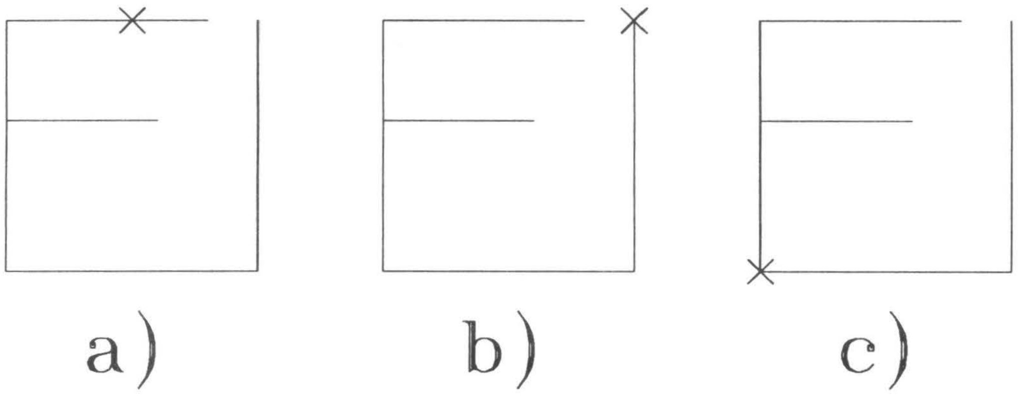


FIGURE 14.

If the same question is asked when looking at Figure 14.b, then

$$\exists \textbf{what: what = origin\_of(right\_wall)} \wedge \textbf{mark = position(what)}$$
$$\rightarrow \exists \textbf{this: this = origin\_of(right\_wall).}$$

is true. This is an example of the application of the identification rule *D.2*.

If the question is asked when looking at Figure 14.c, then

$$\exists \textbf{what: what = follows\_at(left\_wall, back\_wall)} \wedge \textbf{mark = position(what)}$$
$$\rightarrow \exists \textbf{this: this = follows\_at(left\_wall, back\_wall).}$$

is true. This is an example of the application of the identification rule *D.3*. The use of identification rule *D.4* will be illustrated when the representation of regions of the space is introduced.

Through the identification deictic rules the meaning of wh-words can be modelled in $\mathbf{L_{gla}}$; for instance, if the wh-word *what* asks for the property that an individual has, it can be modelled through the the rule,

$$\forall \textbf{this, what, property: this = what} \wedge \textbf{property(what)}$$
$$\rightarrow \textbf{is(this, property).}$$

The meaning of the wh-word *who* can be modelled in similar fashion:

$$\forall \textbf{this, who, property: this = who} \wedge \textbf{property(who)} \wedge \textbf{animate(property)}$$
$$\rightarrow \textbf{is(this, who).}$$

The linguistic *is* is a 2-place predicate symbol of $\mathbf{L_{gla}}$ and performs different roles in the two mentioned rules. In natural language discourse, the kind of function that the verb *to be* performs, for example, asserting an identity relation between two individuals, a class inclusion relation between two classes, or class membership classification relation between an individual and a class, should be solved by means of contextual information. Here, different functions of the verb *to be* are determined by the logical formula from which they are produced.

Note that these rules imply a second order relation because the predicative variable *property* is universally quantified. We can allow this kind of expressions, though some of the logical properties of $\mathbf{L_{gla}}$ are lost. However, the larger expressive power of the representational system is highly desirable. Furthermore, for implementation purposes and considering finite domains these rules can be effectively modelled.

5.2.1. Proper and Common Nouns in deictic Expressions.

For practical purposes, we define some conventions for naming things within the representational system. Some individuals are named by means of ostensive definitions. There are two kind of names: proper and common nouns. When the linguistic argument of the ostensive definition is a **proper noun**, that noun is the name of the individual; but if this argument is a **common noun** it is rather naming a property that the individual has. In case of the architectural drawings, the linguistic argument for an ostensive definition is usually a common noun. It is quite odd to say *This is left wall.* The natural thing to say is *This is a wall.*

For the purpose of our translation function, if the argument of an ostensive definition is a proper noun, that noun is the argument of the corresponding translation function; but if the argument is a common noun, the corresponding place in the representation function is filled by an arbitrary identifier, which is not visible to the user. When the definition *This is a wall* is stated at the time one line is inserted in the drawing in a context independent way, the functions *translate(wall_1, line_id)* and *wall(wall_1)* are produced. The symbol *wall_1* is an arbitrary identifier. The linguistic information asserted in $\mathbf{L_{gla}}$ is the property of the individual manifested by the ostensive definition. For implementational purposes, the translation function do not have to be stored if we take the convention that the name of the individual within $\mathbf{L_{gla}}$ and the pointer or index to the corresponding instance of the graphical objects data-base is the same. For instance, if graphical software is GKS[4] the name of the individual corresponds with the identifier of the GKS segment where the graphical information is stored.

The way the individual is named (by a proper or a common noun) is also relevant for answering deictic questions. If the question *What is this?* is asked while one of the walls is pointed out in Figure 14.a then the answer would have to be *It is a wall.* If the individual is referred to by a common noun, the answer to the question is produced by producing the property that the individual has preceded by the indefinite article *a.*

The distinction between common and proper nouns allows us to model singular and plural ostensive definitions and deictic questions as well. Proper nouns imply singular expressions; on the other hand, common nouns might be introduced by singular or plural expressions. In the drawing of the house the singular expression can be stated when a linguistic interpretation is introduced for each of the walls in the drawing. However, a single plural expression can be used for introducing the interpretation of all the individuals with the same property. Looking at the house drawing, the definition *These are walls* can be stated at the time the different walls in the drawing are sequentially pointed out. Of course *wall* must be classified in the linguistic lexicon as a common noun, and the morphological rule for producing the plural case has to be available.

It is worth noticing that in the context of graphical interaction the full text of deictic expressions (singular of plural) does not have to be typed. Expressions as *this is, this is a, what is this?, these are* and *what are these?* can be associated to codes for triggering the pick or locator devices, and different graphic interactive techniques can be used for developing a dynamic and flexible interactive dialogue. Voice recognisers might be introduced for this particular aspect of graphic interaction. Speech recognition of a highly specific set of expressions might be a constrained enough problem for the definition of practical applications, and there is at least one interesting precedent[3] though it lacked of well founded semantics, and was developed for the American Navy. With specific speech facilities in human-computer interfaces the quality of interaction might be highly improved.

With these considerations, the drawing of the house shown in Figure 1 is introduced by typing the plural ostensive definition *These are walls* followed by selecting the lines from the graphical menu and placing each instance on the drawing. The translation function would then be,

```
translates(wall_1, 1).
translates(wall_2, 2).
translates(wall_3, 3).
translates(wall_4, 4).
translates(wall_5, 5).
```

and the linguistic representation in L_{gla} would have to be,

```
wall(wall_1).
wall(wall_2).
wall(wall_3).
wall(wall_4).
wall(wall_5).
```

6. Identification Function for Graphical Context Dependent Individuals.

With the structure developed, we are in a position to introduce the context dependent individuals referred in Section 1. In Figure 15 an architectural drawing of a house is shown again.

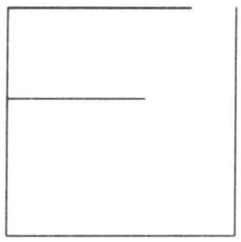


FIGURE 15.

Now suppose that we want the define the house. We can say *This is a house* at the time the house is pointed out, but what has to be picked out from the graphical domain? It is worth recalling that the basic symbols, the walls, were introduced in a context independent manner, and selected from an icons menu. However, in this definition the graphical referent is an arbitrarily defined region of the space. The interpretation for the house is imposed on the drawing by referring to some basic individuals and to the way they are related. The room can be identified, for instance, by a rectangle whose top-right and bottom-left corners are the free extreme of the interior wall and the extreme join between the back and left walls respectively. In this section this identification process is formalised in terms of the graphical structure defined by $\mathbf{L_{gla}}$ by means of the identifying rule *E.3*.

In general, we can say that a region of the space can be identified by imposing a polygon upon it. The polygon might not be overt in the drawing, but its constituting vertexes must be dots emerging from the basic symbols of type line that are explicitly marked on the drawing. It is worth recalling that in the structure defined by $\mathbf{L_{gla}}$ there are not basic individuals represented by symbols of type dot. Every dot has to emerge from a concept within the structure, and has to be identified by identification rules *D.2* and *D.3*

Intuitively, the house has to be identified in terms of the primitive concepts arising from the walls. If at the time the ostensive definition *This is a house* is typed in, the house can be graphically identified by picking one ofter another all the points contributing for the house definition as shown in Figure 16.

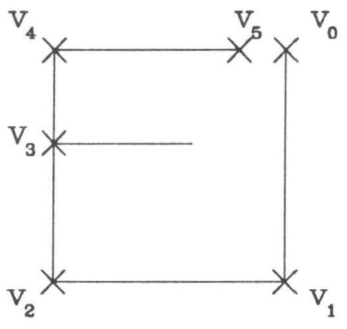


FIGURE 16.

When the house is identified the linguistic fact

house(house_1).

has to be asserted in $\mathbf{L_{gla}}$, and and a graphical object of type polygon can be given a name by the identifying rule *E.3.* The formula by which the emergent spatial entity -the polygon- is identified, can be automatical deduced by the interpreter by means of identification rules *D.2* and *D.3*, and ambiguities can be solved by means of the three criteria shown.

When the ostensive definition is expressed and the vertexes are identified by the identification rules, the formula

$$\begin{aligned}
&\forall \mathbf{v_0, v_1, v_2, v_3, v_4, v_5}:\\
&\quad \mathbf{v_0 = origin\_of(wall\_1)} \wedge\\
&\quad \mathbf{v_1 = follows\_at(wall\_2, wall\_1)} \wedge\\
&\quad \mathbf{v_2 = follows\_at(wall\_3, wall\_2)} \wedge\\
&\quad \mathbf{v_3 = leaves\_at(wall\_4, wall\_3)} \wedge\\
&\quad \mathbf{v_4 = follows\_at(wall\_5, wall\_3)} \wedge\\
&\quad \mathbf{v_5 = end\_of(wall\_5)} \wedge\\
&\quad \mathbf{house\_1 = polygon(v_0, v_1, v_2, v_3, v_4, v_5)}.
\end{aligned}$$

is asserted in $\mathbf{L_{gla}}$.

The translation function is implied if the same symbol is a name in $\mathbf{L_{gla}}$ and also the index to the graphical data-base. It is worth noticing that no instance of this polygon needs to be kept in the data-base of graphical objects. In fact it must not be there. The data-base must keep a representation of the basic graphical symbols, that is symbols of type line; but polygons are fully determined by the formula of $\mathbf{L_{gla}}$. This formula makes a reference to some basic symbol through equality relations and geometrical function symbols.

6.1. Conditional Identification and Construction Lines.

Note that the identification function is not defined in terms of the coordinated values of the references, but in terms of the individual represented by a symbol of type dot which emerges from the graphical structure. This is an important consideration when graphical representations are modified. For any transformation in which all the concepts referred to by the identification function have a denotation and are true, the polygon is well defined, and the corresponding linguistic proposition asserting that individual in $\mathbf{L_{gla}}$ will be also true. If the house is modified as shown in Figure 17a and 17.b

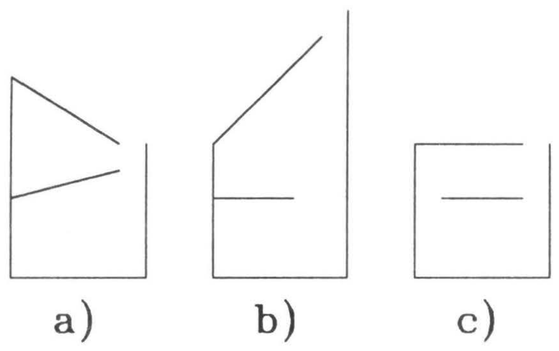


FIGURE 17.

then

house(house_1)

is still true in $\mathbf{L_{gla}}$ because the constant *house_1* has a denotation and this is known because the function *E.3* succedes in identifying the polygon. However, if the house is modified as shown in Figure 18.c, the same proposition in $\mathbf{L_{gla}}$ could not be evaluated because the polygon is not properly defined in the graphical representation. One of the concepts referred to by the identification formula has no denotation in the current drawing, as shown in Figure 18.c, namely

$\mathbf{v_3}$ **= leaves_at(wall_4, wall_3).**

is false, and the whole formula through which the house is identified is false as well.

If the definition of the identification function for some polygon cannot succeed in terms of the current graphical situation, rule *D.3* takes as argument one of the conditional functions from *F.15* to *F.20*. In Figure 18 a possible wall configuration is shown.

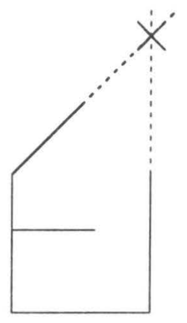


FIGURE 18.

If the mark in Figure 18 is given as one of the polygon parameters, the identification function would have to include, for example, function *F.20*

$\mathbf{v_i}$ **= would_follow_at(wall_1, wall_5).**

The notion of graphical structure allows the identification of more complex objects. If the expression *this is a bedroom* is typed at the time the vertexes of a polygon are identified as shown in Figure 19.a, the space region graphically represented by that bedroom is identified as well. Three of these vertexes are correlated with other dots emerging from the structure, and they can be found by straight application of identification rule *D.3*. However, there is no dot in the structure to match the fourth mark, and the only information provided is that it is in the intersection between *wall_2* and a construction line defined by the extremes of *wall_4* and *wall_5*.

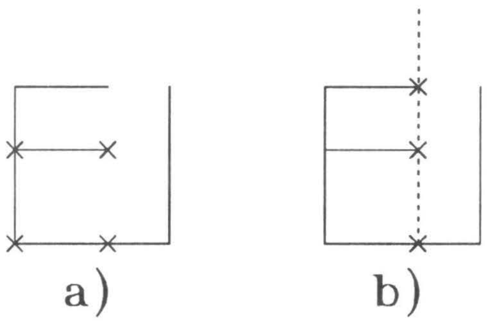


FIGURE 19.

For identifying the fourth vertex a combination of a *would* rule and the function *F.21* of $\mathbf{L_{gla}}$ have to be used. The fourth vertex can be defined in terms of a construction line and its intersection with the wall referred to by the pointing mark by the formula

$$\forall v_0, v_1, l_1: v_0 = \text{end\_of(wall\_5)} \wedge v_1 = \text{end\_of(wall\_4)} \wedge l_1 = \text{would\_be\_a\_line}(v_0, v_1) \rightarrow v_i = \text{would\_join\_at}(l_1, \text{wall\_2}).$$

For the definition of these complex concepts an additional identification rule *D.5* is stated.

D.5. If the identification of v_i of type dot does not succeed by application of rules *D.2* and *D.3* then if v_0 and v_1 are dots identified by rules *D.2* and *D.3* and $\exists v_i$: $on(v_i, \alpha) \wedge mark = position(v_i)$ and $\beta = would\_be\_a\_line(v_0, v_1)$ then $v_i = \Phi(\alpha, \beta)$, where Φ is a function from *F.8* to *F.20*. Ambiguity is solved by means of the information content criteria.

The concept for identifying the fourth vertex of the room can be found by rule *D.5* as shown in Figure 19.b. However, the use of rule *D.5* implies the uses of the other identification rules, and for this two additional graphical referents have to be given. When in the course of an interactive session the interpreter realises the need for rule *D.5*, and that happens when the identification of a graphical referent fails by means of other identification rules, the user is prompted to give two additional graphical referents, that in turn are identified by the graphical interpreter. Rule *D.5* is defined as the last resource for the definition of construction lines, and it can only fail if the lines involved in the rule are parallel.

Once the spatial context dependent individuals have been defined, they can be identified by identification rule *D.4*. If the question *what is this?* is asked at the time Figures 20.a and 20.b are pointed out, the answers will be, *it is a house* and *it is a room* respectively.

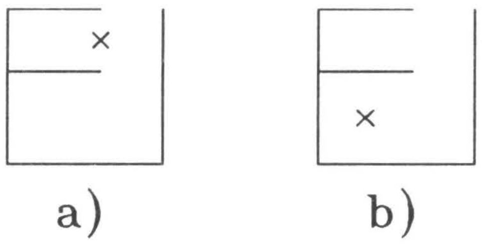


FIGURE 20.

Ambiguity can arise because the graphical identifying individual can be inside several graphical objects, as is the case in Figure 20.b. As was shown, linguistic information can be relevant for the solution of this kind of ambiguity, and rules for this purpose can be expressed through $\mathbf{L_{gla}}$. For example, we can say that *If this is a room and also a house then this is a room.* Is worth recalling that the referent of a deictic expression is never realised in terms of the graphical structure unless it is pointed out in a context independent manner. The ambiguity can only be resolved in terms of the **intention** behind the pointing act, and this intention can be objectively expressed by means of natural language. This sentence can be expressed in $\mathbf{L_{gla}}$ as,

$$\forall \mathbf{x,y,this: this = x} \wedge \mathbf{this = y} \wedge \mathbf{room(x)} \wedge \mathbf{house(y)}$$
$$\rightarrow \mathbf{is(this, x).}$$

where the variable *this* is produced by identification rule *D.4* and *is* is, as was mentioned, a linguistic predicate.

In general, defining or referring to a graphical representation is a behaviour that can be only understood in the context of a "knowing act", and knowing can not be anything but an intentional behaviour.

7. Concluding Remarks.

In this paper a graphical and logical language for representing architectural drawings has been presented. The language is useful for the representing the semantics of drawings made out of dots, lines and polygons. Through the language, basic and emerging graphical symbols can be given a semantic interpretation. The representational system is integrated to the interactive interface and the semantic interpretation of drawings is introduced by means of direct manipulation on drawings and natural language expressions. The sematic representation of drawings is automatically deduced by the interpreter by means of a set of identification rules that act upon the graphical and linguistic input. These rules are defined in terms of the structure produced by the representational language. In this paper, the use of this representational system for the definition of architectural drawings has been illustrated.

Acknowledgements

Luis Pineda is supported by a grant from the Mexican National Council for Science and Technology (CONACYT-44722) and by the Instituto de Investigaciones Electricas (IIE), Mexico.

References

1. F Arbab, ''A Paradigm for Intelligent CAD,'' pp. 20 - 39 in *Intelligent CAD Systems I*, ed. P.J.W ten Hagen and T. Tomiyama,Springer-Verlag, Berlin (1987).
2. F Arbab and Bin Wang, ''Reasoning About Geometric Constrainsts,'' in *Pre-prints of the Second IFIP WG 5.2 Workshop on Intelligent CAD.*, ed. H. Yoshikawa and T. Holden,The University of Tokio, Tokio (1988).
3. R A Bolt, " ''Put-That-There": Voice and Gesture at the Graphics Interface,'' *ACM Siggraph'80 Conference Proceedings, Computer Graphics* **14**(3)(1980).
4. G Enderle, K Kansy, and G Pfaff, *Computer Graphics Programming: GKS - The Graphics Standard,* Springer-Verlag, Berlin Heidelberg (1984).
5. H B Enderton, *A Mathematic Introduction to Logic,* Academic Press, New York (1972).
6. J Lyons, *Introduction to Theoretical Linguistics,* Cambridge University Press, Cambridge (1968).
7. J McCullough, ''Possition Paper on Intelligent CAD issues,'' in *Pre-prints of the Second IFIP WG 5.2 Workshop on Intelligent CAD.*, ed. H. Yoshikawa and T. Holden,The University of Tokio, Tokio (1988).
8. L A Pineda, *Un Algoritmo General para la Comparaci6n de Polígonos,* Instituto Tecnol6gico de Monterrey, México (1986). M. Sc. dissertation (in Spanish)
9. L A Pineda, E Klein, and J Lee, ''GRAFLOG: Understanding Graphics Through Natural Language,'' *Computer Graphics Forum* **7**(2)(1988).
10. L A Pineda, ''A Compositional Semantic for Graphics,'' in *Eurographics'88 Conference Proceedings*, ed. D. Duce and P. Jancene,Elsevier Science Publishers B. V., North-Holland (1988).
11. L A Pineda, *On the Notions of Syntax and Semantics of Graphical Languages,* University of Edinburgh (1989). Forthcoming Ph. D. dissertation
12. A G Requicha, ''Geometric Modelling and Programmable Automation,'' in *Proceedings of the IFIP TC5 International Conference on CAD/CAM. Technology Transfer to Latin America: Mexico City, August 22-26 '88.*, ed. G. Leon Lastra,IIE and Conacyt, México (1988).

13. Z Ruttkay and P ten Hagen, ''Intelligent User Interfaces for Intelligent CAD,'' in *Pre-prints of the Second IFIP WG 5.2 Workshop on Intelligent CAD.*, ed. H. Yoshikawa and T. Holden,The University of Tokio, Tokio (1988).

14. G Schmitt, ''IBDE, VIKA, ARCHPLAN: Architectures for Design Knowledge Representation, Acquisition, and Application,'' in *Pre-prints of the Second IFIP WG 5.2 Workshop on Intelligent CAD.*, ed. H. Yoshikawa and T. Holden,The University of Tokio, Tokio (1988).

15. G Sunde, ''A CAD System with Declarative Specification of Shape,'' pp. 20 - 39 in *Intelligent CAD Systems I*, ed. P.J.W ten Hagen and T. Tomiyama,Springer-Verlag, Berlin (1987).

16. H Suzuki, H Ando, and F Kimura, ''Synthesizing Product's Shapes with Geometric Design and Reasoning,'' in *Pre-prints of the Second IFIP WG 5.2 Workshop on Intelligent CAD.*, ed. H. Yoshikawa and T. Holden,The University of Tokio, Tokio (1988).

17. P J Szalapaj and A Bijl, ''Knowing where to Draw the line,'' pp. 147 - 165 in *Proceedings of the IFIP WG 5.2 Working Conference on Knowledge Engineering in Computer-Aided Design, Budapest, Hungry, 17-19 September 1984*, ed. John S. Gero,North-Holland, Amsterdam (1985).

18. B Tilove, ''Set Membership Classification: A Unified Approach to the Geometric Intersection Problem,'' *IEEE Transactions on Computers* **C-29**(10)(1980).

19. K Weiler, ''Polygon Comparation using Graph Representations,'' *ACM Siggraph'77 Conference Proceedings, Computer Graphics* **11**(2)(1977).

20. M Yukishita, Y Nakamura, and R Nomura, ''An Architecture of Intelligent CAD Having Natural Language Interface,'' in *Pre-prints of the Second IFIP WG 5.2 Workshop on Intelligent CAD.*, ed. H. Yoshikawa and T. Holden,The University of Tokio, Tokio (1988).

Tetsuo Tomiyama, Deyi Xue, and Yoshiki Ishida

An Experience of Developing a Design Knowledge Representation Language

An Experience of Developing a Design Knowledge Representation Language

Tetsuo Tomiyama[*], *Deyi Xue*[†], *and Yoshiki Ishida*[‡]

Department of Precision Machinery Engineering, Faculty of Engineering
The University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan
Tel: 03-812-2111 ext. 6454, Fax: 03-812-8849, Telex: 272 2111 FEUT J
E-Mail: b39711@tansei.cc.u-tokyo.ac.jp

Abstract: *This paper describes our experience of developing a design knowledge representation language named IDDL (Integrated Data Description Language) on the object oriented language Smalltalk-80. IDDL is the kernel language of the IIICAD (Intelligent Integrated Interactive CAD) system and is developed to represent design knowledge about both design processes and design objects. First, the IDDL language specifications are reviewed. Following a brief note about the implementation, we then show a couple of examples illustrating how IDDL represents design knowledge.*

Keywords: CAD, knowledge engineering, design knowledge, design process, design object, modal logic, default reasoning, ATMS, user interface.

1. INTRODUCTION

Computer aided design (CAD) systems are now playing a crucial role in many engineering fields. Particularly in mechanical engineering, excellent tools to deal with geometrical information have been developed and designers are liberated from laborious drafting so as to concentrate on more creative work. Recent advances of artificial intelligence and knowledge engineering technology have implied a new direction for CAD, i.e., so-called intelligent CAD systems (ICAD). There are already enough number of systems and it might not be too early to say that this trend is a definitely promising approach to convert the designer's power to more essential and creative phases of design, because this is perhaps the only way to achieve high

* Associate Professor, † Graduate Student, ‡ Research Associate

This paper was presented at the 3rd Eurographics Workshop on Intelligent CAD Systems "Practical Experience and Evaluation," 3-7 April 1989, The Netherlands.

productivity and quality of design at the same time.

Two different approaches can be observed in developing ICAD [ten Hagen and Tomiyama 1987]. One is top-down and somehow abstract, and it insists that the system should be established on a sound, tough theoretical basis to describe design knowledge; otherwise, development of ICAD will be beaten by the complexity of design knowledge. In this approach, one might begin with analyzing the nature of design to arrive at a general platform or framework on which ICAD elements should be built. The other is bottom-up and rather concrete in that intelligent design systems for a narrow problem domain are first built and then connected to each other to form the entire system. This approach puts more emphasis on analyses about how design is performed using design knowledge and it often leads to development of so-called expert design systems or knowledge based design systems.

The IIICAD (Intelligent Integrated Interactive CAD) project was proposed at the Centre for Mathematics and Computer Science (CWI) in Amsterdam [Tomiyama and ten Hagen 1987], considering current CAD systems have the following three problems:

- Not intelligent.
- Not integrated.
- Poor in man-machine communication.

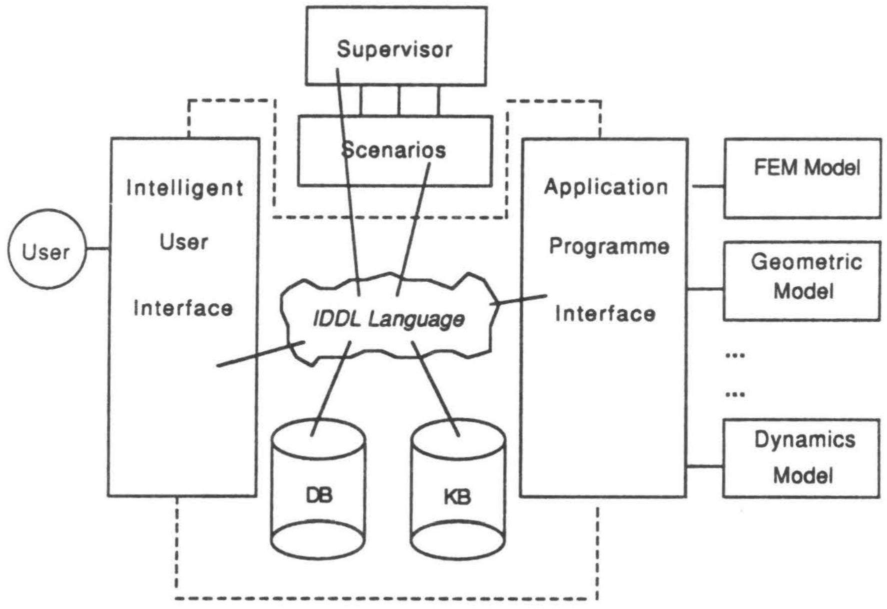


Fig. 1. The configuration of the IIICAD system

The core of the IIICAD project is the development of IDDL (Integrated Data Description Language), a design knowledge representation language that can describe both design processes and design objects [Veth 1987]. Figure 1 shows the configuration of the IIICAD system in which the *supervisor* controls design processes based on *scenarios* written in IDDL, and all the subsystems communicate with each other in IDDL. In this regard, IDDL is a framework to represent design knowledge and the IIICAD project takes, therefore, a top-down, framework approach.

The present paper is a report on our experiences about IDDL in an extended and modified version of the IIICAD project which is currently conducted at the University of Tokyo. In Chapter 2, we briefly review the IDDL specifications which are elaborated from a working draft in [Veth 1987]. Chapter 3 describes techniques used for the implementation of IDDL. Most of useful ideas for the implementation of IDDL were taken from the work by Megens at CWI [Megens 1987] which was, however, only partially finished. Chapter 4 is our contribution: We will describe various types of design processes in IDDL with examples and evaluate the original ideas of IDDL from a viewpoint of representation of design knowledge. Chapter 5 concludes the paper.

2. IDDL LANGUAGE SPECIFICATIONS

2.1. Overview

In the IIICAD project it is emphasized that such an advanced system should be constructed on a sound, tough theoretical basis, and one of the project goals is to establish a *theory of CAD* consisting of theories of design processes, design objects, and knowledge [Veth 1987]. There are commercially available so-called ICAD systems, but most of these systems are much concerned about dealing with constrained information. These systems can deal with prototypical design objects; the structures of these objects are predefined in one way or another and their attributes construct a network of such relationships as equality, equivalence, derivation, and implication. This can be justified in a situation where the purpose of design is to modify details of once designed objects and the majority of design activities is to solve interrelated, complicated constraints, such as geometrical constraints in an intelligent manner, which is the case in mechanical engineering design. However, this point of view lacks one of the most crucial aspects of design; i.e., design processes. Without explicit knowledge about design processes, for example, it might be extremely difficult to handle the trial-and-error nature of design, because such a process cannot be very effectively performed by simple backtracking.

There are three major concepts that are relevant to design and represented in IDDL, viz. design objects (entities), their attributes, and relationships among entities. There is an assumption that a design process is a stepwise refinement process and is decomposed into subprocesses. In other words, a design process is composed of a set of subprocesses and how these subprocesses are combined determines the process.

IDDL takes a logical standpoint; i.e., design knowledge is represented by a set of facts and rules that are denoted by logical formulae and the design solution is obtained as results of logical reasoning about these formulae. Thus, a design process must be (and can be) formalized logically [Veerkamp 1989; Takeda, Tomiyama, and Yoshikawa 1989]. In our formalization, an entity is denoted by a term (both constant and variable) and a relationship between two entities is represented by a binary first-order predicate. Let us call this way of formalization an *extensional* description [Tomiyama and ten Hagen 1987a]. Any design knowledge about these predicates are formalized as simple rules to manipulate them. However, there can be cases where one may want to express and handle attributes of entities. In principle, such an attribute must be expressed by the equality predicate over the attribute and the value in an extensional description method. In order to explicitly state the fact that the attribute belongs to the entity, a function is used. For instance, suppose the entity ''X'' is a box and has the attribute ''depth'' which is equal to 40 cm. This is represented in IDDL as follows (*intensional description*).

```
box(X),
=(depth[X], 40)
```

This type of description methods is convenient for computational and representational purposes, but it has a drawback as well: It assumes *structure*; i.e., that any box (whatever the box is) has depth, rather than height, for example. Otherwise, if used carelessly, any statement that assumes a box has the attribute *depth* becomes meaningless.

The following requirements were taken into consideration during the design of IDDL [Veth 1987].

(a) IDDL should be able to describe design knowledge both about design objects and design processes.

(b) IDDL should be able to describe design objects, their relationships, and attributes of design objects.

(c) IDDL should be able to describe design processes, which are often performed in a trial-and-error manner and starts with some assumptions.

These issues resulted into the following properties of IDDL:

(1) IDDL uses *scenarios* to describe design processes. A scenario is a meaningful chunk of design procedures, such as design of a particular mechanism and decision making based on a certain technique.

(2) *Objects* denote an entity and *facts* are logical formulae that represent relationship among entities. *Worlds* represent a set of objects now being considered and, in fact, partitions in the database.

(3) IDDL has a multiple world mechanism based on *modal logic*.

(4) There is a distinction between real facts and assumed facts. This feature allows *default reasoning* based on ATMS (Assumption-based Truth Maintenance System) [de Kleer 1986].

Figure 2 illustrates how an IDDL *program* is executed. First, we describe design processes in terms of scenarios and functions. The results of its execution are stored in the *active world* as facts, objects, and attributes of objects. The complete IDDL syntax is given in Appendix.

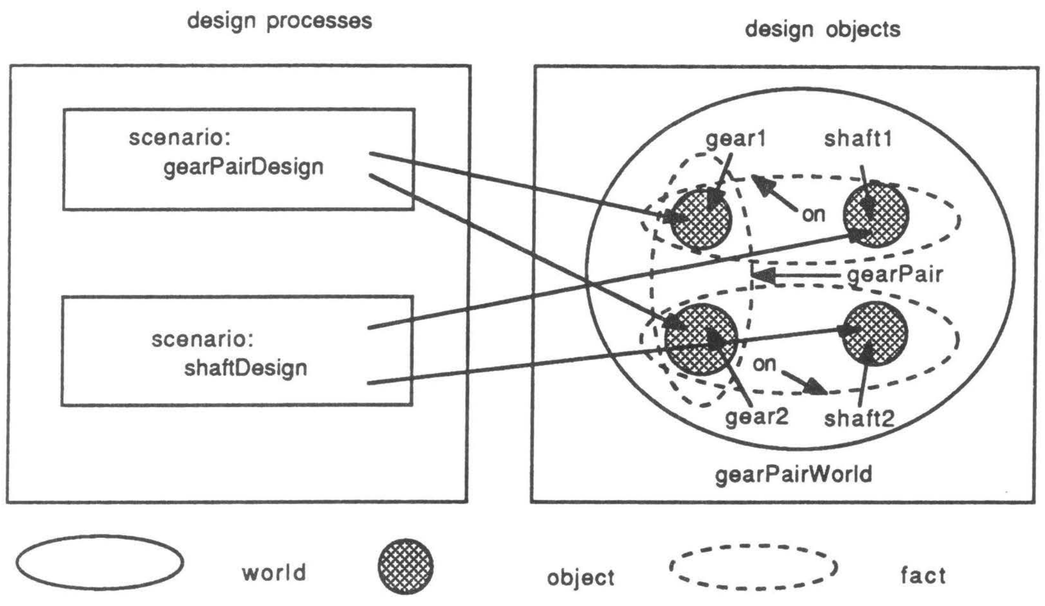


Fig. 2. Execution of an IDDL program

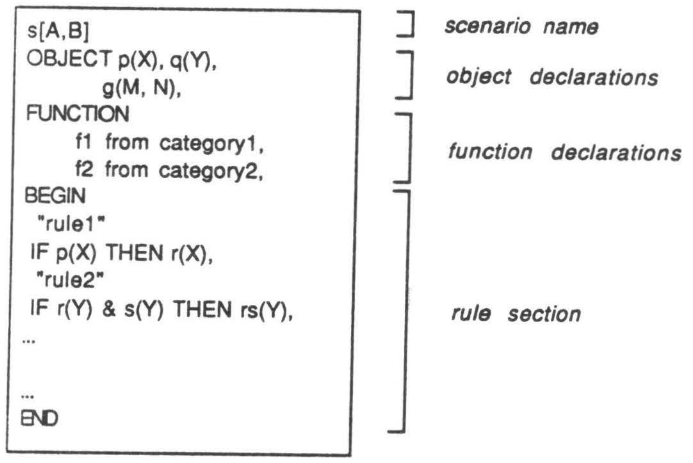
2.2. Scenarios — Descriptions of Design Processes

The construction of a scenario is shown in Fig. 3. By the object declarations and function declarations, objects or functions used in this scenario are declared. The main body of a scenario is the rule section. A rule has the well-known

IF <conditions> THEN <operations>

structure in which, if the condition part is satisfied, then the operation part should be asserted or executed.

In a scenario, we can call other scenarios by using the built-in predicate *select*. The execution of scenarios terminates in one of the following three conditions; the *success* condition, the *fail* condition, or the *no-more-rule* condition which is treated as a success. If the execution of a scenario is a success, all facts derived from this scenario should be saved in the database; if failed, all the facts derived from this scenario should be removed from the database.



```
s[A,B]
OBJECT p(X), q(Y),
        g(M, N),
FUNCTION
     f1 from category1,
     f2 from category2,
BEGIN
 "rule1"
 IF p(X) THEN r(X),
 "rule2"
 IF r(Y) & s(Y) THEN rs(Y),
...

...
END
```

Fig. 3. A scenario

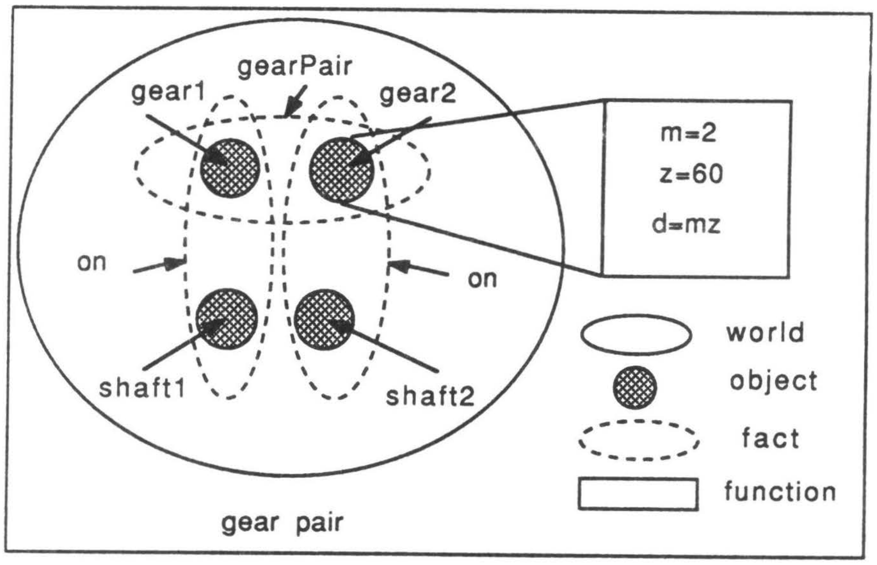


Fig. 4. Design objects

2.3. Objects and Worlds — Descriptions of Design Objects

IDDL has the concepts of objects, attributes of objects and facts, and worlds to describe design objects (Fig. 4). A world consists of objects and facts which describe relationships among these objects and it is in fact a partition of the database. to preserve the results of design. An object of IDDL has attributes and we can define operations over these attributes (see the next section about functions). An object beginning with an upper letter denotes a variable, whereas one beginning with a lower letter denotes a constant.

A world can be created by the built-in predicate *enclose*:

enclose(world1, object1, object2, object3)

This creats a new world called "world1" which contains "object1," "object2," and "object3." There must be always one world active in the database called the *active world* in which execution of scenarios takes place and the results are preserved. The current active world can be changed by using the built-in predicate *enter* or *exit* (Fig. 5).

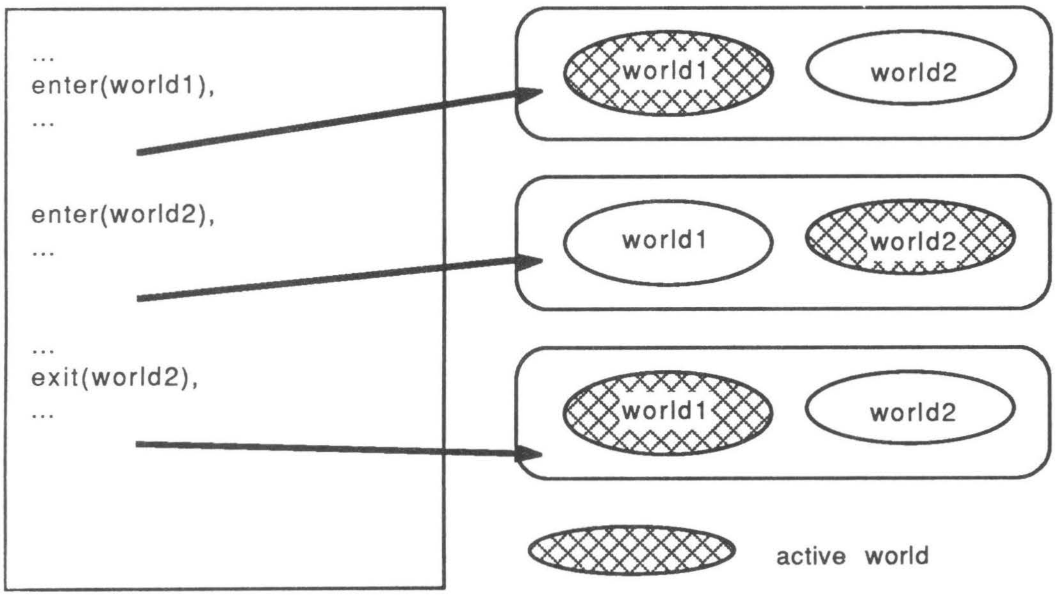


Fig. 5. Changing worlds

An object can be created by the built-in predicate *makeObject*:

makeObject(Gear, gear1)

where "Gear" is the type of the generated object "gear1". In principle, a type implies that objects belonging to this type share the same data structure, i.e., the same set of functions (Fig. 6). This means that IDDL has a class mechanism much the same as other object oriented languages. However, the binding of objects belonging to different worlds is done imperatively by object declarations in the scenario. Therefore, an object belonging to a particular world can be bound to another object in a different world occasionally. This *delegation* mechanism allows much more useful and flexible specialization than the widely-used inheritance mechanism [Tomiyama 1989].

2.4. Functions — Operations to Design Objects

Functions are used to define attributes of objects and there are three kinds of functions:

(1) Functions defined within a particular type of objects.

(2) Functions defined commonly over a set of different types. For instance:

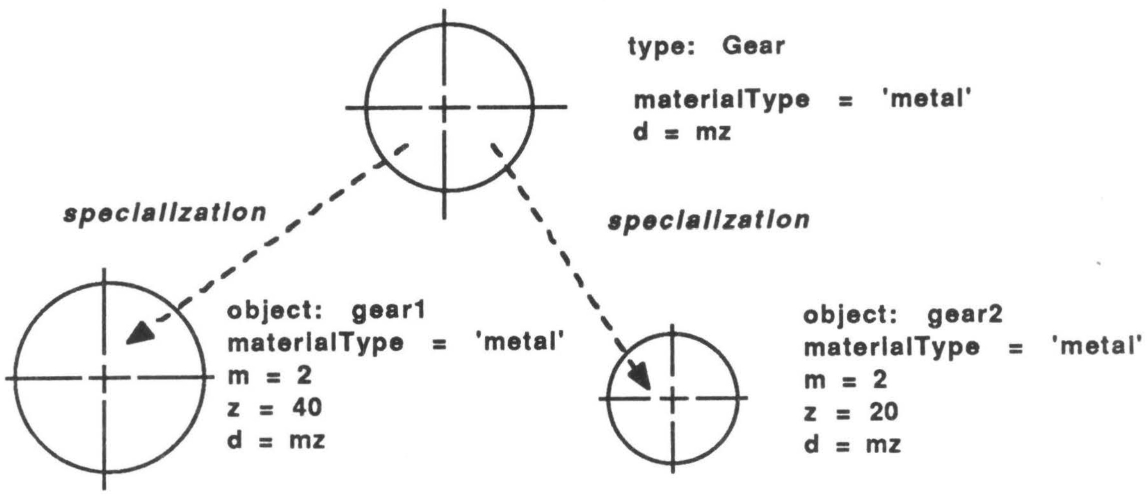


Fig. 6. The structure of objects and the type of objects

```
distance[A, B]
X := x[A] - x[B],
Y := y[A] - y[B],
D := sqrt[sqr[X] + sqr[Y]],
↑D
```

where ↑ denotes "return the following value."

(3) Functions defined to operate attributes of objects. For example:

```
IF =(depth[gear1], 40) THEN ···
```

2.5. Modal Logic

Suppose a design problem such as "If one of the conditions is satisfied, the entire design should be considered as a success." IDDL solves it by introducing modal logic with necessity and possibility modality.

IDDL has two modal operators, viz. "#N" for necessity and "#P" for possibility. For example, consider "#Np(a)." If all of the *accessible* worlds from the current active world has the fact "p(a)," this matching will return true; otherwise, false. In the same way, the matching of "#Pp(a)" to the database returns true, if some of the accessible world has the fact "p(a)"; otherwise, false.

In order to determine which world is *accessible* (i.e., *can be seen*), we need to choose the most appropriate kind of modal logic systems from T, B, S4 or S5 system according to the accessibility properties including reflection, symmetry, and transition. The definitions of accessibility properties and modal logic systems are shown in Table 1 and 2. For example, the modal logic system B has the property of symmetry. In such a case, if it is defined such that the world w1 is accessible from the world w2, the system will automatically conclude that w2 is accessible from w1. These properties are defined by the built-in predicates *multiWorldsSystem* and *canSeeWorlds*.

Table 1. The accessibility properties

reflection	One world can see itself.
symmetry	If the world A can see the world B, the world B can see the world A, too.
transition	If the world A can see the world B and the world B can see the world C, then the world A can see the world C, too.

Table 2. Modal logic systems

system name	properties
T	reflection
B	reflection, symmetry
S4	reflection, transition
S5	reflection, symmetry, transition

2.6. ATMS

IDDL employs the default operator "#D" to allow database management based on ATMS [de Kleer 1986]. IDDL distinguishes real facts and assumed facts, and an assumed fact has descriptions about from which facts it was derived. When a real fact is added to the database, assumed facts are checked for the integrity of the database; assumed facts may be changed to real facts or removed from the database.

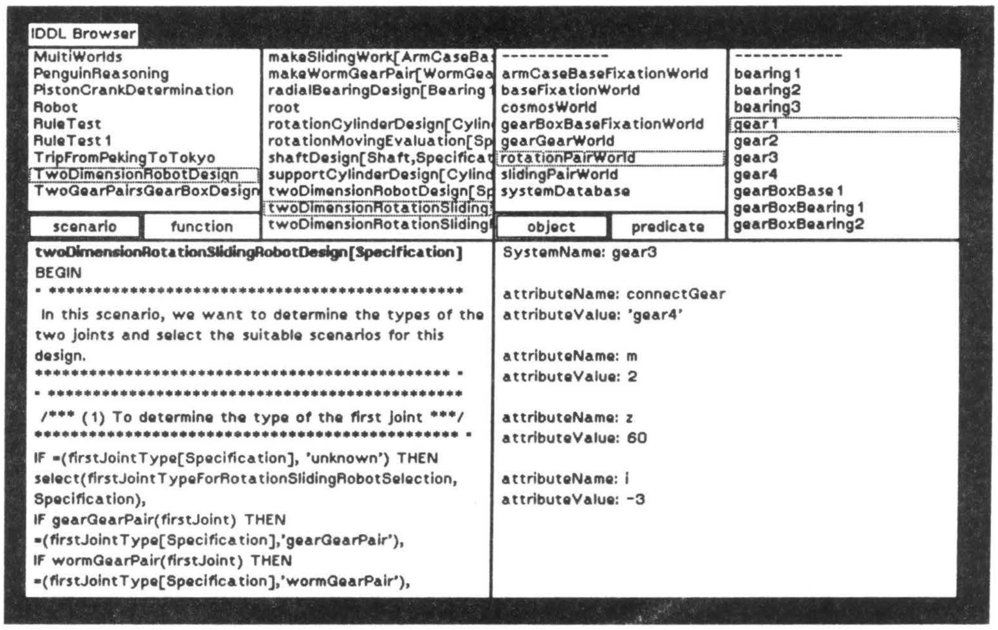
3. IMPLEMENTATION OF IDDL

The IDDL language is implemented in Smalltalk-80 [Goldberg and Robson 1983]. Smalltalk-80 was employed as the implementation language for the following reasons:

(1) IDDL itself is a combination of the logic programming and the object oriented programming paradigms [Veth 1987; Tomiyama 1989]. Most of the properties of IDDL, such as delegation, can be easily implemented using the features of Smalltalk-80.

(2) Smalltalk-80 provides facilities to easily build an excellent user interface environment.

(3) Smalltalk-80 itself is a good tool for rapid prototyping that allows e.g. incremental programming.

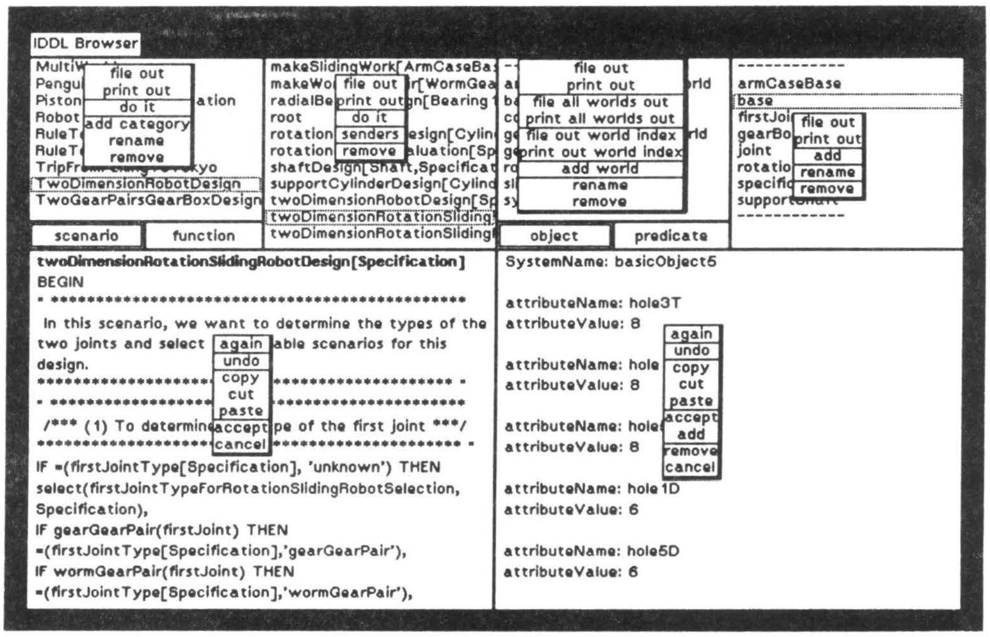


```
IDDL Browser

MultiWorlds                      makeSlidingWork[ArmCaseBa
PenguinReasoning                 makeWormGearPair[WormGea
PistonCrankDetermination         radialBearingDesign[Bearing
Robot                            root
RuleTest                         rotationCylinderDesign[Cylin
RuleTest1                        rotationMovingEvaluation[Sp
TripFromPekingToTokyo            shaftDesign[Shaft,Specificat
TwoDimensionRobotDesign          supportCylinderDesign[Cylind
TwoGearPairsGearBoxDesign        twoDimensionRobotDesign[Sp
                                 twoDimensionRotationSliding
scenario      function           twoDimensionRotationSliding

------------                     ------------
armCaseBaseFixationWorld         bearing1
baseFixationWorld                bearing2
cosmosWorld                      bearing3
gearBoxBaseFixationWorld         gear1
gearGearWorld                    gear2
rotationPairWorld                gear3
slidingPairWorld                 gear4
systemDatabase                   gearBoxBase1
                                 gearBoxBearing1
object      predicate            gearBoxBearing2
```

```
twoDimensionRotationSlidingRobotDesign[Specification]
BEGIN
* ***********************************************
 In this scenario, we want to determine the types of the
two joints and select the suitable scenarios for this
design.
*********************************************** *
* ***********************************************
 /*** (1) To determine the type of the first joint ***/
************************************************ *
IF =(firstJointType[Specification], 'unknown') THEN
select(firstJointTypeForRotationSlidingRobotSelection,
Specification),
IF gearGearPair(firstJoint) THEN
=(firstJointType[Specification],'gearGearPair'),
IF wormGearPair(firstJoint) THEN
=(firstJointType[Specification],'wormGearPair'),
```

```
SystemName: gear3

attributeName: connectGear
attributeValue: 'gear4'

attributeName: m
attributeValue: 2

attributeName: z
attributeValue: 60

attributeName: i
attributeValue: -3
```

Fig. 7. IDDL browser



```
IDDL Browser

MultiW                           makeSlidingWork[ArmCaseBa
Pengu                            makeWo              r[WormGea
Piston                ation      radialBe            gn[Bearing
Robot                            root
RuleT                            rotation            esign[Cylin
RuleT                            rotation            aluation[Sp
TripFr                 kyo       shaftDesign[Shaft,Specificat
TwoDimensionRobotDesign          supportCylinderDesign[Cylind
TwoGearPairsGearBoxDesign        twoDimensionRobotDesign[Sp
                                 twoDimensionRotationSliding
scenario      function           twoDimensionRotationSliding

--                               ------------
a                       rld      armCaseBase
b                                base
c                                firstJoi
g                       rld      gearBo
g                                joint
r                                rotatio
sl                               specific
sy                               suppor
                                 ------------
object      predicate
```

```
twoDimensionRotationSlidingRobotDesign[Specification]
BEGIN
* ***********************************************
 In this scenario, we want to determine the types of the
two joints and select       able scenarios for this
design.
****************        ******************** *
* ****************       ********************
 /*** (1) To determin      pe of the first joint ***/
******************       ********************* *
IF =(firstJointType[Specification], 'unknown') THEN
select(firstJointTypeForRotationSlidingRobotSelection,
Specification),
IF gearGearPair(firstJoint) THEN
=(firstJointType[Specification],'gearGearPair'),
IF wormGearPair(firstJoint) THEN
=(firstJointType[Specification],'wormGearPair'),
```

```
SystemName: basicObject5

attributeName: hole3T
attributeValue: 8

attributeName: hole
attributeValue: 8

attributeName: hole
attributeValue: 8

attributeName: hole1D
attributeValue: 6

attributeName: hole5D
attributeValue: 6
```

Fig. 8. Menus of the IDDL browser

The programming environment of IDDL is the *IDDL browser* shown in Fig. 7. Figure 8 shows available menus on the IDDL browser. In the left side of the browser, we can write and compile scenarios and functions and eventually execute scenarios. In

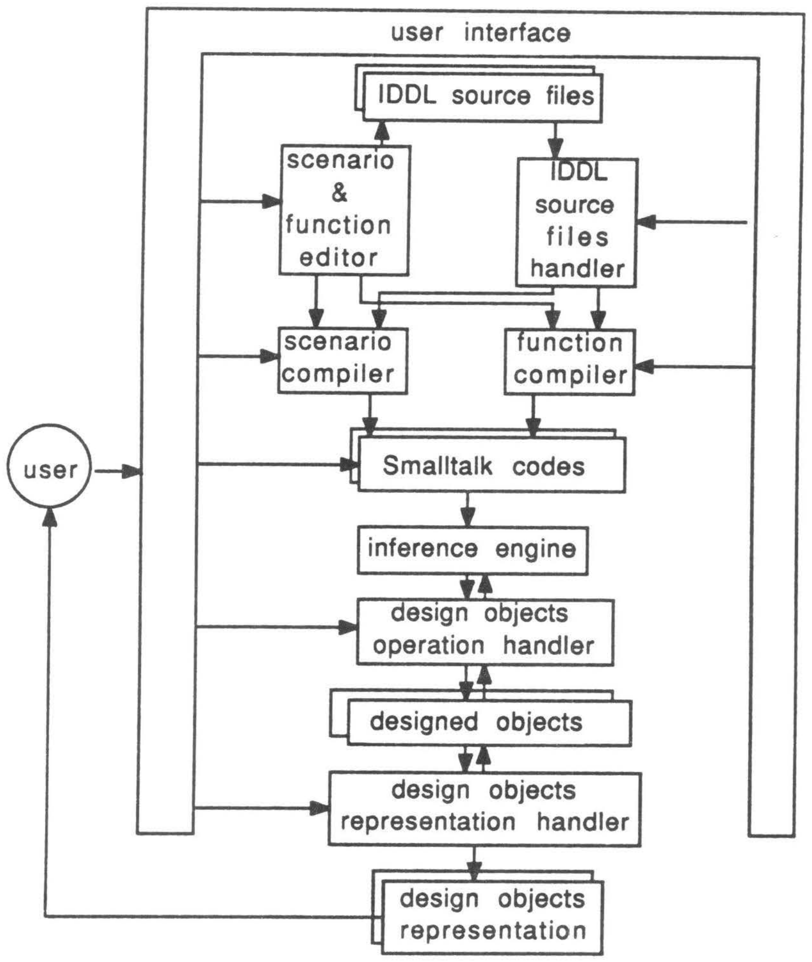

user interface
IDDL source files
scenario & function editor
IDDL source files handler
scenario compiler
function compiler
user
Smalltalk codes
inference engine
design objects operation handler
designed objects
design objects representation handler
design objects representation

Fig. 9. The architecture of the IDDL system

the right side of the browser, we can see the results of the execution described in terms of worlds, objects, attributes of objects and facts. There are two compilers, viz. scenario compiler and function compiler, to translate scenarios and functions into Smalltalk-80 codes, respectively. Therefore, an IDDL object is a Smalltalk-80 object. This idea was taken from the work by Megens [Megens 1987].

Figure 9 depicts the architecture of the IDDL system. The *scenario & function editor* is used for editing IDDL codes. The *scenario compiler* and *function compiler* are used to translate scenarios and functions into Smalltalk codes. The *inference engine* sends messages to the *design object operation handler* to operate objects and the database. The *design object representation handler* is used for presenting the status of the database.

4. REPRESENTING DESIGN KNOWLEDGE IN IDDL

The following sections are based on an analysis about design knowledge in an intelligent CAD environment [Tomiyama and ten Hagen 1987b].

4.1. Representing Design Processes in IDDL

4.1.1. Design Processes and Subprocesses

A design process can be often divided into several subprocesses. The result of the whole design process is obtained as a collection of solutions of the subprocesses. Since a scenario describes a chunk of design procedures, this nature of design can be represented by the scenario structure, i.e., a scenario calling other subscenarios. For example, we can divide the design of diving clothes into several subprocesses such as the design of the hat, the coat, the trousers, and the shoes (Fig. 10).

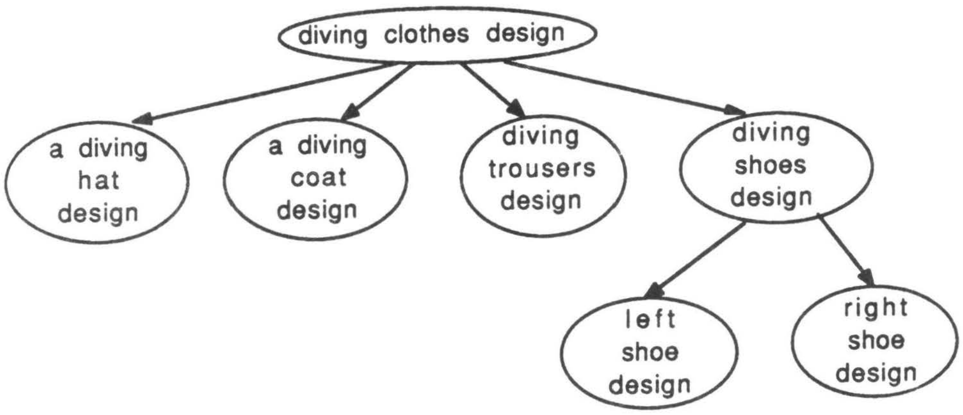


Fig. 10. Design subprocesses

4.1.2. Design Knowledge and Meta Design Knowledge

Design is not a simple problem solving process; rather, it includes also a process to determine the way to solve the problem, to find out which knowledge should be used, etc. This means that we need meta knowledge (or knowledge about knowledge) to control the design process. IDDL allows to write meta scenarios to select appropriate lower-level scenarios.

4.1.3. The Trial-and-Error Method

To solve a problem, we often use the trial-and-error method. By using of the multi-world mechanism of IDDL and the default reasoning feature, we can develop alternative solutions in different worlds until we arrive at satisfactory solutions.

4.1.4. Application of Modal Logic

IDDL allows the designer to develop alternative different worlds at the same time. We use the ''#N'' operator to describe conditions common to all the related worlds and the ''#P'' operator to describe conditions which hold at least in one of the worlds. In the example of Fig. 10, we can design components of the diving clothes separately in different worlds and the waterproof condition can be added by saying ''#Nwaterproof.''

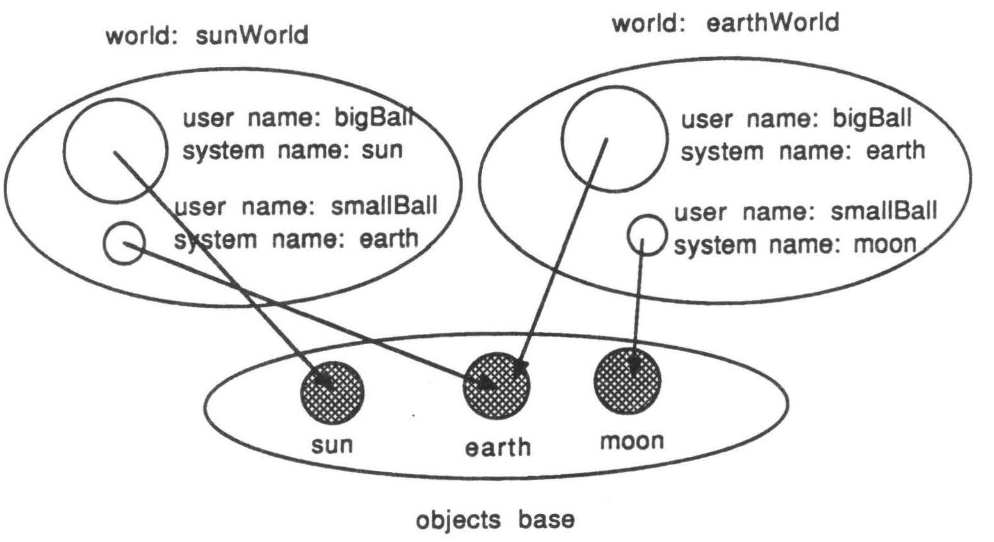
4.2. Representing Design Objects in IDDL

4.2.1. Design Objects from Multiple Viewpoints

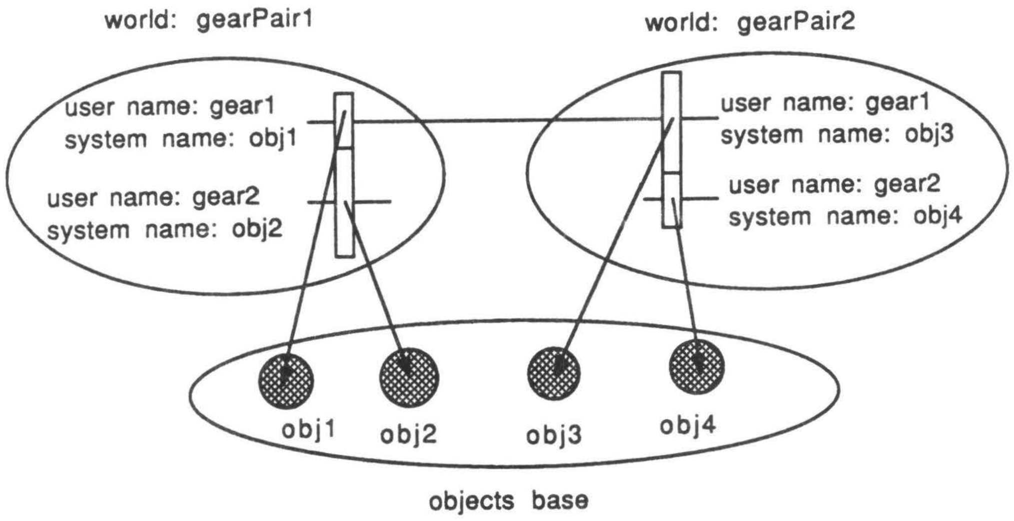
The multi-world mechanism allows to describe the same object from different points of view. We may have the same object in different worlds with different user names (Fig. 11 (a)). The properties of such an object can be different. This allows viewing an identical object from multiple viewpoints, which is essentially useful for mechanical engineering design. On the other hand, it is also possible to have different objects under the same user name in different worlds (Fig. 11 (b)).

4.2.2. Database Integrity

The IDDL system incorporates ATMS to keep the database integrity. As the design proceeds, it sometimes happens that initial assumptions must be changed. In such a case, the facts derived from these assumptions should be also changed. When a fact is added into the database, all the related facts will be automatically checked in order to maintain the consistency of the database.



(a) Same objects with different user names in different worlds



(b) Different objects with the same user names in different worlds

Fig. 11. User names and system names of objects

4.3. Intelligent Design Interface

IDDL is merely a language for system designers, so to speak, and not for domain design experts. This requests to provide an intelligent design interface on top of IDDL in the IIICAD architecture. The primary step we take is to make *IDDL programming* interactive. Figure 12 shows a system called *design browser* for this purpose. This browser has the following abilities:

(1) Drawing figures and inputting data, such as attributes, types, and names of the objects identified by figures.

(2) Selecting adequate scenarios to be executed.

(3) Displaying output, such as figures and results of IDDL programs.

(4) Storing information about the designed object, such as design models, the initial input, and the order of execution of the scenarios.

A future, advanced step might be to develop a graphical interface which extracts the designer's concepts from the designers operations for graphical objects and which translates them into both design process knowledge as IDDL scenarios and design object knowledge as IDDL objects.

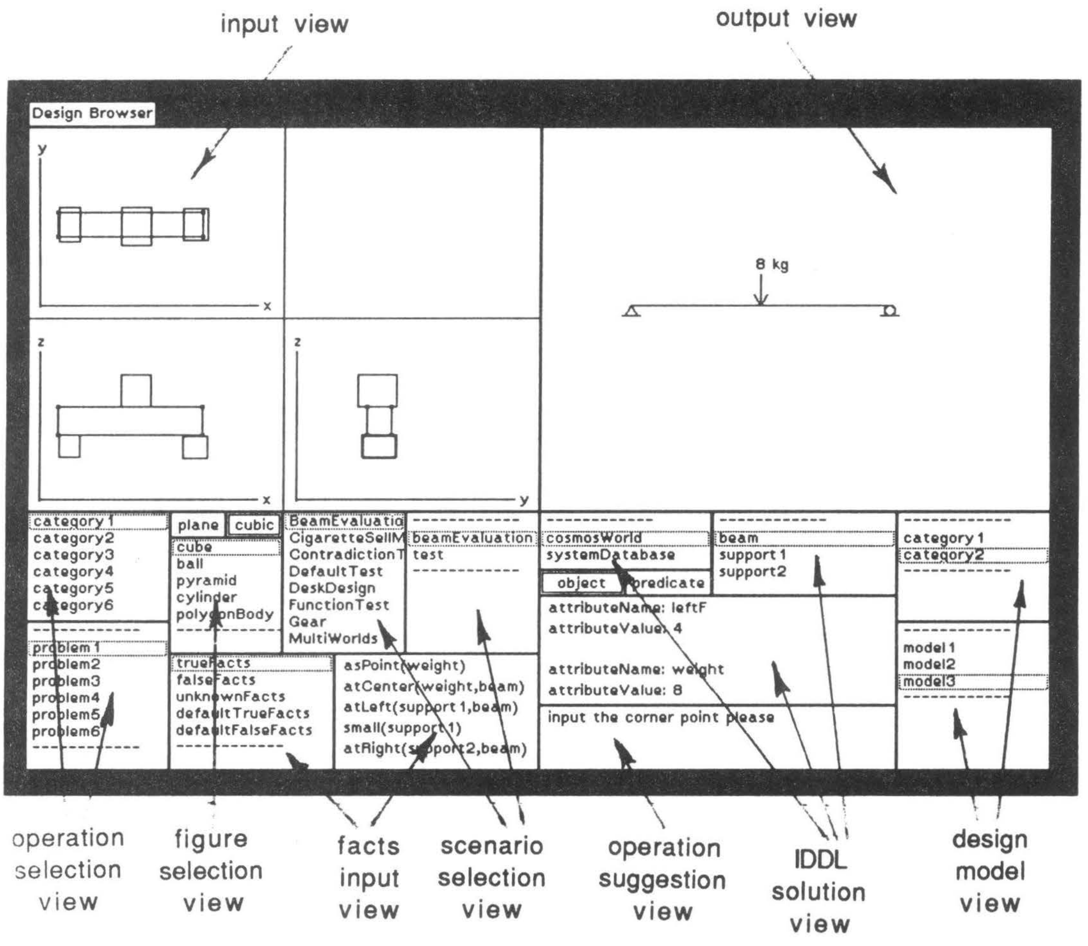


Fig. 12. Design browser

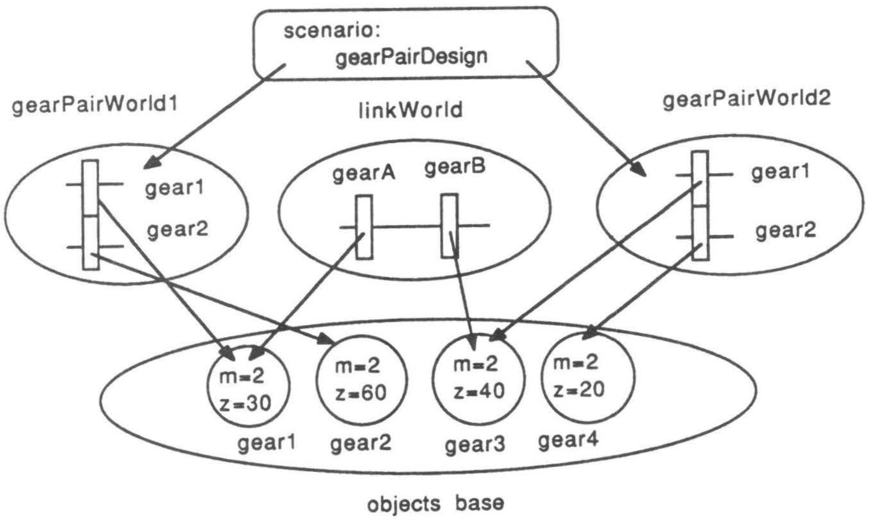
4.4. Examples

4.4.1. Representing Design Objects by Using Worlds and Objects

A simple example of mechanical design, the design of a gear box, is illustrated here. We have a scenario for a gear box and a subscenario for a gear pair. Because the same scenario was executed twice, two objects with the same user name are first generated. (Of course, since the system distinguishes user names and system names, this is not a problem.) Second, ''gear1'' is sent from ''gearPairWorld1'' to ''linkWorld'' with the user name ''gearA'' using the built-in predicate *sendObject*. In the same way, we create ''gearB'' and fix ''gearA'' and ''gearB'' on the same shaft. Third, we determine the parameters of the four gears (i.e. attributes) and complete this design (Fig. 13 (a)). The designed gear box has not only attributes but also the inner relationships that represent its structure.

4.4.2. Representing designed objects in different models

Figure 14 depicts an example for evaluations of a beam. First, as shown in Fig. 14 (b) to (d), the boundary conditions are determined. The rules used in this example are shown in Fig. 14 (e) and the results are obtained as shown in Fig. 14 (f) and Fig. 14 (g). This example shows most clearly how a design process on the design browser proceeds. The designer is always allowed to examine the database to check the results. If something is wrong, she/he can change interactively whatever needed including IDDL codes. However, this might not be the best, because the current version of IDDL does not really provide a user-friendly interface.



(a) Determining parameters and relationships

```
example3 scenario:
BEGIN
select(twoGearPairsGearBoxDesign),
enter(gearPairWorld2),
=(n[gear1],40),
proceedFunction(calculateGearBox[gear1]),
exit(gearPairWorld2),
END
```

```
twoGearPairsGearBoxDesign scenario:
BEGIN
enclose(linkWorld),
enclose(gearPairWorld1),
enter(gearPairWorld1),
select(gearPairDesign),
=(z[gear1],30) & =(z[gear2], 60),
=(m[gear1], 2) & =(m[gear2], 2),
sendObject(gear1,linkWorld,gearA),
exit(gearPairWorld1),
...
...
enter(linkWorld),
proceedFunction(linkGear[gearA, gearB]),
exit(linkWorld),
END
```

```
gearPairDesign scenario:
BEGIN
makeObject(Gear, gear1),
makeObject(Gear, gear2),
proceedFunction(gearPair[gear1, gear2]),
END
```

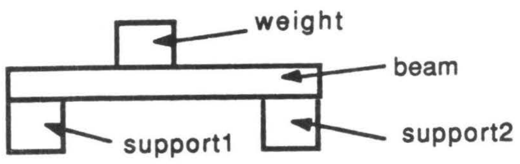
(b) IDDL program

```
... ...
world:
  'gearPairWorld2'

user name: 'gear1'
system name: 'gear3'

user name: 'gear2'
system name: 'gear4'

world:
  'linkWorld'

user name: 'gearA'
system name: 'gear1'

user name: 'gearB'
system name: 'gear3'
```

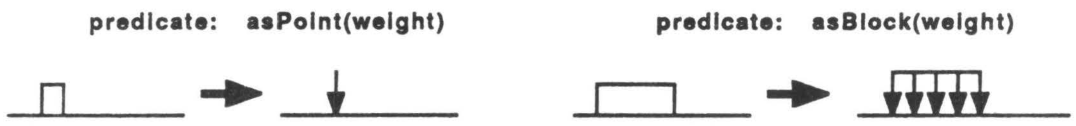
```
system object name:
  'gear1'

attribute name: m
attribute value: 2

attribute name: linkGear
attribute value: gear3

attribute name: n
attribute value: 40

... ...
```

(c) Result of the execution

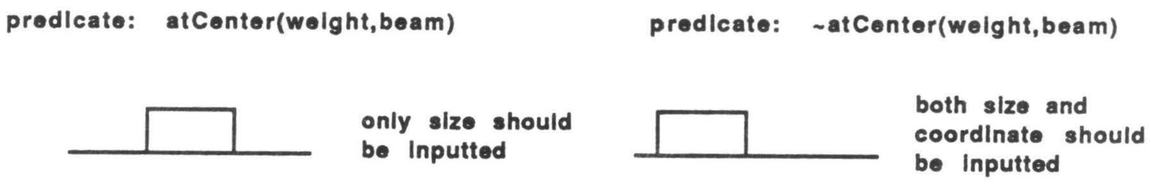
Fig. 13. A gear box design



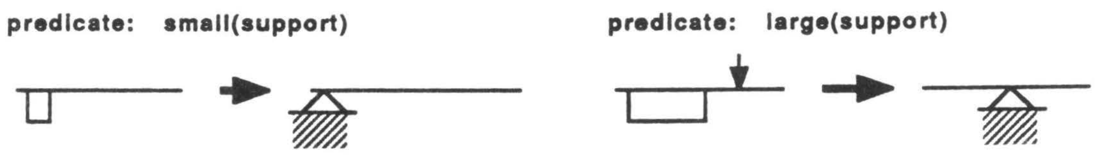
(a) The beam structure



(b) Determination of the type of load


predicate: atCenter(weight,beam)
only size should be inputted
predicate: ~atCenter(weight,beam)
both size and coordinate should be inputted

(c) Determination of the location of load

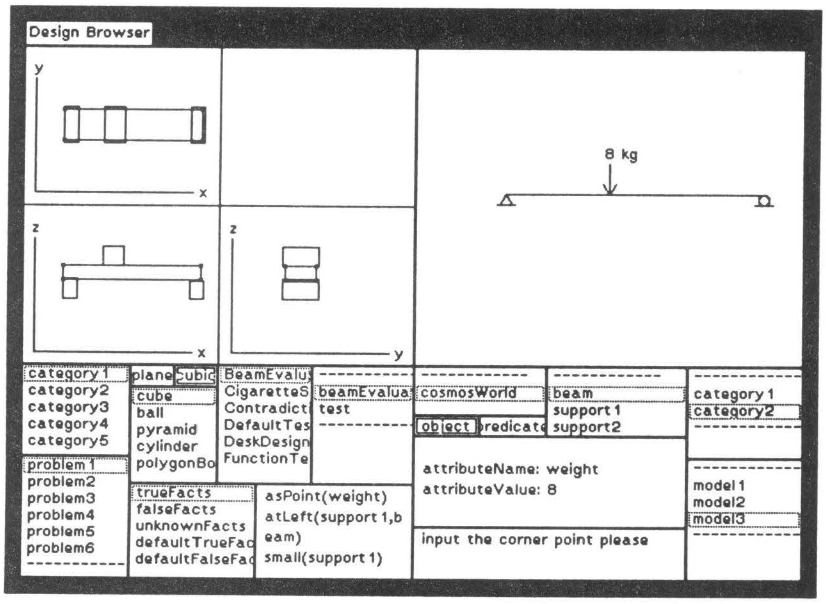


(d) Determination of the support condition

Fig. 14. Evaluations of a beam (cont.)

```
beamEvaluation scenario:
BEGIN
IF asPoint(weight) & atCenter(weight, beam) THEN proceedFunction(carryWeight1[beam, weight]),
IF asPoint(weight) & ~atCenter(weight, beam) THEN proceedFunction(carryWeight2[beam, weight]),
IF asBlock(weight) & atCenter(weight, beam) THEN proceedFunction(carryWeight3[beam, weight]),
IF asBlock(weight) & ~atCenter(weight, beam) THEN proceedFunction(carryWeight4[beam, weight]),
IF atLeft(support1, beam) & small(support1) THEN proceedFunction(smallLeftSupport[beam, support1]),
IF atLeft(support1, beam) & large(support1) THEN proceedFunction(largeLeftSupport[beam, support1]),
IF atRight(support2, beam) & small(support2) THEN proceedFunction(smallRightSupport[beam, support2]),
IF atRight(support2, beam) & large(support2) THEN proceedFunction(largeRightSupport[beam, support2]),
...
```

(e) Rules



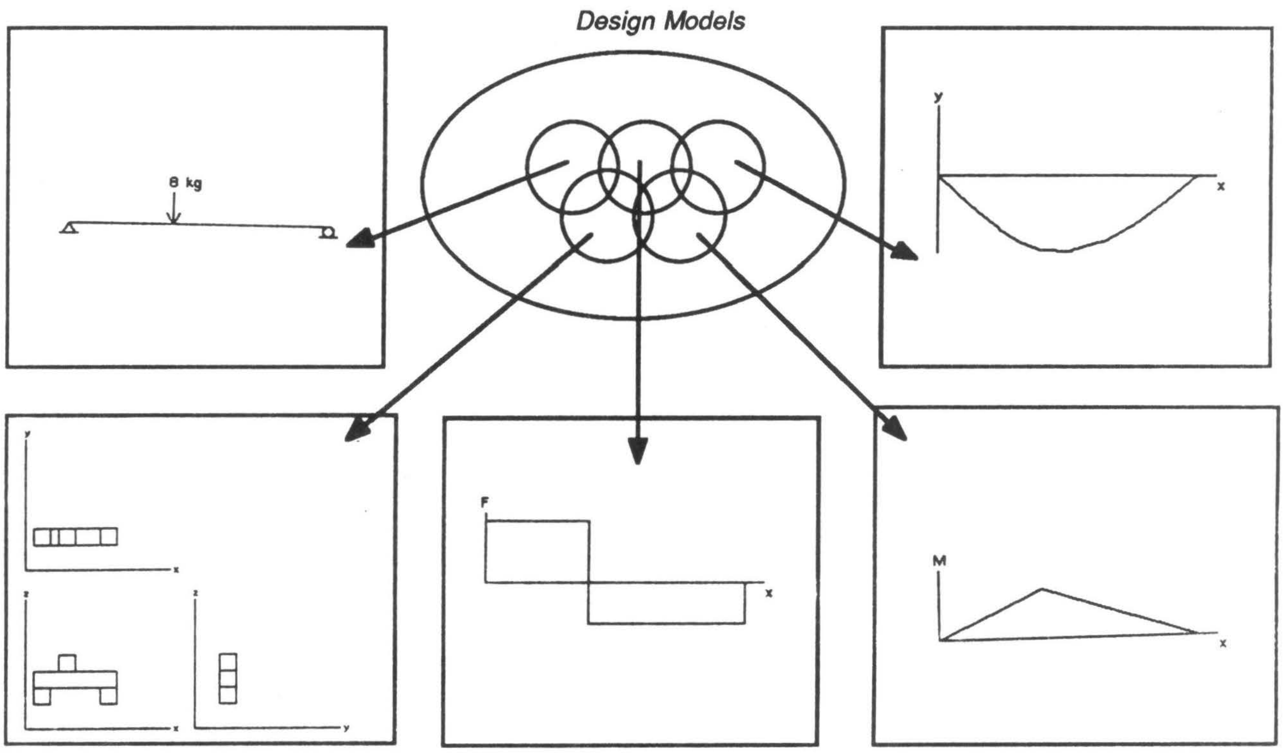
```
asPoint(weight)
atLeft(support1,b
eam)
small(support1)
```

```
attributeName: weight
attributeValue: 8
```

```
input the corner point please
```

(f) A solution

Fig. 14. Evaluations of a beam (cont.)



(g) Other solutions

Fig. 14. Evaluations of a beam

4.4.3. Data integrity inspection

Suppose we have the following rules:

[rule1]: IF T THEN #D p1(a) & #D q(a),
[rule2]: IF p1(X) THEN p2(X),
[rule3]: IF p2(X) THEN p3(X),
[rule4]: IF p3(X) & q(X) THEN r(X),
[rule5]: IF T THEN p2(a),
[rule6]: IF T THEN ~q(a)

where T is a logical constant which is always true. By executing rule1 to rule4, we get five default facts in the database (Fig. 15 and Fig. 16 (a)). When rule5 is executed, the real fact p2(a) is added to the database. Because the p2(a), p3(a), and r(a) are related with p2(a), these three default facts are checked and p2(a) and p3(a) will be changed into real facts (Fig. 16 (b)). In the same way, when rule6 is executed, q(a) and r(a) will be removed from the database (Fig. 16 (c)).

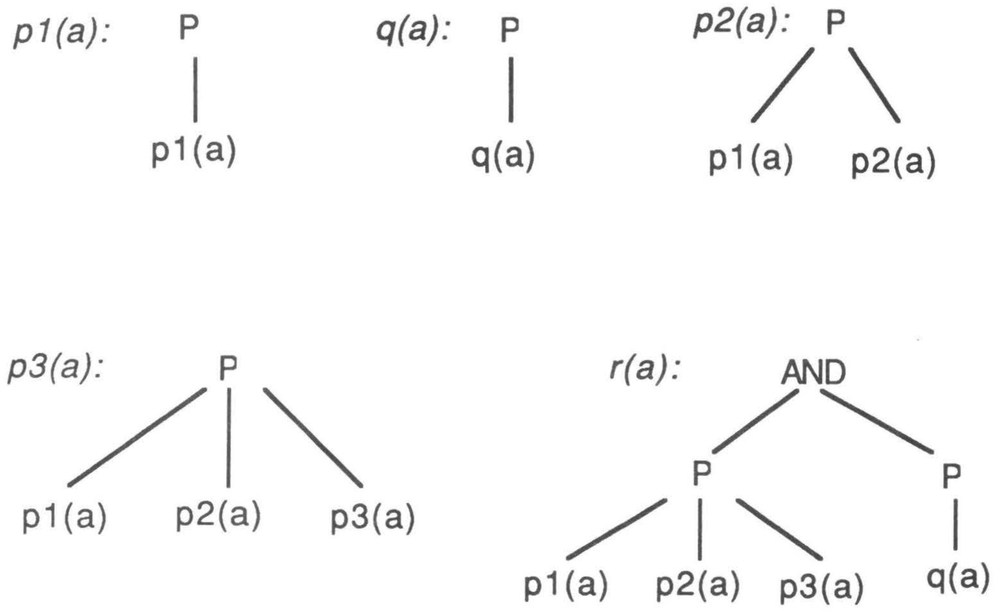
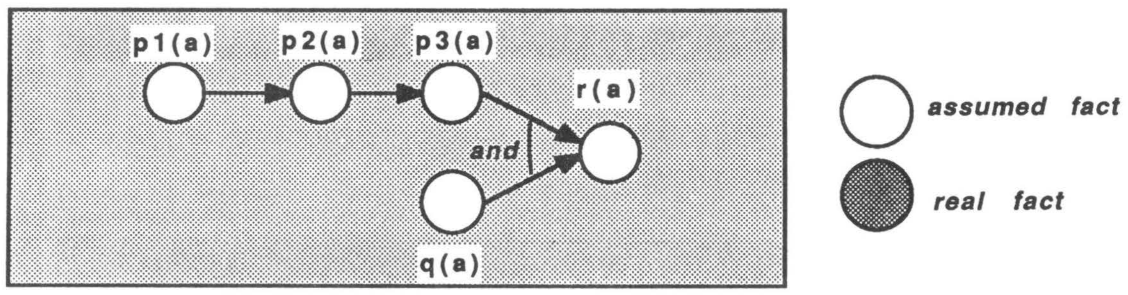
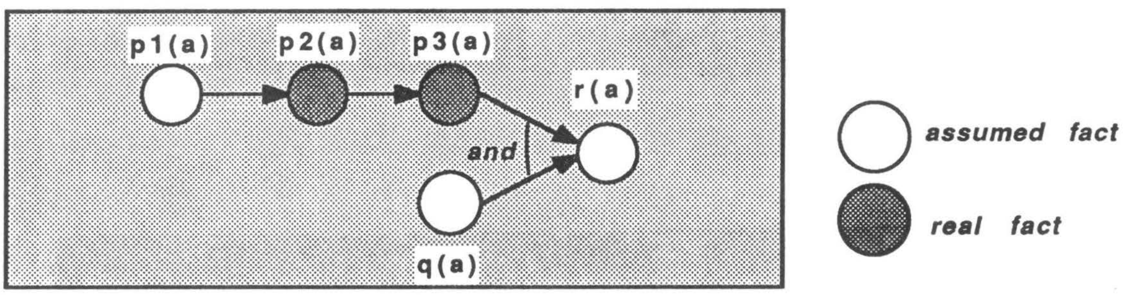


Fig. 15. Descriptions of facts

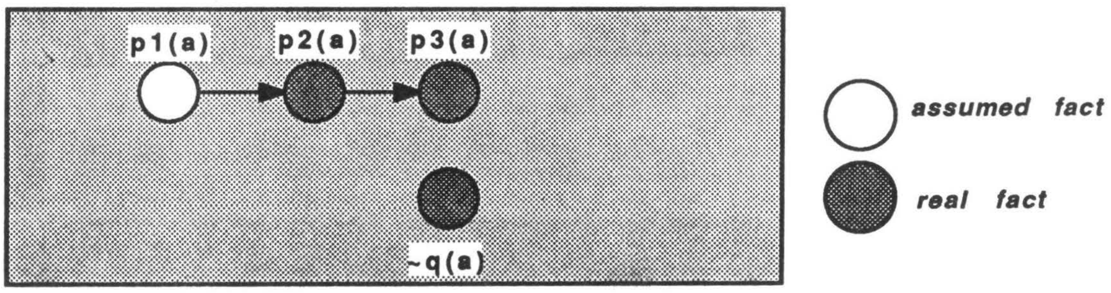


(a) Database after the execution of rule1 to rule4



(b) Database after the execution of rule5

Fig. 16. Changes in the database (cont.)



(c) Database after the execution of rule6

Fig. 16. Changes in the database

4.4.4. Design of a two degrees-of-freedom robot

In this section, we show a comparatively complicated design example of two degrees-of-freedom robots (Fig. 17). The design process is described by the execution sequence (i.e. the sequence of the executed *select* built-in predicates) of the IDDL program in Fig. 18. Figure 17 shows the design solution.

This example shows that IDDL has potential for describing fairly wide range of design, provided the design process can be explicitly *downloaded* in IDDL. This essentially means that IDDL programming using the IDDL browser (Fig. 7) is equivalent to designing a new type of design objects, whereas just executing existing IDDL codes is a routine design.

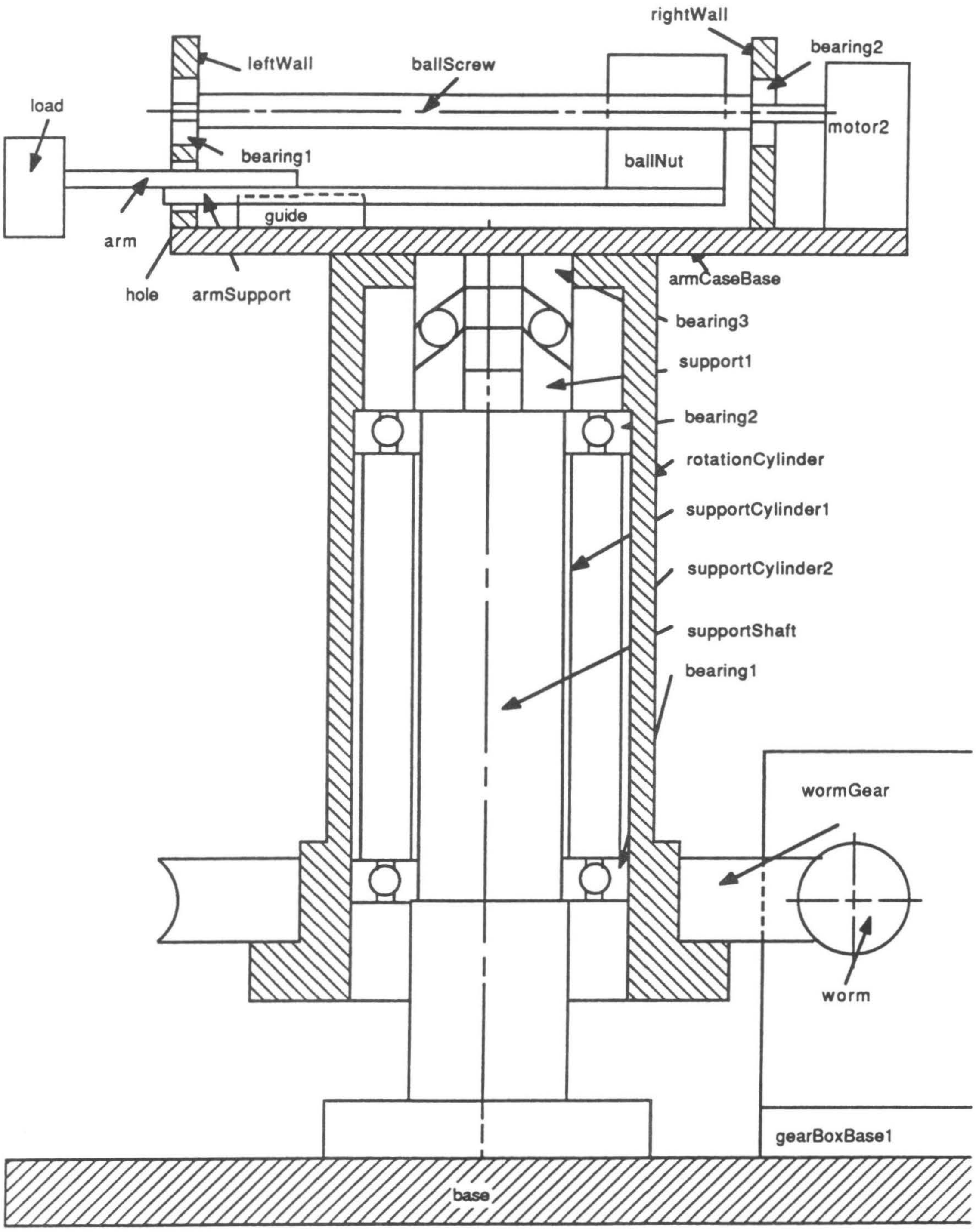


Fig. 17. The structure of the designed robot

```
<1>: root.
<2>: select(twoDimensionRobotDesign,specification)
<3>: select(twoDimensionRotationSlidingRobotDesign,Specification)
<4>: select(firstJointTypeForRotationSlidingRobotSelection,Specification)
<5>: select(rotationMovingEvaluation,Specification)
<6>: select(evaluateGearGearPair,Specification)
<7>: select(evaluateWormGearPair,Specification)
<8>: select(twoDimensionRotationSlidingRobotType1Design,Specification)
<9>: select(makeBaseFixationWork,base,supportShaft,Specification)
<10>: select(makeFixationWork,Shaft,Base)
<11>: select(determineThinPlateHoles,Object1,Object2,screw1,screw2,screw3,screw4)
<12>: select(determineThickPlateHoles,Object1,Object2,screw1,screw2,screw3,screw4)
<13>: select(makeGearBoxFixationWork,gearBoxBase1,base,Specification)
<14>: select(makeFixationWork2,GearBoxBase,Base)
<15>: select(determineThinPlateHoles,Object1,Object2,screw1,screw2,screw3,screw4)
<16>: select(determineThickPlateHoles2,Object1,Object2,screw1,screw2,screw3,screw4)
<17>: select(makeRotationWork,supportShaft,rotationCylinder,gearBoxBase1,Specification)
<18>: select(makeRotationPair,SupportShaft,RotationCylinder,Specification)
<19>: select(shaftDesign,Shaft,Specification)
<20>: select(radialBearingDesign,bearing1,bearing2,Shaft)
<21>: select(rotationCylinderDesign,Cylinder,bearing1,bearing2,Shaft)
<22>: select(axialBearingDesign,bearing3,Shaft,Cylinder)
<23>: select(supportCylinderDesign,supportCylinder1,supportCylinder2,Shaft,Cylinder,bearing1,bearing2)
<24>: select(fixWormGear,wormGear,RotationCylinder)
<25>: select(makeWormGearPair,wormGear,worm,Specification)
<26>: select(gearBox1IDetermination,Specification)
<27>: select(wormDesign,Worm,WormGear,Specification)
<28>: select(wormSupportBearingDesign,wormSupportBearing1,Worm)
<29>: select(wormSupportBearingDesign,wormSupportBearing2,Worm)
<30>: select(makeGearBox,motor1,Specification,GearBoxBase)
<31>: select(twoStepGearBoxDesign,Motor,Specification,GearBoxBase)
<32>: select(twoStepGearBoxSupportDesign,gear1,gear2,gear3,gear4,GearBoxBase)
<33>: select(makeFixationWork,rotationCylinder,armCaseBase)
<34>: select(determineThinPlateHoles,Object1,Object2,screw1,screw2,screw3,screw4)
<35>: select(determineThickPlateHoles,Object1,Object2,screw1,screw2,screw3,screw4)
<36>: select(makeSlidingWork,armCaseBase,Specification)
<37>: select(ballScrewSlidingPairDesign,ArmCaseBase,Specification)
<38>: select(caseBaseDesign,CaseBase,Specification)
<39>: select(twoWallsFixation,leftWall,rightWall,CaseBase,Specification)
<40>: select(makeBallScrewPair,ballScrew,ballNut,Specification)
<41>: select(armDesign,arm,armSupport,guide,rightWall,CaseBase,Specification)
<42>: select(guideDesign,Guide,CaseBase,RightWall)
<43>: select(holeDesign,hole,leftWall,arm,armSupport,guide)
<44>: select(ballScrewSupportDesign,ballScrew,leftWall,rightWall)
<45>: select(gearBox2Design,CaseBase,Specification)
```

Fig. 18. The execution sequence of the IDDL program

5. CONCLUSIONS

This paper has reported the development of an experimental version of IDDL (Integrated Data Description Language) for the IIICAD system which was implemented on Smalltalk-80. It was discussed that IDDL should have the capability of describing both design processes and design objects, and this point of view is most missing from current ICAD research.

IDDL has the concepts of scenarios to represent design processes, objects for design objects, and functions for operations over objects. The concept of worlds is useful to represent an area of interest or a situation in which the design currently takes place. The scenario mechanism together with the multi-world mechanism and the ATMS-like default reasoning facility allows various kinds of design techniques such as the trial-and-error method.

For future work, the following issues can be pointed out.

(1) Research about more domain dependent design knowledge is needed, because IDDL only provides a framework to represent design knowledge. It must be clarified how real design processes is driven by which kind of knowledge in e.g. mechanical engineering. For this purpose, qualitative physics as a modeling framework might be necessary [Murthy and Addanki 1987; Faltings 1987; Kurumatani, Tomiyama, and Yoshikawa 1989].

(2) IDDL assumes only that design is a stepwise refinement process. This is not perhaps sufficient to model real designer's thought process and we need a more elaborated design process model, preferably, based on logic [Takeda, Tomiyama, and Yoshikawa 1989]. This may further requests a sophisticated design information management system beyond ATMS.

(3) Although one of the main goals of IIICAD is the integration of design knowledge [Tomiyama and ten Hagen 1987], the current prototype of IDDL does not have such an ability. For this, we need to incorporate the metamodel mechanism which integrates various kinds of models of design objects in a single, uniform framework [Tomiyama and ten Hagen 1987b; Tomiyama *et al.* 1989].

ACKNOWLEDGEMENT

We would like to thank students of Yoshikawa-Tomiyama Laboratory of the Department of Precision Machinery Engineering, Faculty of Engineering, The University of Tokyo, for their help and constructive comments to this study. Especially, we would like to mention that Susumu Hamada made important contributions compiling the IDDL language specifications from unorganized materials and providing good working examples. We would like to thank Professor Hiroyuki Yoshikawa of the University of Tokyo for his valuable advices and continuous support.

REFERENCES

[de Kleer 1986]
de Kleer, J: ''An assumption-based TMS,'' *Artificial Intelligence,* No. 28, pp. 127-162.

[Faltings 1987]
Faltings, B.: ''Qualitative kinematics in mechanisms,'' *Proceedings of IJCAI-87,* pp. 436-442.

[Gero 1987]
Gero, J.S. (ed.): *Expert Systems in Computer-Aided Design,* North-Holland, Amsterdam.

[Goldberg and Robson 1983]
Goldberg, A. and Robson, D.: ''Smalltalk-80: The Language and its Implementation,'' Addison-Wesley, Reading, MA, USA.

[ten Hagen and Tomiyama 1987]
ten Hagen, P.J.W. and Tomiyama, T. (eds.): *Intelligent CAD Systems I: Theoretical and Methodological Aspects,* Springer-Verlag, Heidelberg.

[ten Hagen, Tomiyama, and Akman 1989]
ten Hagen, P.J.W., Tomiyama, T., and Akman, V. (eds.): *Intelligent CAD Systems II: Implementational Issues,* Springer-Verlag, Heidelberg, in printing.

[Kurumatani, Tomiyama, and Yoshikawa 1989]
Kurumatani, K., Tomiyama, T., and Yoshikawa, H.: ''Qualitative representation of machine behaviors for intelligent CAD systems,'' *Journal of Mechanism and Machine Theory, Special Issue on ''Computational Theories of Design — Applications to the design of machines,''* in printing.

[Megens 1987]
Megens, M.: ''An Implementation of a Simple Design Description Language,'' Master's thesis, Computer Science Department, University of Amsterdam.

[Murthy and Addanki 1987]
Murthy, S.S. and Addanki, S: ''PROMPT: An innovative design tool,'' in [Gero 1987], pp. 323-347.

[Takeda, Tomiyama, and Yoshikawa 1989]
Takeda, H., Tomiyama, T., and Yoshikawa, H.: ''Logical formalization of design processes for intelligent CAD systems,'' to appear in [Yoshikawa and Holden 1989].

[Tomiyama 1989]
Tomiyama, T.: ''Object oriented programming paradigm for intelligent CAD systems,'' in [ten Hagen, Tomiyama, and Akman 1989], in printing.

[Tomiyama *et al.* 1989]
Tomiyama, T., Kiriyama, T., Takeda, H., Xue, D., and Yoshikawa, H.: ''Metamodel: An key to intelligent CAD systems,'' *Research in Engineering Design,* Vol. 1, No. 1, in printing.

[Tomiyama and ten Hagen 1987]
Tomiyama, T. and ten Hagen, P.J.W.: ''The Concept of Intelligent Integrated Interactive CAD Systems,'' CWI Report No. CS-R8717, Centre for Mathematics and Computer Science, Amsterdam.

[Tomiyama and ten Hagen 1987a]
Tomiyama, T. and ten Hagen, P.J.W.: ''Representing Knowledge in Two Distinct Descriptions: Extensional vs. Intensional,'' CWI Report No. CS-R8718, Centre for Mathematics and Computer Science, Amsterdam, (also submitted for publication to *the International Journal for Artificial Intelligence in Engineering*).

[Tomiyama and ten Hagen 1987b]
Tomiyama, T. and ten Hagen, P.J.W.: ''Organization of design knowledge in an intelligent CAD environment,'' in [Gero 1987], pp. 119-147.

[Veerkamp 1989]
Veerkamp, P.: ''Multiple worlds in an intelligent CAD systems,'' in [Yoshikawa and Gossard 1989], in printing.

[Veth 1987]
Veth, B.: ''An integrated data description language for coding design knowledge,'' in [ten Hagen and Tomiyama 1987], pp. 295-313.

[Yoshikawa and Gossard 1989]
Yoshikawa, H. and Gossard, D.C. (eds.): *Intelligent CAD 1, Proceedings of the First IFIP Working Group 5.2 Workshop on Intelligent CAD, 6-8 October 1987, Cambridge, MA, USA,* North-Holland, Amsterdam, in printing.

[Yoshikawa and Holden 1989]
Yoshikawa, H. and Holden, T. (eds.): *Intelligent CAD 2, Proceedings of the Second IFIP Working Group 5.2 Workshop on Intelligent CAD, 19-22 September 1988, Cambridge, UK,* North-Holland, Amsterdam, in preparation.

APPENDIX: THE SYNTAX OF IDDL

```
<scenario>        ::= <declarations> <rule block>

<declarations>    ::= <object decl> <function decl>

<object decl>     ::= OBJECT <object list>

<object list>     ::= <predicate> ,| <predicate> , <object list>

<function decl>   ::= FUNCTION <function list>

<function list>   ::= <external func> ,
                    | <external func> , <function list>

<external func>   ::= <function name> from <category name>

<category name>   ::= <identifier>

<rule block>      ::= BEGIN <rules> END

<rules>           ::= <rule> ,| <rule> , <rules>

<rule>            ::= <wff> |  <if> <wff> THEN <wff>

<if>              ::= IF | IFF | IFU

<wff>             ::= <fact> |  <wff> <binary l-op> <wff>| ( <wff> )

<fact>            ::= <general fact> | <modal l-op> <general fact>

<general fact>    ::= <basic fact> | <default l-op>  <basic fact>

<basic fact>      ::= <l-const> | <predicate>
                    | <unitary l-op> <predicate>

<predicate>       ::= <identifier> | <identifier> ( <term list> )

<term list>       ::= <expression> | <expression> , <term list>

<modal l-op>      ::= #N | #P

<default l-op>    ::= #D

<unitary l-op>    ::= ~ | %

<binary l-op>     ::= & | |

<l-const>         ::= T | F | U

<function>        ::= <function name> [ <term list> ] <function body>

<function name>   ::= <identifier>

<function body>   ::=  <local assign> <return value>

<local assign>    ::= <assignment> | <assignment> <local assign>

<assignment>      ::= <variable> :=  <expression> ,

<return value>    ::= ^ <expression>
```

Paper Session *Geometric Reasoning*

Rajiv S. Desai, Rajkumar S. Doshi,

Raymond K. Lam

GARE: Geometric Analysis and Reasoning Engine

GARE: Geometric Analysis and Reasoning Engine

Rajiv S. Desai, Rajkumar S. Doshi, Raymond K. Lam

Artificial Intelligence Group
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, U.S.A.

Abstract

Geometric modeling is a crucial component in the design and operation phases of modern automated systems. Automated robot task planning systems require complex geometric reasoning and motion planning capabilities. The basic concept underlying the design of Geometric Analysis and Reasoning Engine (GARE) is that many complex geometric reasoning problems can be solved by combining simple geometric algorithms. GARE consists of a geometric modeling system, a geometric task description system, and a geometric reasoning system. The modeling system allows for modeling 3D objects, assemblies, articulated objects and physical constraints. A hierarchical representation scheme is used to represent object and constraint models. The task description system allows the user or an interacting system to specify geometric reasoning tasks to GARE. The geometric reasoning system consists of various geometric algorithms and control routines. Using a rule base, GARE can combine various geometric algorithms to solve geometric problems. Typical problems that GARE can handle include, determining stable positions of objects, obstacle free paths for robots, determining assembly/disassembly sequences etc. GARE is also integrated with the motion planning system Handey. GARE was motivated by the need of geometric reasoning in robot task planning and a robot fault diagnosis systems. However, the concept of GARE is applicable to other domains.

1.0 Introduction

Computer modeling is a crucial and integral part of the modern automated systems, such as robot systems. A model based automated system must represent the system, environment of the system and, the complex interactions among the components of the system and the environment. The models must reflect the behavior of the system and that of the environment. In general modeling techniques dependent on the characteristics of the system and the function of that system. Several commercial CAD modeling systems allow sophisticated modeling features. Detailed surveys of geometric modeling systems can be found in [1] [2].

While several very sophisticated geometric modeling systems have been developed there are very few that allow for modeling and analysis of the physical processes. Usually modeling of physical processes is done separate from the geometric modeling. Typical analysis systems include dynamic simulation, finite element analysis, geometric simulation of manufacturing processes [3][4].

The Geometric Analysis and Reasoning Engine (GARE) was motivated by the need for a system for geometric reasoning during development of robot task planning [5] and knowledge based diagnostic systems [6] for NASA's telerobot project. Typical robot tasks involve reasoning about collision free path, finding stable poses of objects, determining assembly sequences etc. An automated robot system may consists of robots, parts, sensors, computer vision systems etc. A typical task may require assembling a set of parts in some prespecified configuration. This can be posed as a geometric planning problem, where the world can be represented by a geometric model, then the problem is equivalent to moving the parts from one geometric configuration to another in that world. However this may require reasoning about several geometric problems, e.g., generation of assembly

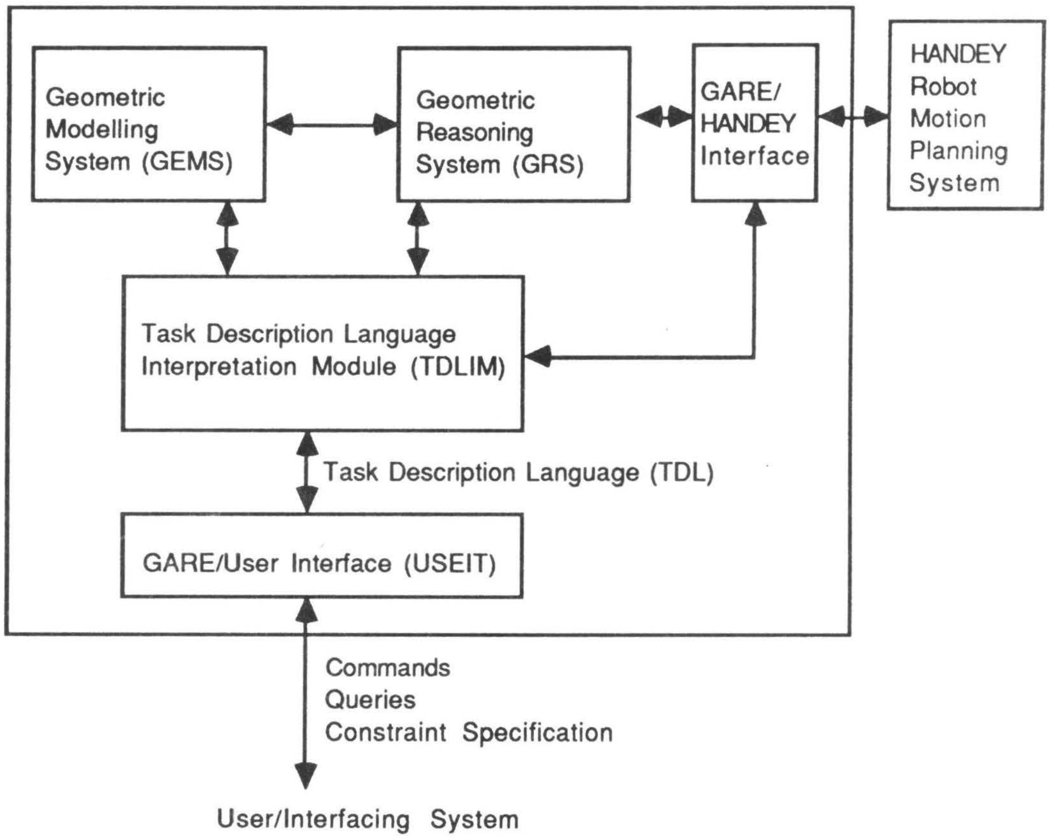

Geometric Modelling System (GEMS)
Geometric Reasoning System (GRS)
GARE/ HANDEY Interface
HANDEY Robot Motion Planning System
Task Description Language Interpretation Module (TDLIM)
Task Description Language (TDL)
GARE/User Interface (USEIT)
Commands
Queries
Constraint Specification
User/Interfacing System

Figure 1: GARE: Geometric Analysis and Reasoning Engine

sequences, planning collision free paths for the robots, determining grasp positions etc. In general such a system must be able to reason about the complex geometric interactions among it's different components, such as, interference among objects, stable positions of objects, kinematic constraints of the system etc. Geometric Analysis and Reasoning Engine (GARE) is a general purpose tool for reasoning about various complex geometric interactions.

2.0 Geometric Analysis and Reasoning Engine (GARE): An Overview

GARE consists of four major subsystems, namely, geometric modeling system, geometric reasoning system, task description language interpretation module (TDLIM) and GARE/user interface as shown in Figure 1.

The geometric modeling system is used to create the the model of the environment. Geometric reasoning systems (GRS) is a collection of geometric algorithms, geometric and motion analysis routines, and controls routines. A user or an interfacing system can make a geometric reasoning task specification using the task description language (TDL). TDLIM interprets the TDL commands and passes the commands to appropriate subsystem. Each of these systems are described later in more detail.

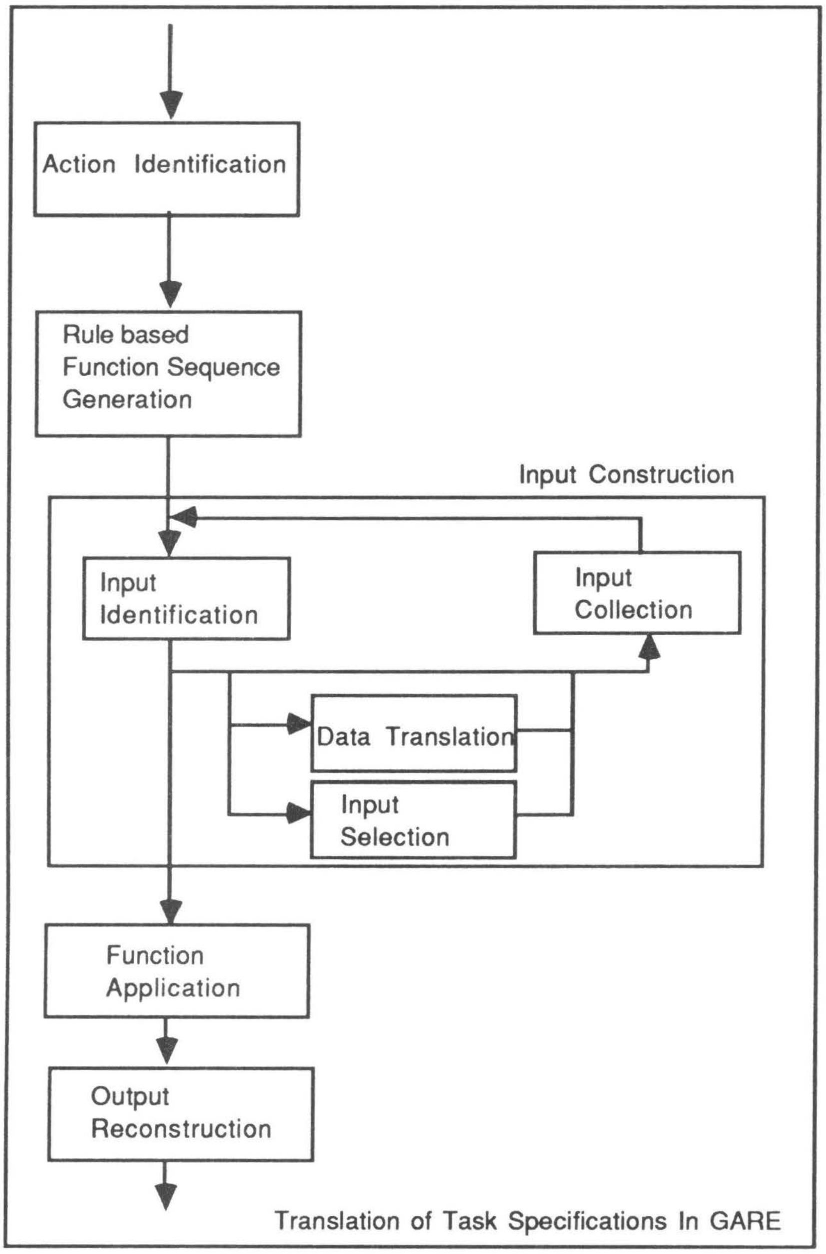


Figure 2: Translation of Task Specifications in GARE

The GARE system is based on the idea that many complex geometric problems may be solved by combining solution methods for simpler geometric problems. Consider the task of determining a position for an object such that object is statically stable. The stable poses of an object can be found be found by computing the convex hull of an object and determining the stable faces on the convex hull. This geometric problem can be solved by appropriately combining property computation routines (convex hull, center of mass etc.) and performing a static stability analysis for each pose. Now suppose the problem was to determine stable positions that are graspable by a given robot. This problem can be solved by combining the previous computations with selector function that reduces a set of positions to a set of positions that are graspable. Now, the problem could be further complicated by requiring that the positions should be stable, graspable and reachable by the robot. GARE would apply another selector function to the output of the previous problem that reduces a set of positions to set of positions reachable by a robot.

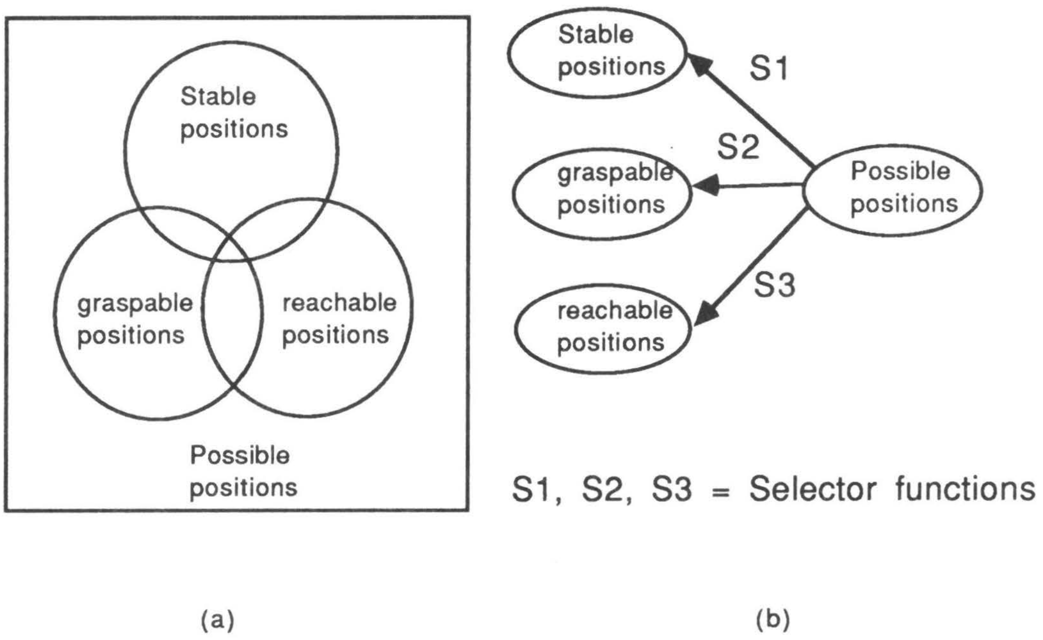


Figure 3: Selector Functions: (a) Partitions of positions by constraints (b) Selector transition graph

When a task specification is presented to GARE, the TDLIM parses the input and then performs task translation as shown in Figure 2. Given a geometric problem like the one described above, GARE attempts to construct a solution plan, which is an ordered list of function calls to various geometric computation modules in the geometric reasoning system, and determines the inputs at each stage in the plan. Solution plans are generated using a rule base. If the inputs are not available, appropriate input constructions functions are invoked.

Input construction is done by determining the availability of the data at correct level of representation (logical, topological or metric). If the data is not available at the correct level of abstraction, data transformation routines are invoked. Sometimes there are some constraints that may need to be applied before inputs are collected. For example, given all the stable positions of an object only the positions that are reachable by a specified robot may be of interest as shown in Figure 3a. GARE determines the appropriate selector functions using the selector rule base and selector transition graph (shown in Figure 3b).

Once all the inputs for each function call to the geometric reasoning system are constructed, functions are applied in order. Finally, the output at each stage of the plan is reconstructed to conform to the next function call or desired output representation. The rules for ordering the functions and determining appropriate data transformation routines and selector functions are predetermined by a human expert.

Consider the example of a command to find space for an object such that it is stable, graspable and reachable by the robot. Figure 4 depicts the task translation. The command is parsed by the TDLIM and action is identified to be 'find space' and the constraints are also translated to appropriate level of abstraction. The GARE rule base associates the 'ind space' action with the function find_space(face_list1, face_list2) function in GRS. Next GARE checks if any constraints were specified. For each constraint (i.e., stable, graspable and reachable) a function sequence or selector function is determined by constraint analysis in GRS. Now the functions are ordered according to a solution plan and input construction is performed. Finally, each function is invoked preceded and proceeded by corresponding input construction and output reconstruction routines respectively.

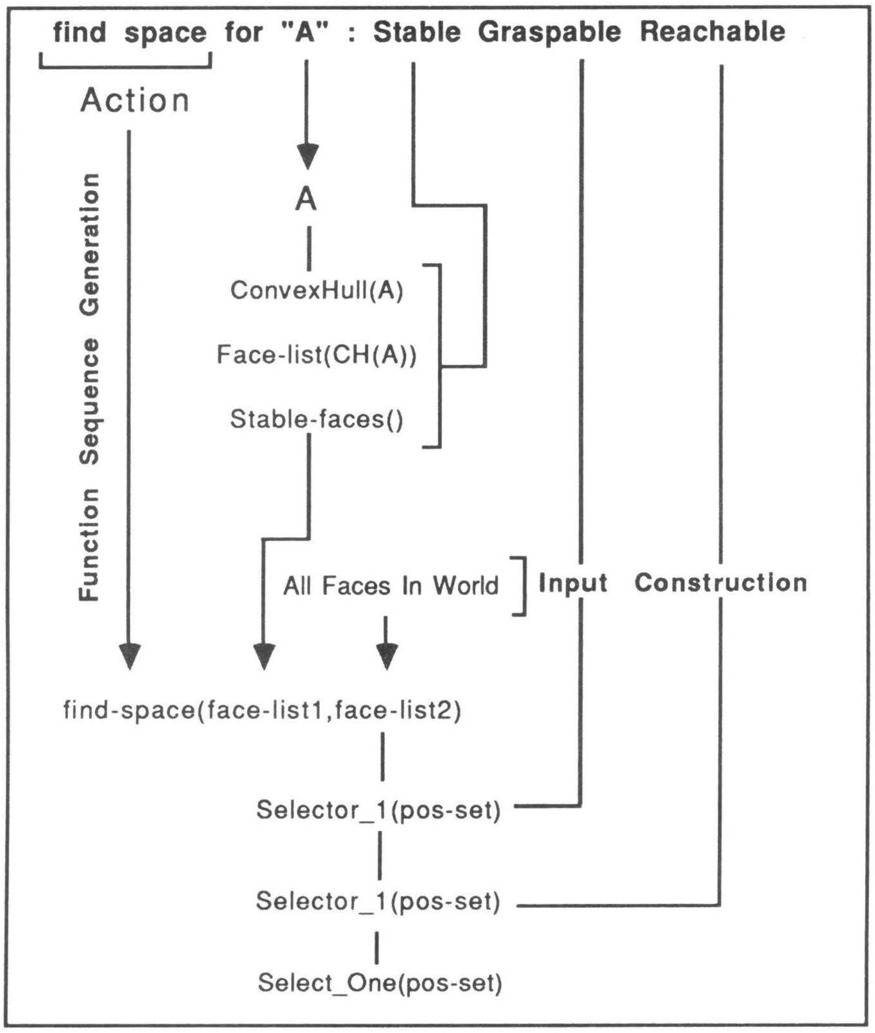


Figure 4: TDL Task translation for a find space with constraints command

An important component of a modeling system is it's user interface. The interface besides being easy to use must also be able to prompt the user to reasonable and correct inputs. The system must be able to detect incorrect inputs and suggest alternatives. GARE's user interface (USEIT) consist of a graphical input system and a command line interface. A user can not only specify the basic modeling commands but also complex assembly and kinematic relations among objects via this interface. A special feature of the interface is that the input to the modeling system may be provided any of the several levels of abstractions. For example, position of an object may be specified by specifying exact coordinates or it's relative contact conditions with respect to other objects in the environment. GARE internally reduces all specifications to appropriate canonical forms. When specifying some inputs, GARE checks for completeness, correctness, and consistency of the input. For example, if a position and orientation of a block is to be specified and if only a vertex position is specified the system would prompt for more input. On the other hand if redundant or inconsistent inputs are provided, the system would indicate that as well. All inputs specifying the object configurations directly or indirectly are translated into canonical forms consisting of algebraic relations and constraints. For instance, input that a vertex should lie on a face is translated to set of linear algebraic constraints. The consistency of the input is then tested by verifying the consistency of these constraints by simultaneously solving the algebraic inequalities.

Next, we describe the major subsystems of the geometric analysis and reasoning engine.

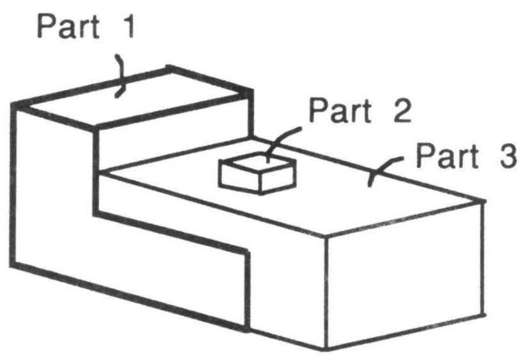


Figure 5: A three part assembly

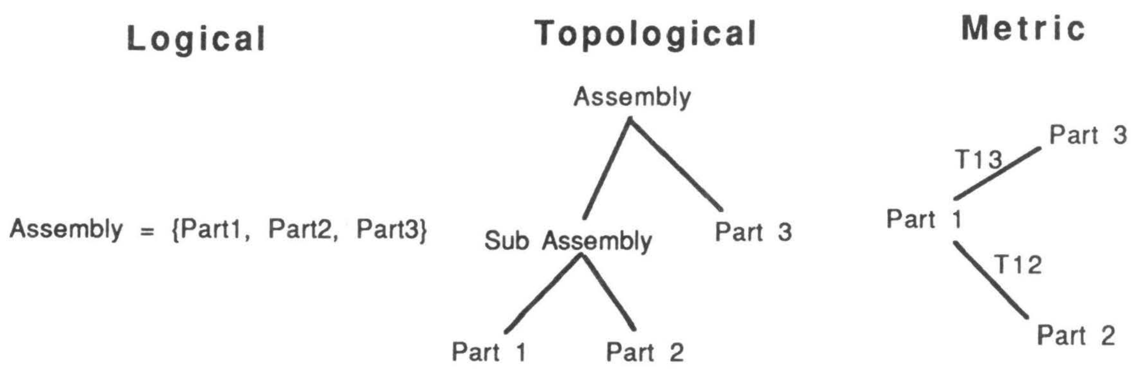


Figure 6: Hierarchical representation for assembly in Figure 5: Logical, Topological, Metric levels

3.0 GEMS : Geometric Modeling System

The Geometric Modeling System (GEMS) of GARE is used to create and manipulate the geometric models of objects. GEMS is a simple boundary representation based general purpose 3D object modeling systems. Currently, the modeling is limited to polyhedral objects. GEMS also allows for modeling assemblies and articulate objects (e.g., robot, linkages etc.). Besides modeling mechanical objects such as assemblies and articulate objects, GEMS also allows for modelling two basic types of physical constraints for object models, namely, contact constraints, which are modeled using the concept of contact formations [7], and non-contact constraints such as motion ranges, allowable motion types etc.

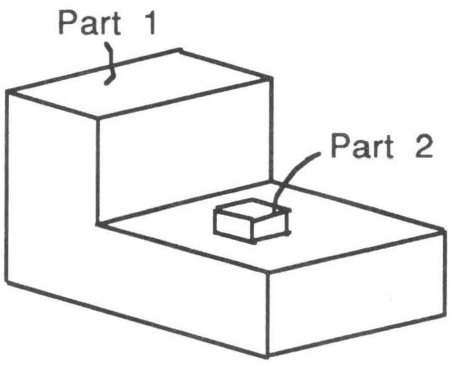


Figure 7: Two contacting Parts

Logical	**Topological**	**Metric**
Part 1 contacts Part 2	F1 contact F2	$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 < b_1$
		$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 < b_2$
		$\vdots$

Figure 8: Hierarchical representation of contact constraint for contact shown in Figure 7: Logical, Topological, Metric levels

3.1 Knowledge Representation in GEMS

Various representations schemes are used in solid modeling [8][9][10][11]. GEMS uses a hierarchical representation of objects and constraints. There are three types of representations in GEMS; logical, topological and metric. Consider the assembly depicted in Figure 5; as shown in Figure 6, at logical level the assembly is represented as a collection of parts 1, 2 and 3. At the topological level the assembly's structure is represented by an assembly tree, which reflects the order of assembly. Finally, at the metric level the relative transformations among the objects are stored. Object topology in GEMS is represented using the wind-edge data structures [12][13][14].

The constraints are also represented at the logical, topological and metric levels. Consider the contact constraint shown in Figure 7. As shown in Figure 8, at the logical level a contact is represented by a pairs of objects in contact, at the topological level contacts are represented as contact formations. At the metric level the contacts is represented by system of algebraic equations.

Similar hierarchical representations are used to model articulated objects such as robots and linkages. GEMS also supports a simple graphics system that forms the basis for USEIT.

4.0 GRS: Geometric Reasoning System

As mentioned above, geometric reasoning system is a collection of geometric algorithms, geometric and motion analysis routines, and control routines. In addition GRS also has data transformation functions, and functions for interacting with the Handey robot motion planning system.

The geometric algorithms are of two types, namely property computation algorithms and geometric interactions routines. Property computations routines include integral property computation procedures [15][16], such as, volume, center of mass, etc., and, geometric property routines, such as, convex hull, swept volumes etc. [17][18]. The geometric interaction algorithms include intersection computations, distance functions etc.

The control routines include, selection functions and data transformation routines. As we discussed in examples above, selection routines are used to select data from a set by application of constraints to members of that set. Data transformation routines are invoked when it is necessary to transform data from one representation to other. For example if a user specifies a contact between a vertex of on object and face of another object. The data transformation routines can convert this topological information to equivalent algebraic inequalities.

As we mentioned above, GEMS has special representation of contacts. The geometric reasoning system can reason about contacts from first principles to derive several very useful properties of objects in contact, such as, separation and compliant directions. Set of routines in GRS that perform such analysis and analysis of degrees of freedom of systems, static force analysis etc. are collectively know as geometric and motion analysis routines. These include routines used to analyze the kinematics of articulated objects, relative degrees of freedom among contacting objects etc. One of the crucial components in the analysis of a design or in robot task planning, is motion analysis. GARE is interfaced with a robot motion planning system Handey [22], which was developed at MIT. Handey system provides path planning and grasping planning capabilities to GRS.

Several geometrical problems lend themselves to simple algebraic formulations. The algebra engine, a subsystem of the geometric reasoning system, is a tool for solving simple algebraic problems such as solving a set of linear equations.

Geometric reasoning system also has an assembly analysis engine. GRS can generate simple assembly and dissassembly sequences given the model of an assembly. The assembly /dissassembly sequences are generated from local analysis of contacts and by interacting with the path planner. Only geometric knowledge is used. More complete treatment of this problem can be found in [20] [21].

5.0 TDL: Task Description Language

As we mentioned above, all interactions with GARE must be done using the task description language (TDL). The TDL constitutes a collection of procedures through which a user or other systems can interact with GARE. TDL commands can be presented to GARE graphically, on command line or via function calls. Typically, graphics and command line input are given by a user, while interacting systems such as robot task planner or diagnosis system interact with GARE via function calls.

As shown in Figure 9. There are three modes in TDL, namely, the command mode, query mode and constraint specification mode. In the command mode, the interacting system can directly command the GARE subsystems to perform some task. For example the user can issue a command to GEMS to create a model or set viewing parameter in the graphics system.

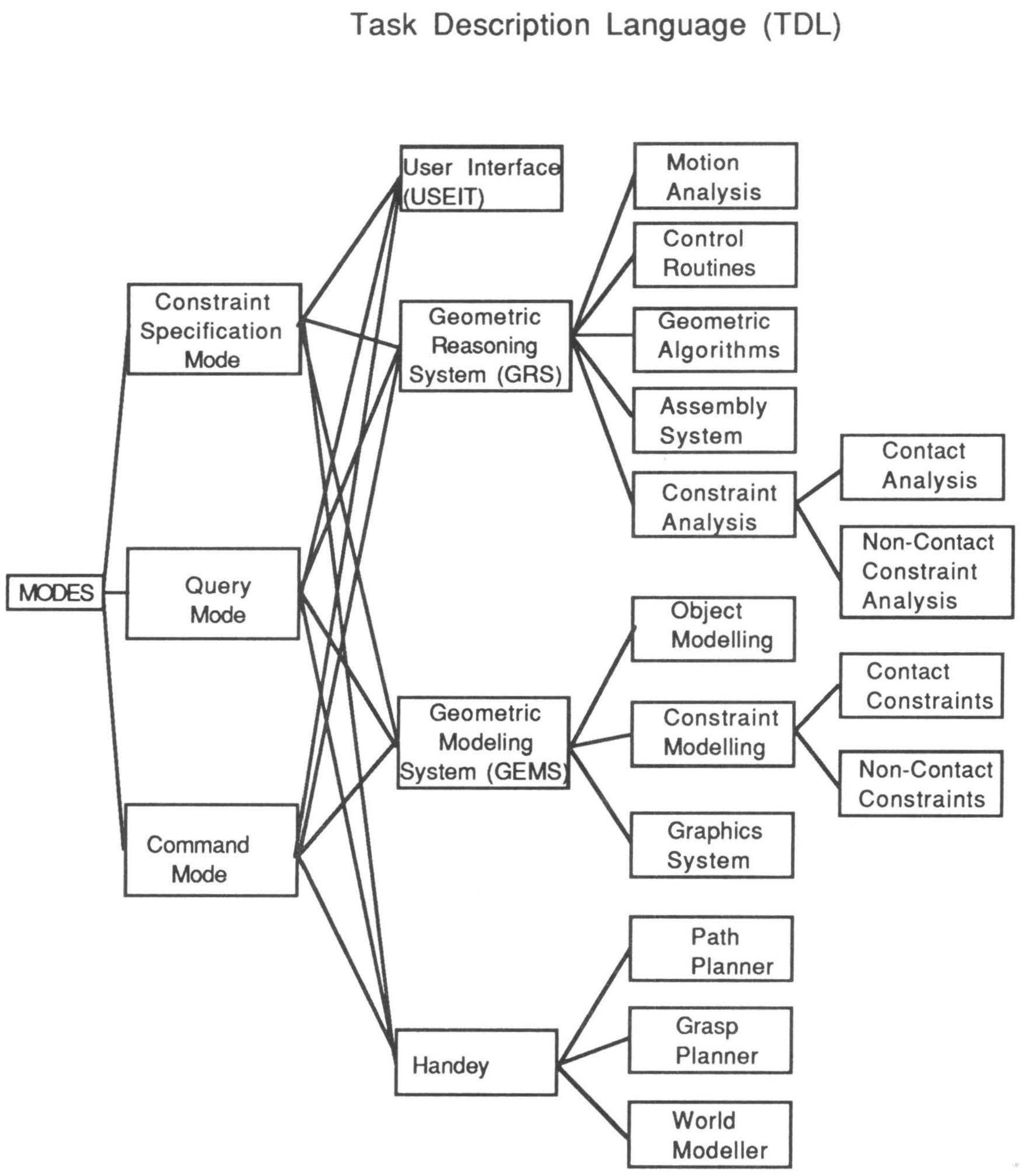


Figure 9: Task Description Language (TDL): Three modes (1) Constraints Specification Mode, (2) Query Mode, (3) Command Mode

All TDL interactions are processed by task description language interpretation module (TDLIM). TDLIM parses the TDL inputs and depending on whether it is a command, a query or a constraint specification, it either directly sends it to the appropriate module, or to GARE task specification translator that was discussed earlier.

There are two types of queries, passive and active. The passive queries seek information that is already available in the system and only needs to be looked up, such as topology of a predefined assembly. Therefore passive queries are directly passed to the appropriate modules of GARE. The active queries require computation and they are resolved by GARE as task specifications. For instance queries to determine if an object is convex or whether a linkage system is statically stable require computation. These queries are translated into task specifications by TDLIM.

In the constraint specification mode, constraints are posted in appropriate module by TDLIM. Suppose a user specifies a contact between the vertex of one object and face of another object, the constraint is immediately transformed to metric level. Constraint is then verified with the respect to the geometric data for consistency. If there are other constraints at metric level that were predefined and apply to the current pair of objects, then they are also tested for consistency against the new constraint. If the constraints are mutually consistent the new constraint is accepted, otherwise the user is advised to change the new constraint or delete the old constraints. If the interacting system uses function calls, the new constraints is rejected.

As mentioned before TDL allows the user to specify constraints at any level of abstraction. The data transformation, when possible are done automatically. At the time of task specification translation, GARE accounts for all the constraints in the system.

6.0 Implementation

The GARE system described above is being implemented on the Symbollics Lisp machines in LISP. A prototype version of the major components of the GARE system, namely, the geometric modeling system (GEMS), the geometric reasoning system and simple integration with Handey system have been completed. Typical geometric problems GARE can currently solve include, find space with constraints, determining assembly/dissassembly sequences for simply separable objects (i.e., objects can be removed one at a time; no sub-assemblies are required), determining obstacle free paths with constraints on paths and grasps, determine degrees of freedom and allowable ranges for linkages.

The current implementation of the GARE system was done with little consideration to computing efficiency. Typical geometric queries can take anywhere from a few seconds to several minutes depending on the complexity of the task specification. The tasks that involve interaction with the motion planning system are relatively slow because, first the path planning and grasp planning algorithms have relatively high computational complexities, secondly, the Handey system uses a different solid modeling system and therefore, considerable amount of computation time is spent in translation of data from the GEMS representation to the representation used by the geometric modeler in the Handey system.

As we described above, whenever a constraint is presented to GARE, it tries to check for it's to verify it's consistency with the existing knowledge. In case of geometric and contact constraints, this requires simultaneous solution to a system linear equalities and inequalities. At first, we implemented the algebra system such that it would solve for equations symbolically. However, this was found to be computationally very expensive. The new version of algebra engine uses fast optimization techniques to converge to solution. While it computationally more efficient, it is not guaranteed to find a solutions as the symbolic algebra system did.

The prototype GARE system has been tested for simple robot task plans and motion planning problems. The performance of the system is quite satisfactory except for the processing time.

7.0 Conclusions

GARE is an evolving system. Current emphasis is on identifying the problems and merits of this approach. The prototype implementation has been successfully used to design and reason with simple robot task planning scenarios that included linkages and assemblies.

The future research is aimed at extending the modeling and the reasoning systems to include curved surfaces, deal with representation of and reasoning with the uncertainty in geometry. Often the spatial models are constructed not by user, but by an external sensing system, such as a vision system or a range imaging system. However the data available from these sensing modalities can only give information about spatial occupancy. We will explore the issues of modeling and reasoning using multiple representation schemes.

Geometrical modeling is used in numerous other domains such as automated manufacturing, computer vision etc. Automated systems must be able to reason about geometrical interaction. Therefore concept of GARE should be applicable in a wide variety of domains.

References

[1] Requicha A., Voelker H., "Solid Modeling: A Historical Summary and Contemporary Assessment", *IEEE Computer Graphics and Applications*, 2 (2):9-24, March 1982.

[2] Baer A., Eastman C. M., Henrion M.,"Geometric Modeling: A Survey", *Computer Aided Design*, 11 (5):253-272,1979.

[3] Voelker H. B., Requicha A.,"Geometric Modeling of Physical Parts and Processes",*IEEE Computer* , 10(2):pp 48-57,1977.

[4] Boyse J., Gilchrist J.,"GMSolid: Interactive Modeling for design and analysis of solids", IEEE Computer Graphics and Applications, 2 (2):86-97, March 1982.

[5] Doshi R., Desai R., Lam R., White J.,"Integration of Artificial Intelligence Planning and Robotic Systems with AIROBIC", *Proc. IMAC Conf on Expert Systems for Numerical Computing*, Dec 1988

[6] Lam R., et al,"Diagnosing Faults in Autonomous Robot Plan Execution", *Proc. SPIE Conf.*, November 1988

[7] Desai R. S., "On Fine Motion in Mechanical Assembly In the presence of uncertainty", *Ph.D. Thesis*, The University of Michigan, Ann Arbor, 1988.

[8] Yamaguchi K., Kunii T., Fujimura K., Toriya H., "Octree related data structures and algorithms", *IEEE Computer Graphics and Applications*, 4(1):pp 53-59, 1984.

[9] Requicha A. A. G.,"Representation of solid objects-theory, methods, and systems.", *ACM Computing Surveys*, Vol 12, No.4, pp 437-464, Dec 1980.

[10] Requicha A. A. G.,"Mathematical Models for Rigid Solids", *Tech Memo No. 28,* Production Automation Project, University of Rochester, 1977.

[11] Reddy D. R., Rubin S., "Representation of Three-Dimensional Objects",*Technical Report,* CMU-CS-78-113, Computer Science Department, Carnegie Mellon Univ., Pittsburgh, Apr 1978

[12] Baumgart B., "Geometric Modeling for Computer Vision", *Ph. D. Thesis,* Stanford University, 1974.

[13] Baumgart B., "A polyhedron representation for computer vision", *Proc of AFIPS Conf.,* pp589-596, 1975.

[14] Yamaguchi F., Tokieda T., "Bridge edge and triangulation approach to solid modeling", *Proc. Computer Graphics,* Tokyo, 1984.

[15] Timmer H. G., Stern J. M.,"Computation of global geometric properties of solid objects", *Computer Aided Design,* Vol 11, No 6, Nov 1980

[16] Lee, Y.T., Requicha A. A. G,"Algorithms for computing the volume and other integral properties of solid objects.I. Known methods and open issues.",*Communication of ACM*, Vol. 25, No. 9, pp635-641, Sep 1982.

[17] Lee, Y.T., Requicha A. A. G,"Algorithms for computing the volume and other integral properties of solid objects.II. A family of algorithms based on representation conversion and cellular approximation.",*Communication of ACM*, Vol. 25, No. 9, pp642-650, Sep 1982.

[18] Shamos, M. I., Preparata F.,"Computation Geometry", Springer Verlag, New York, 1985.

[19] Faux I. D., Pratt M. J., "Computation Geometry for Design and Manufacture", John Wiley, New York, 1979.

[20] Wolter J.,"On Automatic Generation of Plans for Mechanical Assembly", *Ph.D. Thesis*, The University of Michigan, Ann Arbor, 1988.

[21] Homem de Mello L., Sanderson A. C., "Planning repair sequences using AND/OR graph representation of assembly plans", *Proc. IEEE Conf. Rob. & Auto.*, pp 1861-1862, 1988.

[22] Lozano-Perez T., et al.,"Handey: A Robot System that Recognizes, Plans, and Manipulates", *Proc. IEEE Int. Conf. Rob. & Auto*, pp 843-849, 1987.

Can A. Baykan and Mark S. Fox

Constraint Satisfaction Techniques for Spatial Planning

Constraint Satisfaction Techniques for Spatial Planning

by

Can A. Baykan and Mark S. Fox

Intelligent Systems Laboratory
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA. 15213

10 March 1989

ABSTRACT

WRIGHT is a CAD system for designing two dimensional layouts consisting of rectangles. This problem arises in space planning, i.e. the design of floorplans, arrangement of equipment in rooms, and site planning. Space planning is a search process characterized by very large search spaces. Constraint directed search provides a basic problem solving methodology for intelligent CAD by providing a formal method for representing domain knowledge uniformly as constraints, and by using constraints for efficient search. Constraints reduce search complexity by opportunistically choosing the most constrained decision at each point. The knowledge that enables us to identify opportunistic decisions in space planning is formulated as a certainty measure associated with each constraint. Certainty depends on importance, reliance, cooperation and contention between constraints.

In WRIGHT, constraint propagation restricts the set of alternatives as information becomes available during search. Least commitment representations (value ranges) delay decisions until enough information becomes available. And abstractions reduce complexity by allowing abstract constraints to prune away entire design subsets.

Constraint Satisfaction Techniques for Spatial Planning

1. Introduction

Intelligent CAD requires a fundamental problem solving methodology which can incorporate arbitrary amounts of knowledge in a principled manner. Constraint-directed search provides a formal method for representing expertise uniformly as constraints. From constraints an understanding of the structure of the problem (search) space, that leads to more efficient search, can be derived [11, 8, 2]. Thus constraint-directed search addresses the needs of intelligent CAD by enabling the representation of knowledge from diverse sources and enabling the selection of efficient search strategies based on an understanding of search space structure.[1]

This basic problem solving methodology provides the framework within which the following issues can be addressed:

- knowledge representation,
- acquisition and maintenance of design expertise,
- user interface for graphical specification of constraints,
- user interface for interactive design.

WRIGHT is an intelligent interactive *space planning* system for generating two dimensional layouts consisting of rectangular shapes using constraint-directed opportunistic search. Space planning deals with the design of two dimensional layouts, such as floor plans, the arrangement of equipment in rooms and site planning. In space planning, topological relations and shape, dimension, distance and other functions of spatial arrangement are a principal concern. Almost all aspects of design have spatial implications, and influence space planning decisions.

2. Problem

WRIGHT deals with the generation of two-dimensional layouts consisting of configurations of rectangles. Inputs for generating a layout are:

[1]This differs from encapsulating expertise in the form of rules in that rules do not provide for an understanding of problem space structure, but simply identify situations of applicability without any guarantee that the search being performed is efficient.

- An existing layout which may be an empty space and dimensions of which may be specified as ranges, as seen in figure 2-1.

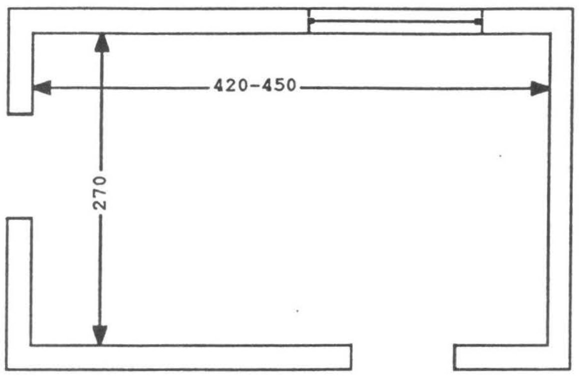


Figure 2-1: Plan of kitchen showing existing layout

- Design units to be located and/or dimensioned, as seen in figure 2-2.

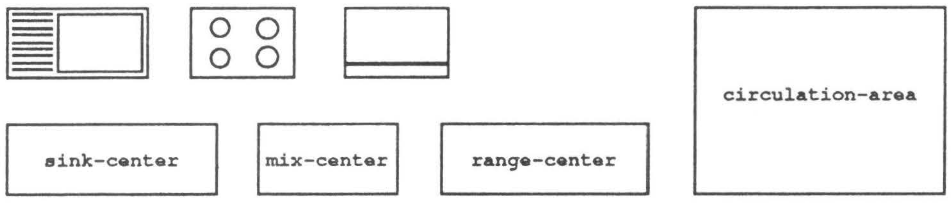
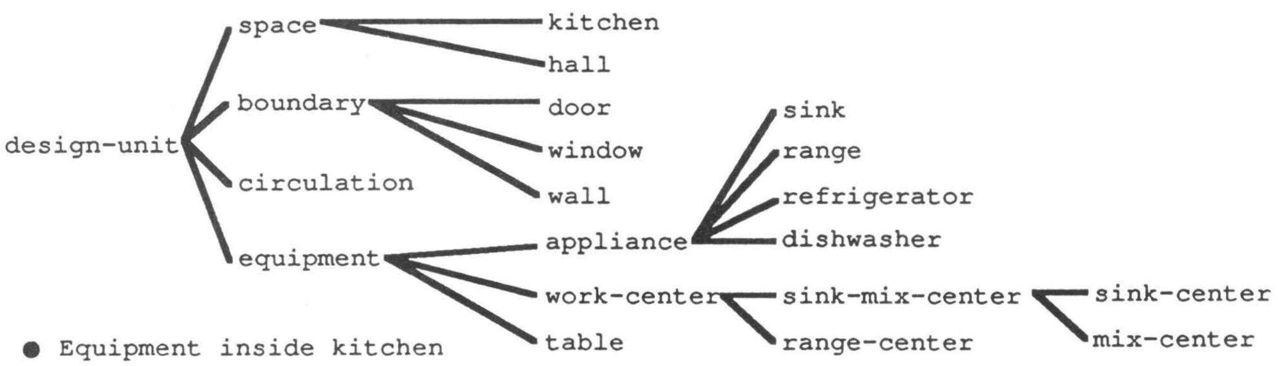


Figure 2-2: Design units to be dimensioned and located in kitchen

- Knowledge about the design domain in the form of a class hierarchy of prototype design units and constraints on them, as seen in figure 2-3.



```
● Equipment inside kitchen
● Appliance non-overlap appliance
● Appliance completely-next-to circulation
● Work-center next-to circulation
● Range-center next-to sink-mix-center
● Sink inside sink-center
● Sink next-to window
● Sink length ≥ 120 cm.
```

Figure 2-3: Class hierarchy of kitchen design units and some kitchen constraints

The output of WRIGHT is a set of pareto optimal layouts that are significantly different from each other, as seen in figure 2-4.

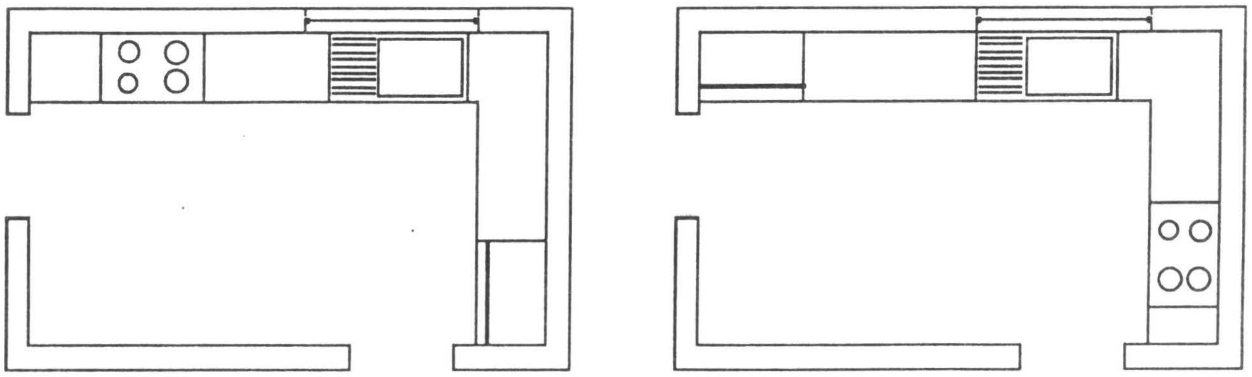


Figure 2-4: Two solutions to the kitchen design problem defined above.

3. Background

Based on their underlying representations, previous approaches to layout can be classified as *grid based*, *drawing based* and *relational*. Grid based representations partition objects to be located into subparts of equal area and divide the site into a grid of cells where each cell is equal in area to one subpart. Drawing based representations use polygons of fixed size and shape to represent objects. A polygon is represented as a set of sides, and a side as a set of points. Relational representations use adjacency or incidence between points, between lines and regions, or between regions to model layouts.

Search in space planning operates by selecting a spatial element(s) and an operator, and generates a new configuration by applying the operator to the element(s) in some state. Structuring these elements of search gives rise to different strategies. Starting search with an empty initial configuration results in a *build-up strategy*. There are two basic variations in a build-up strategy: *organize by element* and *priority* solution methods [4]. In an organize by element strategy, the next object to enter the layout is selected, placed at alternative locations and tested. All relevant attributes of the object are determined at the time it enters the configuration, and all applicable tests are carried out to select satisfactory locations. Search continues by selecting a new object to enter the design. Priority strategy orders search operators as in ABSTRIPS [17] and other hierarchical planning systems. Operators with high priority are applied first, creating macro objects or configurations in unbounded space by determining the important attributes first. In an *improvement strategy*, search starts from a configuration which contains all the elements. Changes are made in response to failing constraints or in order to improve the score of an objective function.

Quadratic assignment formulation [12, 13] uses a grid based representation and both build up and improvement strategies. The grid based representation can not deal with variable sizes, and makes it very hard to deal with issues of shape and alignment. DPS [16] and GSP [4] use drawing based representations. GSP objects must be rectangles and DPS objects can be arbitrary polygons. In both systems dimensions of objects must be fixed. GSP uses an organize by element strategy, whereas DPS can also utilize a priority strategy. In systems using drawing based representations, locations tried for placing an object depends on the existing layout, as seen in figure 3-1. As a result of this, configurations generated depends on the order in which objects enter design. Since GSP and DPS try only one ordering, they may miss existing solutions. Their correctness is not guaranteed.

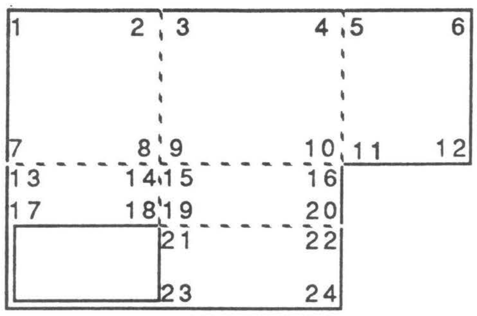


Figure 3-1: Locations considered by GSP for placing the next design unit, from [4;p.57]. Locations are defined by lines projected by the edges of the space and the objects that are in place. Placing an object at every location above, in four possible orientations, results in 96 new configurations.

GRAMPA [10], DIS [5] and LOOS [7] are space planners that use a relational representation. GRAMPA uses adjacencies between regions, and DIS and LOOS use the adjacencies between regions and lines. These systems systematically generate all distinct configurations based on their respective representations. They use a two step process that deals with relational and dimensional aspects separately, thus are not opportunistic in the use of constraints. The relations that are used for generating solutions are not based on the requirements of the problem, but on a restricted set of relations defined in each system.

4. Insights and Approach

Design is the process of constructing a description of an artifact that satisfies a functional specification, meets explicit or implicit performance criteria, is realizable and satisfies restrictions on the design process itself [14]. The artifact is initially defined by its desired properties, and it is natural to express them in terms of constraints. In WRIGHT, any knowledge that defines or restricts the domain is expressed as a constraint.

Space planing is a search process [3]. It is characterized by very large search spaces. Constraints play a major role in reducing search complexity, by opportunistically choosing the most constrained decision to make at each step [8]. Constraint propagation restricts the set of alternatives by eliminating values that are inconsistent with the current decision. WRIGHT uses constraints to select an efficient search path, and uses constraint propagation to ensure that values of all variables are consistent.

Design representations should tolerate ambiguity and incomplete specification. They are different from representations of existing situations. Least commitment [17] representations delay decisions about uncertain aspects, while making it possible to reason about the certain aspects of designs. Abstractions can further reduce complexity by allowing abstract constraints to prune entire design subsets [15]. Abstract constraints bound the solution space, and objects at different levels of aggregation simplify search in WRIGHT. Decisions are not forced by the representation used. When a commitment is made, it is always for satisfying a constraint. Attributes of design not specified by the selected constraint are deferred.

5. Representation

The elements of WRIGHT are design units (objects), relations and constraints. Knowledge is represented by a class hierarchy of prototype design units, and constraints specifying desired relations between design units or restrictions on their attributes such as length or orientation. *Abstraction by aggregation* combines design units into larger objects, such as combining a set of rooms into a house; or a sink, dishwasher, and counter-top area into a sink-center. WRIGHT can handle problems involving design-units at different levels of aggregation.

5.1. Objects

Layouts are represented by instances of design units having some relations and values assigned to them. Each design unit instance is a rectangular object and has an orientation that specifies its front. Each rectangle is a network-region which consists of a vertical interval, a horizontal interval and an area. Each interval is composed of two lines and a dimension. Figure 5-1 shows the objects and the relations that connect a structured object to its component objects and variables.

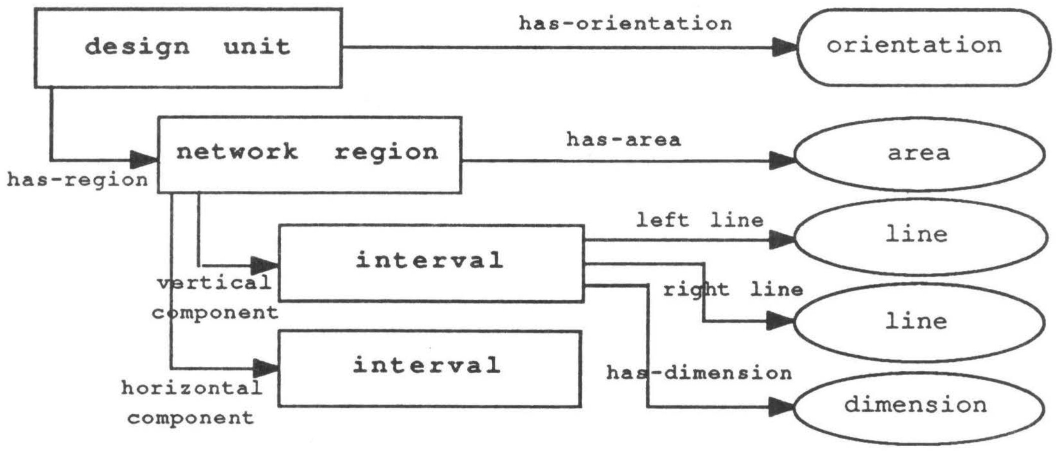


Figure 5-1: Structured objects and variables for representing rectangular design units

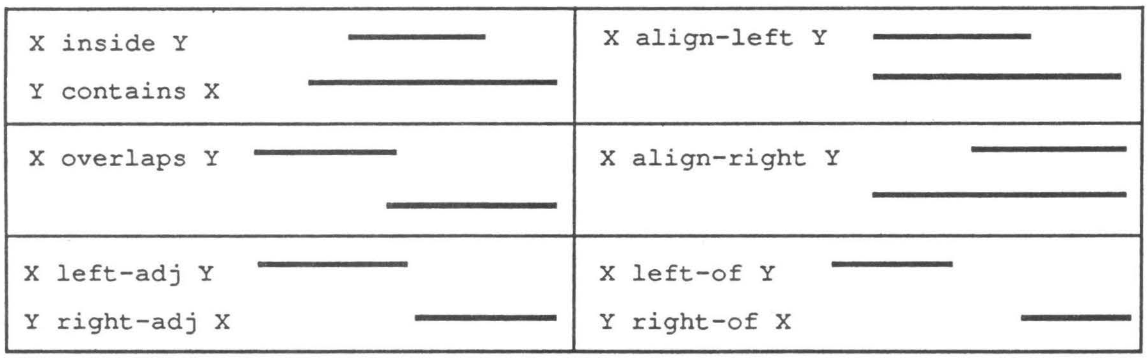
There are three types of objects: *structured objects*, *continuous variables* and *discrete variables*. Structured objects are composed of other objects or variables. Design unit, network region, and interval are structured objects. Line, dimension and area are continuous variables. Continuous variables have a range of values defined by a minimum and a maximum. For vertical lines, minimum and maximum values indicate the range of allowable x-coordinates of the line. For horizontal lines, they indicate values of the y-coordinates. Orientation is a discrete variable and may take a single value from the set *{0, 90, 180, 270}*.

5.2. Relations

Relations enable us to express the types of configurations that are of interest in space planning problems. There are relations defined for every type of object. Relations on structured objects are mapped into relations on the components of that object.

The set of relations on continuous variables are: >, ≥ and = . These may be between two variables as well as between a variable and a number. *Vertical-line1 > vertical-line2* means that vertical-line1 is to the east of vertical-line2, since the value assigned to a vertical line specifies its x-coordinate. The only relation defined on discrete variables is the = relation.

There are 9 relations defined between intervals seen in figure 5-2. Each box in the figure shows a relation and its inverse. Three of the relations are symmetric, i.e. their own inverses.



NOTE: X denotes the top interval, Y denotes the bottom interval

Figure 5-2: Relations between intervals

Relations between network regions are defined with respect to the global directions such as in location relations, or disregard directions such as in spatial or adjacency relations. Network region relations are:

- **Spatial:** inside, contains, overlaps, completely-overlapping, one-dimensional-overlap, non-overlapping
- **Adjacency:** next-to, completely-next-to, covers
- **Location:** north-of, south-of, east-of, west-of
- **Distance:** distance

Design unit relations are similar to network region relations, but take into account the orientations of one or both of the design units involved:

- **Orientation:** parallel-to, perpendicular-to, opposite
- **Relative location:** at-front, at-back, at-left, at-right, at-side
- **Relative alignment:** align-front, align-back, align-left, align-right
- **Relative Distance:** front-distance, back-distance, left-distance, right-distance

When required for expressing desired configurations, new relations on structured objects are created by defining them in terms of the >, ≥, = relations between lines. As a result of observing the problem solving behavior of WRIGHT, on kitchens, we have identified new spatial relations such as: completely-overlapping, one-dimensional-overlap, at-side in addition to the ones we have started with.

5.3. Constraints

Any knowledge that defines or restricts the domain is a constraint. Constraints indicate restrictions or desired relations. In WRIGHT constraints are of the form:

- *<object> <relation> <object>*, or
- *<variable> <algebraic relation> <number>.*

Constraints expressing domain knowledge are posted to prototype design units, as seen in figure 2-3. These are called *domain constraints*. When it is posted, each constraint is assigned an *importance* value specifying the importance of satisfying that constraint in a solution.

Design knowledge is expressed in terms of required spatial relations in WRIGHT. Consider the relationship of the sink to windows: "The average housekeeper spends nearly 1 and 1/4 hours at the sink each day so there is a good case for putting the sink at a window for good light and view." [1;p.72] One way of satisfying the requirements is placing the back of the sink completely next to the window. The following two constraints express the light and view requirements better:

- *Sink distance window, max=120 cm.*
- *Sink one-dimensional-overlap window, min=30 cm.*

Distance is measured between closest points. One dimensional overlap means overlap in either the vertical direction or horizontal direction.

Placing the sink also against a wall and completely next to the circulation area results in the range of locations seen in figure 5-3. In case there is more than one window, both of the constraints should hold between the sink and the same window. This is expressed by posting a *similarity* condition between the constraints. Also when there is more than one window, the constraints should be satisfied for one window only. Whether the constraint should apply to all windows or just one window is either posted to the constraint or inherited from the relation specified. There are defaults for relations, i.e. for non-overlap the default is *and* whereas for most other relations the default is *or*.

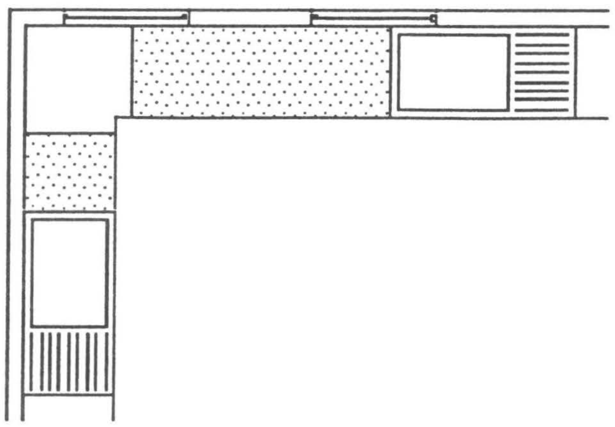


Figure 5-3: Range of locations satisfying the constraints between sink and window

Design knowledge is refined into a design specification by the constraint graph. A constraint graph maps domain constraints onto design unit instances and their component network regions, intervals and lines, as seen in figure 5-4. There is a grammar for specifying how to map a constraint expressing a relation between design units into constraints specifying relations between their components. All spatial relations are defined using this grammar. New relations between design units can be defined by expressing them as combinations of interval relations between their components. This is a language for expressing design knowledge.

The constraint graph is an and/or network that indicates *reliance* between constraints. In figure 5-4, sink1 and window1 are always drawn in the same orientation because orientations are unknown, not because orientations are fixed. The relation next-to is between network regions, therefore orientations of the design units are not considered.

All constraints are binary, so they are either satisfied or contradicted. Solutions are rated by subtracting the importance values of failing constraints. *Relaxations* of constraints are specified by other constraint expressions that specify alternative relations, alternative design units, or looser bounds on numerical variables. Relaxations are tried when a constraint can not be satisfied in its original form. States where a constraint is relaxed are assigned lower ratings as specified by the *utility* of the relaxation.

The constraint graph specifies alternative ways of satisfying a constraint. When there are conditions which hold true in all the alternatives, they can be used to bound solutions without committing to a specific alternative. These are called abstract constraints. Abstract constraints exist for constraints specifying adjacency and distance relations, and for dimensional constraints. Figure 5-5 shows the abstract constraints for the constraint graph seen in figure 5-4. The configuration in the figure shows the location defined by the abstract constraints for the sink when the location of the window is fixed.

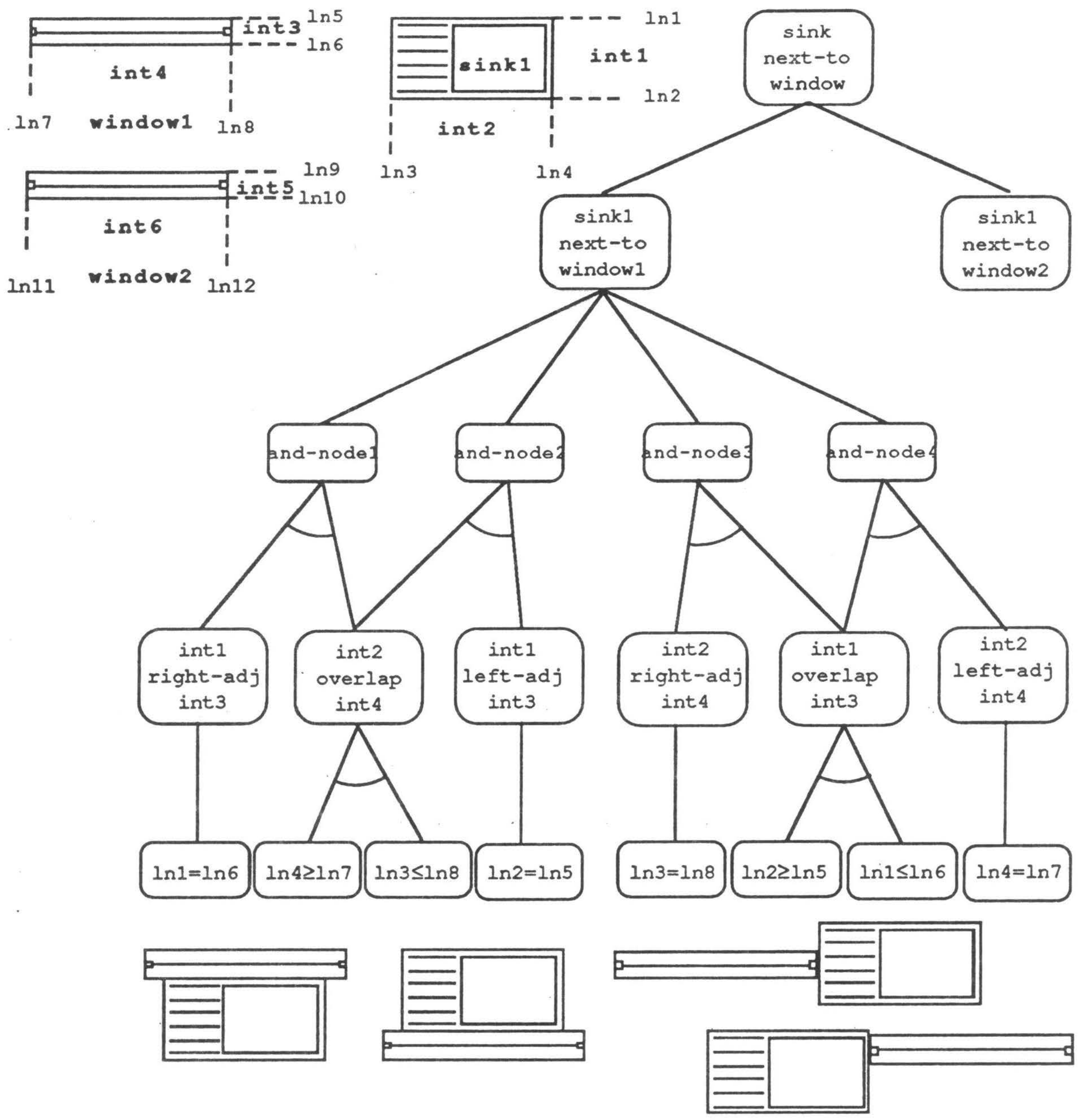


Figure 5-4: Partial constraint graph mapping the constraint *sink next-to window* to constraints on the components of sink1 and window1

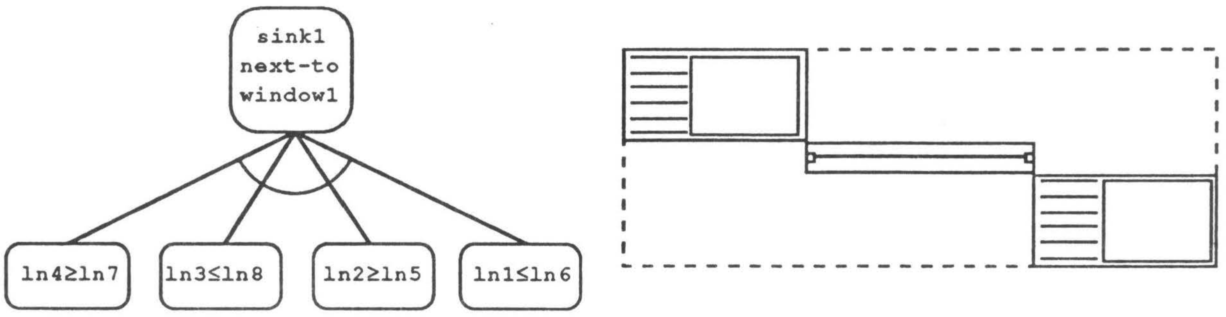


Figure 5-5: Abstract constraints for *sink1 next-to window1* and resulting bounds on location of sink

6. Search

The search architecture used in WRIGHT operates by selecting a constraint and satisfying it in ways specified in the constraint graph. This is a priority strategy where measures of certainty select the constraint to be satisfied. Operators are associated with constraints as in means ends analysis. The variables indicated by the constraint are assigned new values, constraints are propagated, and any constraints that have only one way of being satisfied are also satisfied. Dealing with variables that are not affected is deferred until later. The constraint graph provides a mesh over the problem space, and is used for generation and testing.

6.1. Search Architecture

Problem solving starts by compiling the constraint graph based on the givens of the problem: class hierarchy of design units and domain constraints, an existing layout, and design units to place. The constraint graph maps domain constraints that are posted to prototype objects to and/or combinations of constraints between instances and their components. Interactions between the newly created constraints on instances of structured objects and variables are found. Types of interactions are *contradicting* and *satisfies*.

Search is carried out by selecting a state, a constraint to satisfy, and generating new states that satisfy the constraint. The criteria for selecting a state is its rating and its closeness to a complete solution - a branch and bound strategy. Then uncertainties are assigned to constraints according to the four criteria: reliance, importance, looseness, and contention. The most certain constraint is selected and new states satisfying that constraint in different ways are created. Constraints that have no alternatives are identified from the constraint graph and satisfied. This may cause other constraints to have only one way of being satisfied. This loop continues until quiescence.

6.2. Measures of Opportunism

The measure of the structure of the search space which identifies opportunistic decisions is *certainty*. Certainty values are assigned to constraints based on reliance, importance, *looseness* and *contention*.

Importance of a constraint is used to rate solutions that violate the constraint. When used as a measure of problem space texture, importance selects constraints that assign a higher negative rating to a state when violated. Reliance indicates alternative ways of satisfying a constraint, which is the number of distinct alternatives specified as a result of the *or nodes* in the constraint graph of a constraint. Reliance measure favors a constraint that can be satisfied in fewer alternative ways. Looseness considers the range of values allowed by a constraint. It selects a constraint that permits a smaller range. Contention is a criterion of the number of disjunctive constraints that are competing to assign values to the same variable. Contention chooses a variable for which there are fewer competing constraints.

6.3. Search Operations

Each state contains a different layout and a different constraint graph, as seen in figure 6-1. When generating a state, the layout is changed by operators that satisfy a constraint. The operators carry out one or both of these actions:

- add new relations between objects,
- assign minimum or maximum values to line locations, dimensions, areas, and distances.

As a result of adding new relations or values, constraints are propagated to ensure that the values are globally consistent. Constraint propagation removes inconsistent values at continuous variables by increasing the minimums and decreasing the maximums. The constraint graph is changed by marking the nodes as *satisfied*, *contradicted* or *inactive*. These labels propagate through the constraint graph according to rules of propagation defined for and-nodes and or-nodes.

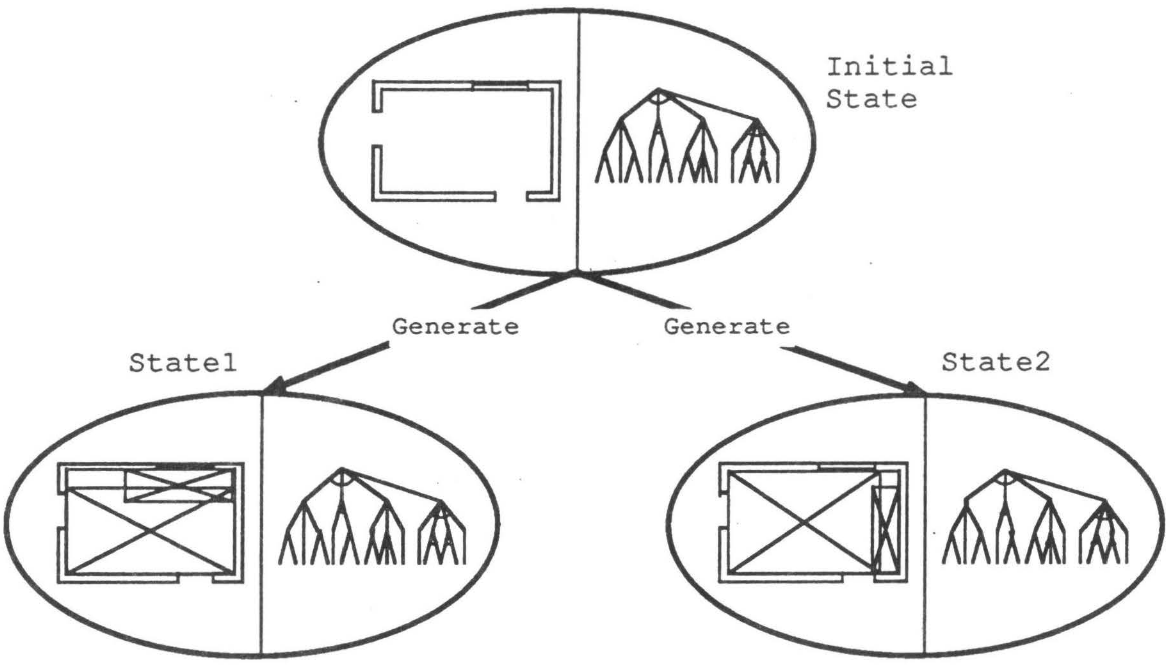


Figure 6-1: Search states

WRIGHT's representation and search operators are such that:

- Search is monotonic. States are generated by adding new relations or by tightening the bounds on numerical variables. A constraint that is satisfied can not be violated later.
- Satisfying a set of constraints in any order leads to the same set of solutions. Solutions are not dependent on the order constraints are applied, but the efficiency of finding solutions is.
- Alternatives specified in the constraint graph are mutually exclusive. Therefore it is not possible to get duplicate solutions.

Each solution satisfies a different subset of the constraints at the leaves of the constraint graph. Figure 6-2 shows the set of solutions for the kitchen seen in figure 2-1. The numbers show the order of generation of the state. Mix-center is the rectangle with the diagonals. The difference between

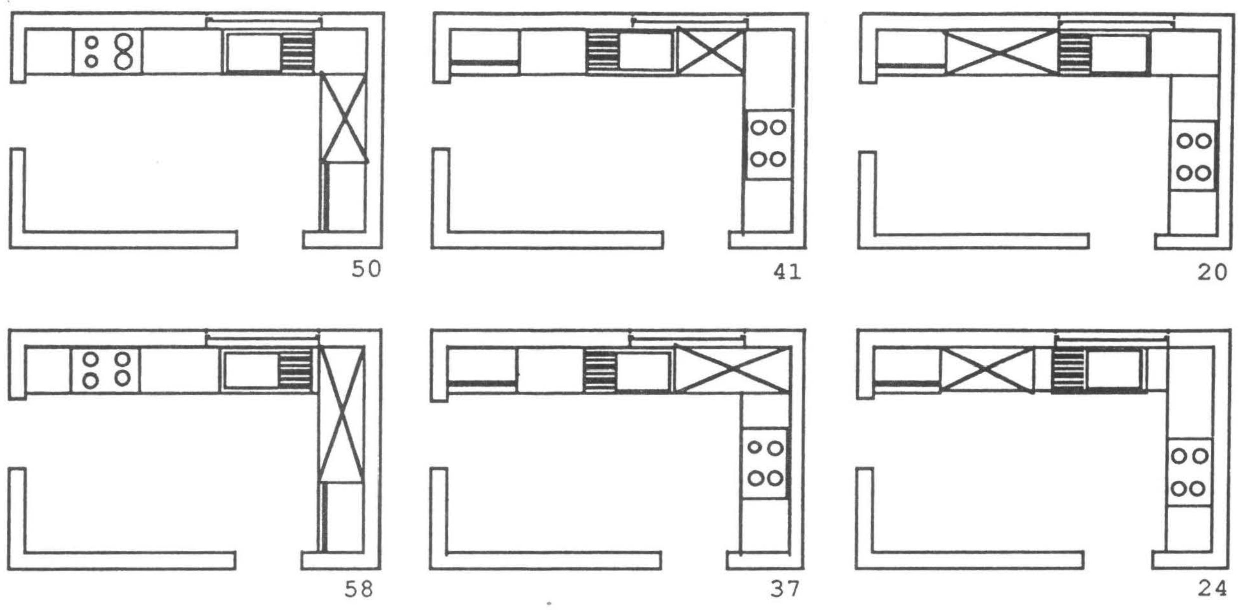


Figure 6-2: Complete set of solutions to kitchen problem

solutions 50 and 58 is that sink-center is placed next to the mix-center in different ways. The difference between solutions 24 and 55 is that in 24 refrigerator is next to the sink-center whereas in 58 it is next-to the mix-center. First difference is in instantiating the domain constraint, second one is in mapping a spatial relation to relations between intervals. WRIGHT uses constraints to define significant differences between alternatives.

The attributes of the layout that are not constrained are treated as unimportant. This permits solutions at a higher level of abstraction than in other space planning systems while at the same time enabling exact determination of relevant aspects. Unless the constraints force the assignment of a unique value, a variable has a range of values even in a solution. It is possible to attain different configurations by further refining a solution by assigning unique values to all variables.

6.4. Performance

WRIGHT looks at a smaller number of search states by selecting decisions with fewer alternatives, and by eliminating inferior alternatives earlier. Figure 6-3 shows comparison of WRIGHT with DPS

	First Solution	All 24 Solutions
WRIGHT	5 plies 14 states	5-6 plies 119 states
DPS	6 plies 72 states	*(not available)*
LOOS	6 plies 68 states	6 plies 232 states

Figure 6-3: Comparison of WRIGHT, DPS and LOOS in terms of search efficiency

[16] which uses a drawing based representation, and LOOS [6] which uses a relational represen-

tation[2]. The problem used in the comparison is arranging six fixed size blocks in a box such that no blocks overlap. Exactly the same set of constraints can be used by all three programs due to the simplicity of the problem. They are compared in terms of the number of states and search plies generated when finding the first solution and when finding all 24 solutions. In DPS and LOOS, the number of search plies is always equal to the number of objects to be located, as a result of the organize by design unit strategy. WRIGHT's performance in terms of number of search plies and number of search states depends on number and strength of available constraints and their interactions.

Another area of comparison is how search behavior changes when problems are under or over constrained. In an underconstrained problem, DPS and LOOS find the first solution faster, but there will be a large number of solutions. WRIGHT also finds the first solution more quickly, and will avoid generating a large number of solutions by having solutions at a higher level of abstraction. In an overconstrained problem, DPS will not be able find any solutions because it rejects a solution that fails any constraints. It is not possible to determine whether a solution that can be found by a different order of inserting the objects exists. For LOOS, overconstrained problems pose the same difficulty as underconstrained ones: too many states with equivalent scores. Finding the first solution will take much longer too. Overconstrained problems will cause WRIGHT to search longer before finding the first solution. When all constraints can be satisfied, solutions are defined by alternative ways of satisfying all constraints. When all constraints can not be satisfied, combinations of constraints that result in equal ratings need to be tried. By defining explicit relaxations for some domain constraints in its knowledge base, WRIGHT avoids searching a large number of constraint combinations.

7. User Interface

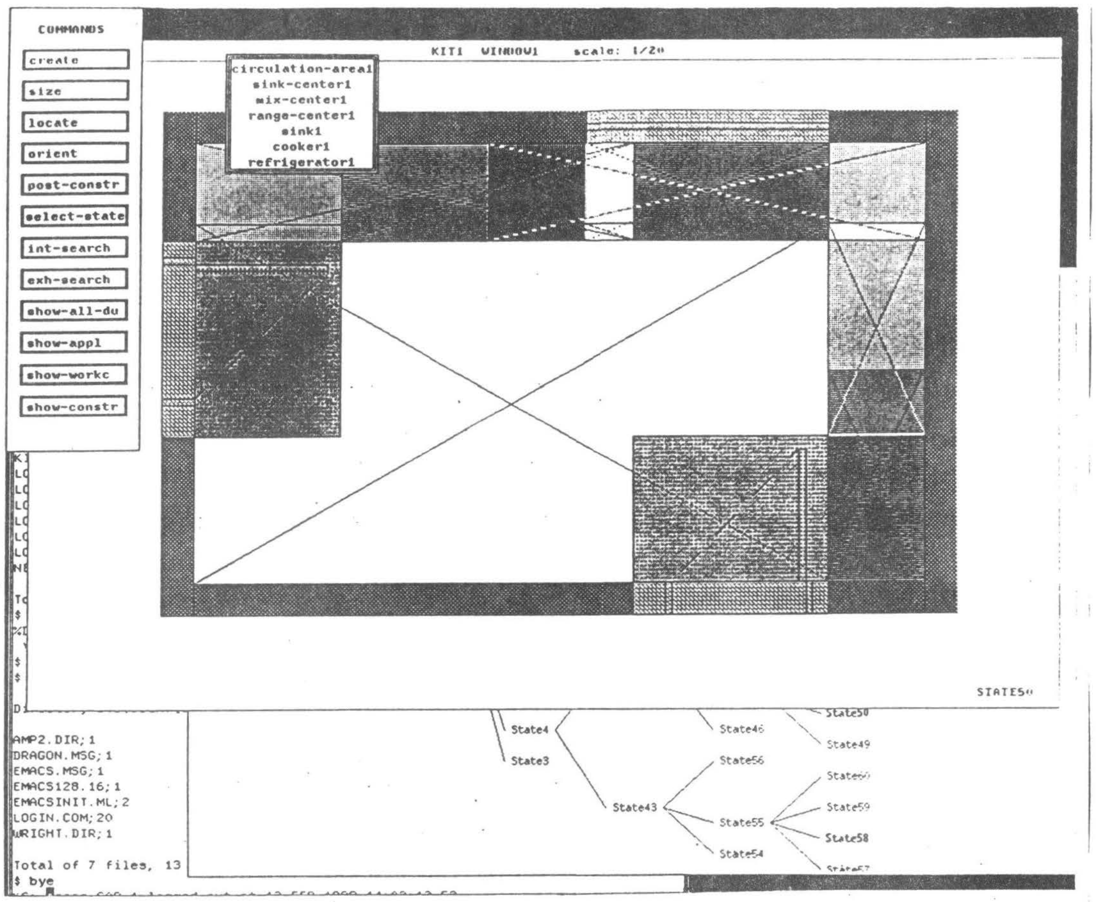
The user interface provides a menu-driven, graphical means for designing, posting constraints and carrying out search. The user interface is designed to make it easy to:

- observe search decisions and why that decision was picked,
- examine partial solutions,
- select which partial solution to pursue next,
- add missing constraints,
- relax overly restrictive constraints.

Changes, such as adding or removing a constraint, and bounding the location of a design unit take place in the current search state and affect states that are generated from it.

Figure 7-1 shows three windows: for graphics, for the search tree, and for text. There is a command menu in the top left corner. When the user selects a command by clicking on it with the

[2]see Section 3 Background for a discussion of these programs



```
AMP2.DIR;1
DRAGON.MSG;1
EMACS.MSG;1
EMACS128.16;1
EMACSINIT.ML;2
LOGIN.COM;20
WRIGHT.DIR;1

Total of 7 files, 13
$ bye
```

Figure 7-1: WRIGHT's user interface

mouse, menus pop up for selecting the parameters of the command. A pop-up menu for selecting design unit instances is seen in figure 7-1, superimposed on the plan.

There are commands to **create**, **size**, **locate**, and **orient** design units. The designer can define a rectangle by clicking at its top left and bottom right corners in the graphics window using the mouse. Such rectangles are used for inputting minimum size, maximum size and location of objects. It is possible to think of them as constraints, because they are bounds. Sizing and locating operations will not allow the user to violate existing bounds on the design unit. For that, one needs to move up the search tree to a state where those variables have looser bounds. Constraints specifying relations between design units at any level of the class hierarchy, including particular instances of design units, are posted by first selecting **post-constraint** from the command-menu, and then selecting a design unit, a relation and another design unit from pop-up menus.

The **select-state** command brings the search tree window in the bottom right corner of the screen to the top, and enables the user to select a state by pointing and clicking with the mouse. The configuration in the selected state is displayed in the graphics window, and following operations take place in that state. The next two commands, **interactive-search** and **exhaustive-search**, are for generating a set of new states from the current state and for carrying out exhaustive search.

The rest of the commands are for displaying information: displaying all design units - as seen in figure 7-1, displaying only the appliances, and displaying only the work centers. **Show-constraints** command displays the constraints that have not been satisfied yet, in the text window at the bottom left hand corner of the screen. It is also possible to inspect the database and call functions, as the text window is the lisp listener.

8. Conclusion

WRIGHT has been tested on kitchen design, house layout and blocks problems. The representation scheme used in WRIGHT is more flexible than other space planning systems in two respects. It allows design units at different levels of aggregation, and enables declaratively defining new relations. If, in some domain of application we identify new relations that express the conditions we are interested in, they can be defined as combinations of $>$, $\geq$, $=$ relations between lines and can be used in constraints.

The opportunistic constraint directed search approach of WRIGHT leads to significant increases in performance relative to other spatial planning techniques. This is due to two factors. First, WRIGHT uses knowledge of constraints to develop a mesh over the problem (search) space. Propagation of restrictions occurs within the mesh resulting in a reduction in size of the problem space. Second, WRIGHT opportunistically selects variables to instantiate once propagation ends. Knowledge of problem space "texture" is used to identify the appropriate variable. Examples of texture measures include reliance, which chooses to satisfy a constraint for which there are fewer alternative disjunctive decisions, importance which reduces backtracking by bringing up front the decisions that affect the outcome, looseness which chooses to satisfy a variable for which there are fewer values, and contention which chooses a variable for which there are fewer competing constraints [9].

In summary, the philosophy behind this approach is to use constraints to understand the structure of the search space to make search efficient. WRIGHT generates fewer alternatives as a result of selecting decisions opportunistically, avoiding premature commitment, and eliminating inferior alternatives earlier. Constraints guide generation of significantly different alternatives. Insignificant aspects of the design do not cause alternatives to be generated while relevant differences are explored in all possible ways.

References

1. Architects Journal. "Domestic kitchen design: conventional planning". *Architects Journal* (3 October 1984), 71-78.

2. Dechter R. and Pearl J. "Network-based heuristics for constraint-satisfaction problems". *AI 34*, 1 (1988), 1-38.

3. Eastman C.M. On the analysis of intuitive design processes. In *Emerging Methods in Environmental Design and Planning*, Moore, Gary T., Ed., MIT Press, Cambridge,Mass., 1970.

4. Eastman C.M. "Automated space planning". *AI 4* (1973), 41-64.

5. Flemming U. "Wall representations of rectangular dissections and their use in automated space allocation". *Environment and Planning B 5* (1978), 215-232.

6. Flemming U. On the representation and generation of loosely packed arrangements of rectangles. Tech. Rept. DRC-48-05-85, Carnegie-Mellon University Design Research Center, 1985.

7. Flemming U., Rychener M.D., Coyne R.F., Glavin T. A Generative expert system for the design of building layouts. Center for Art and Technology, CMU, Pittsburgh, PA., 1986.

8. Fox M.S. Observations on the role of constraints in problem solving. Proceedings Sixth Canadian Conference on Artificial Intelligence, May, 1986, pp. 172-187.

9. Fox M.S., Sadeh N., and Baykan C. Constrained heuristic search. Forthcoming in: Proceedings of IJCAI-89, 1989.

10. Grason J. *Methods for the computer-implemented solution of a class of floor plan design problems*. Ph.D. Th., Carnegie-Mellon University, May 1970.

11. Haralick R.M. and Elliott G.L. "Increasing tree search efficiency for constraint satisfaction problems". *AI 14* (1980), 263-313.

12. Koopmans J.C., Beckmann M.J. "Assignment problems and the location of economic activities". *Econometrica 25* (1957), 53-76.

13. Liggett R.S. "The quadratic assignment problem: an analysis of applications and solution strategies". *Environment and Planning B 7* (1980), 141-162.

14. Mostow J. "Toward better models of the design process". *AI Magazine 6*, 1 (1985), 44-57.

15. Newell A., Shaw J.C. & Simon H.A. The processes of creative thinking. In *Contemporary Approaches to Creative Thinking*, Gruber,H.E., Terrel,G. & Wertheimer,J., Ed., Atherton, 1962.

16. Pfeffercorn C. *Computer Design of Equipment Layouts Using the Design Problem Solver*. Ph.D. Th., Carnegie-Mellon University, May 1971.

17. Sacerdoti E.D. "Planning in a hierarchy of abstraction spaces". *AI 5* (1974), 115-135.

Farhad Arbab, Bin Wang

A Geometric Constraint Management System in Oar

A Geometric Constraint Management System in Oar

Farhad Arbab

Bin Wang

Computer Science Department
SAL 200, MC 0782
University of Southern California
Los Angeles, CA 90089
USA

ABSTRACT

This paper presents a constraint management system to experiment constraint-based design for mechanical engineering applications. The design is object-oriented and constraint-oriented. A design process can be viewed as incrementally creating design objects and imposing constraints on them. The system provides a sketch model and a constraint model which are loosely-coupled so that a designer can evolve a design by evolving both models separately. This system architecture is more appropriate for interactive design than existing systems. We use Operational Transformation Planning (OTP) as an approach to satisfy a network of geometric constraints [2]. The approach is different primarily in its use of geometric reasoning for satisfaction planning. An OTP system automatically generates a sequence of operations to satisfy a network of constraints, based on its understanding of the higher-level semantics of constraints and the geometric implications of the operations. Our system is implemented using Oar, an object-oriented programming environment with symbolic reasoning capabilities.

1. Introduction

The increasing demands for design automation present the challenging goal of adding "intelligence" to CAD systems, especially in the domain of computer-aided mechanical design where satisfying geometric constraints is a major concern. We expect an intelligent CAD system to be sufficiently knowledgeable in the domain of geometry to "understand" the implications of geometric constraints, and to find appropriate means to satisfy a set of constraints as a design evolves.

Constraint-based design systems have been developed in the past, some, specifically to deal with geometric constraints. Among them, the systems based on algebraic interpretation of geometric constraints seem dominant, e.g., the well-known variational geometry developed by [7, 12, 13, 14]. The algebraic approach translates dimensional constraints into a system of algebraic equations (non-linear equations generally) and finds an embedding of the geometry by solving the system of equations numerically. The algebraic approach sacrifices the more abstract semantics of geometric constraints for numerical computation, leading to problems including (i) large amount of computation is required when the number of equations and variables grows; (ii) the numeric iterative methods depend on *good* starting points to converge and there are no general rules to find them for non-linear equations; (iii) attempting to solve the equations may reveal the non-uniqueness of the resulting geometry and the system has no basis for choosing "the intended" one; (iv) consistency detection and maintenance in these systems are problematic since constraints are solved numerically and it is difficult for a designer to create and maintain a consistent set of constraints.

To avoid the problem of converting the constraints into a large system of equations, Rossignac [15] proposed an alternative approach based on the operational interpretation of constraints. Constraints are evaluated independently in a sequential manner and the sequence of evaluation is defined by the user. Two problems with this approach are (i) users are responsible for designing a sequence of operations to satisfy constraints and for resolving the conflicts in evaluation of the constraints, and, (ii) it is inadequate for solving problems formulated by several simultaneous constraints for which no simple sequential process is available.

As it was observed in [1] , the analytic view of geometric constraints is not sufficient for high level applications like mechanical design. We also need to use the axiomatic view of geometry explicitly to represet and reason about geometric constraints at an abstract level. We are using an approach, called Operational Transformation Planning (OTP), which is based on the high level understanding of the semantics of constraints and the geometric implications of operations. The main ideas are (i) constraint satisfaction and maintenance can be carried out in terms of operations and (ii) a network of constraints can be satisfied by planning a sequence of operations through geometric reasoning.

Oar is an object oriented programming system where general declarative information about objects and their interrelationships dynamically determines their imperative behavior [4]. In Oar, logic programs determine what objects receive what messages and what procedures they use to respond to them. A procedure itself is a sequence of message passing imperatives that request the services of other objects. Oar programs can exercise or relinquish explicit control during the course of their computation in proportions suitable for the application at hand. Oar is especially suitable for highly dynamic applications such as design, where objects and their behavior must evolve throughout the course of a computation, as more information about them and their interrelationship is assimilated by a program. The dynamic, context-dependent inheritance of methods by objects and the symbolic reasoning capability supported by Oar make it a suitable environment for OTP.

The rest of the paper is organized as follows. Section 2 is a brief review of some related work. Section 3 is an overview of a prototype design system based on OTP from designer's point of view. The system architecture is introduced in Section 4. The principles of the OTP approach are presented in section 5. In Section 6, an implementation scheme in Oar is presented. A case study is discussed in Section 7. Section 8 summarizes some of the important characteristics of the system. Finally, Section 9 concludes the paper.

2. Related Work

In 1962, Ivan Sutherland built the first constraint-based drawing system SKETCHPAD [18] that allowed the definition of arbitrary objects and constraints. It pioneered the use of interactive computer graphics and constraint systems. Alan Borning's THINGLAB system [5] carried on where SKETCHPAD left off. THINGLAB is a generalized simulation laboratory based on constraints. Users sketch a design and tell THINGLAB what the parts are and how they behave, and THINGLAB performs a simulation.

Variational geometry [7, 12, 13, 14] is a typical algebraic approach that represents and satisfies geometric constraints algebraically. The central idea of their effort is that dimensions are a natural descriptor of geometry and provide the most appropriate means for altering a geometric model. In variational geometry, a geometric model is defined with respect to a set of characteristic points in space. Dimensions are treated as constraints limiting the permissible locations of these characteristic points. Each dimensional constraint is described by a nonlinear equation involving the coordinates of associated characteristic points. A given dimensioning scheme is represented by a set of such equations. The geometry corresponding to an altered dimension is found through the simultaneous solution of the set of constraint equations. The Newton-Raphson method is used to simultaneously solve the constraining equations which yields the geometry corresponding to the set of dimensions. Some techniques are used to identify invalid dimensioning schemes and to improve the efficiency of solving equations.

Bruderlin et al [6] developed a system for automatically building 3-D geometric objects that are defined by their topology and by geometric constraints. A 3-D solid modeller providing set operations on polyhedra is used as a graphic editor for sketching spatial objects and thus defining their topology. Sketched objects are dimensioned (constrained) by selecting points and entering values. The geometric constraints are first evaluated symbolically by a constraint satisfaction mechanism written in the logic programming language Prolog. The result of the symbolic evaluation is a set of construction steps. In the second step, the symbolic solution is then numerically evaluated by procedures written in the procedural programming language Modula-2.

Kimuara and Suzuki [8,9] presented a uniform framework for product modelling where geometric constraints capture a wide range of geometric information, such as dimensions, tolerances and assembly. They proposed a method that combines first order predicate logic and an object oriented approach to deal with geometric constraints. In their framework, they use solid models for storing the product's shape description and logical formulae for representing geometric constraints on the elements of the solid models. Dimensions and tolerances are represented explicitly as logical formulae. Geometric reasoning (logic inferencing) is a process which manipulates these formulae to generate new formulae and to find out a solution satisfying a set of constraints.

Arbab and Wing [1] explore how axiomatic knowledge of geometry can be incorporated into geometric modellers to enhance the semantics of representation and manipulation of geometric information. They consider geometric problem solving as a process that involves the axiomatic view of geometry as well as its analytic view. A programming environment, Oar (Object and Reasoning) [3] was developed, which combines logic programming and object oriented programming to accommodate geometric reasoning.

Rossignac [16,15] proposed an approach to facilitate specification and editing parameterized models of solids through a user-friendly interactive graphical front-end in systems based on a dual representation. The main idea is to specify positions and dimensions of primitives in terms of geometric constraints but not through parameters of rigid motions. In his approach, a user specifies a binary CSG graph and a sequence of motions that operate on collections of primitives. The rigid motions are stored in terms of unevaluated constraints on graphically selected boundary features, the constraints are evaluated sequentially in a user-specified order.

Recently, Serrano and Gossard presented a constraint-based environment MCAE, for Mechanical Computer Aided Design [17]. Constraint networks are represented as directed graphs, where nodes represent parameters and arcs represent constraint relationships. They developed a bipartite matching representation to generate the parameter dependencies automatically. They use algorithms based on the constraint network topology to evaluate constraint networks, detect over- and under-constraint systems, and identify and correct redundant and conflicting constraints.

3. System Environment

We are building a prototype constraint-based design system for the domain of two-dimensional geometry using OTP. As the front-end of this system, a graphical interface is built that provides users with means to interact with the system. The interface provides a sketch pad and a list of command icons. A user can experiment his design by picking the command icons and position points on the sketch to create objects, modify objects or impose constraints.

Command icons are organized hierarchically. Two main classes of icons are *to create objects* and *to impose constraint* (abbreviated as commands *CREATE* and *CONSTRAIN*). *CREATE* consists of a list of commands to create an instance of various classes of objects recognized by the system. When *CREATE* is picked, its list of command icons is displayed at the bottom of the sketch pad. A user can create an instance of any object by clicking the icon of its corresponding object type and picking the position points for the instance. A user is required to enter a character string as the name of the object created which is used to refer to this object later. For example, a user can create a line by clicking *LINE* and picking its two end-points and entering l_1 as its name. When this is done, a line is displayed in the sketch (as in Figure 1). An instance of a circle can be constructed similarly by picking its center and a point on its circumference. The construction process is guided by the system by prompting for user input. A user can end the creation phase and returns to the higher command level by clicking the QUIT icon. Additional commands are provided by *CREATE*, such as to *delete* an object and to *read in* objects previously created in a file.

The second class of icons, under *CONSTRAIN*, contains a list of commands to impose a constraint of various types recognized by the system. When *CONSTRAIN* is clicked, the list of command icons is displayed at the bottom of the sketch pad. A user can impose a constraint of any type by clicking its corresponding icon and picking the constrained objects in the sketch. A user is required to enter a name for the constraint which will be used for reference either in satisfying a constraint or maintaining a constraint. For example, to impose the constraint l_1 *join* l_2, a user can click icon *JOIN*, pick l_1 and l_2 on the sketch and enter con_1 as the constraint name. When done, l_1 and l_2 join at their end-points (as in Figure 2). A user can end the constraint phase and return to the higher command level by clicking the QUIT icon.

Another important command *MOVE* is used to *move* existing objects around the sketch pad. *MOVE* can be interpreted in a generic sense as a combined translation and rotation involving the positions of objects only. To move an object, a user simply clicks the *MOVE* icon and picks an object in the sketch, then picks a point where this object is moved to. When done, the object is redisplayed in the new position. The execution of the *MOVE* may result in violation of satisfied constraints, therefore some other changes may be made by the system to maintain the constraints. The details are discussed in later sections.

A simple example is demonstrated below which involves two lines l_1, l_2 and a circle c_1. The constraints are: l_1 joins l_2, l_1 and l_2 are both tangent to c_1. Firstly, l_1, l_2 and c_1 are created separately as in Figure 1. Click *CONSTRAIN* for constraint commands, and click *JOIN* to impose the constraint l_1 *join* l_2. As in Figure 2, l_2 is moved (by the system) to join l_1. Click *TANGENT* to impose constraint l_1 *tangent* c_1, as in Figure 3, c_1 is moved (by the system) to be tangent to l_1. Imposing the third constraint l_2 *tangent* c_1, instead of moving c_1 to satisfy the constraint, results in the system rotating l_2 around the joint to become tangent to c_1 as in Figure 4. This is done through reasoning about the context of the satisfied constraints.

4. System Architecture

The system consists of three major components: the user interface, the sketch model, and the constraint model (Figure 5). Internally, the three components are objects and communicate with each other through message passing. The sketch model is an internal representation of the geometric information displayed on the screen. The constraint model is an abstract representation of the constraints imposed by a user. A user can directly manipulate both models. Unlike most other constraint systems, the constraint model is not necessarily *complete* enough to imply a unique solution. Specifically, the loose binding between the two models means that the sketch model is always only an instance of a solution for the constraint model. Thus, adding a constraint may result in a modification to the sketch model, while deleting a constraint causes no change to the sketch model. Similarly, modifications to the sketch model may or may not entail further changes to the sketch in order to keep the sketch model an instance of the constraint model.

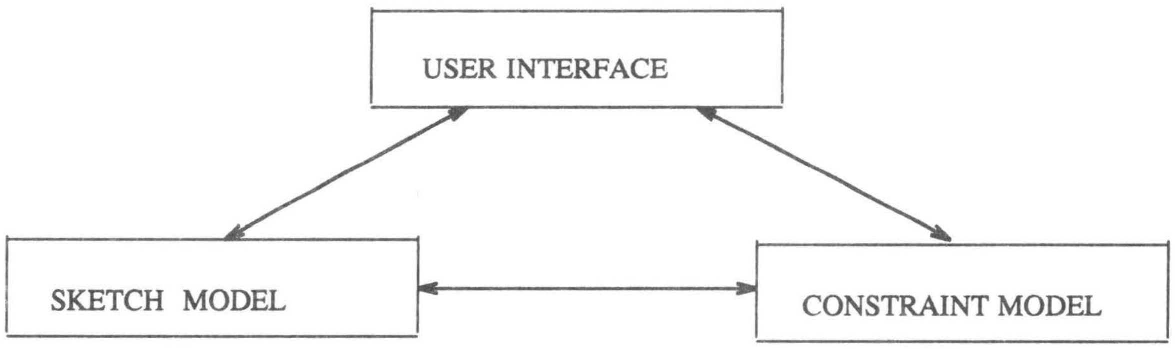


Figure 5

User Interface

The user interface is a graphical window with command panes by means of which a user can interact with the system. Basically, the user interface acts as a command interpreter which interprets users' input and passes proper messages to either the sketch model or the constraint model. Users can change the state of the system only through the sketch window and command panes. Simultaneously, in the other direction, the user interface also interprets system's response (messages from the sketch model or the constraint model) on the sketch.

Sketch Model

The sketch model is essentially a system object that manages the dimensioned representations of objects in the sketch window. It has methods corresponding to messages for creating objects and modifying objects. It deals with individual object instances in terms of their dimensional values in the sketch. It can also communicate with the constraint model via message passing when changes are made to the sketch or when a constraint is imposed. The sketch model may request the user interface to redisplay objects whose dimensional values are changed.

Constraint Model

The constraint model is conceptually a system object that maintains the symbolic information about the objects in the sketch. It represents the properties of the objects and their relationships (constraints) symbolically, and has an inference mechanism to reason about this information. The constraint model has methods to satisfy constraints and to maintain (previously satisfied) constraints. The constraint model responds to messages from the sketch model by invoking its methods to satisfy constraints or to maintain constraints. Geometric reasoning is used to choose the proper methods.

5. Operational Transformation Planning Approach

We use operational transformation planning (OTP) as an approach to satisfy a network of geometric constraints incrementally. The main ideas are as follows. A constraint is satisfied by performing a sequence of operations specifically designed for this constraint. A satisfied constraint can either tolerate or propagate an operation performed on its participant objects. Satisfying a constraint in the context of a network of satisfied constraints is to perform the operations associated with the constraint and to propagate the changes through the network in term of operations. The satisfaction process is planned through symbolic reasoning. A set of constraints is satisfied incrementally.

5.1. The Computational Model

The operational transformation planning model is a 5-tuple $<\mathcal{O},\mathcal{C},\mathcal{P},\mathcal{R},\mathcal{I}>$. $\mathcal{O}$ is a finite set of *object types* that contains all object types of interest. An object type actually defines a class of *object instances.* An *object instance* of an object type (abbreviated as *object*) is a concrete entity with dimensional values.

$\mathcal{C}$ is a finite set of *constraint types* that specify various relationships among participating object types. A *constraint type* $c_i(o_1, o_2, ..., o_r)$ consists of two parts: constraint name c_i and a list of participating object types $<o_1, o_2, ..., o_r>$. A *constraint type* defines a class of *constraint instances.* A *constraint instance* (or *constraint*) is a constraint type where its participating object types are replaced correspondingly with object instances. These object instances are called *participants* in the constraint.

$\mathcal{P}$ is a finite set of *operation types.* An *operation type* $p_j(o_1, o_2, ..., o_s, \phi_1, \phi_2, ..., \phi_t)$ consists of three parts: operation name p_j, a list of object types $<o_1, o_2, ..., o_s>$ and a list of parameters $<\phi_1, \phi_2, ..., \phi_t>$. An *operation type* defines a class of *operation instances* (abbreviated as *operation*). An *operation* is an operation type where its participating object types are replaced correspondingly with object instances. Operations are applied to object instances in order to (1) satisfy a new constraint imposed on them, and (2) maintain the already satisfied constraints they participate in. An operation can be as simple as a primitive translation or a rotation, or it may be some complex procedure for finding the clipped version of a polygon.

$\mathcal{R}$ is the constraint propagation function used to maintain the satisfied constraints in a network after a change. A satisfied constraint $c_i(a_1, a_2, ..., a_k, ..., a_r)$ receiving an operation $p_j(b_1, b_2, ..., b_s, \phi_1, \phi_2,$

..., ϕ_l) from its kth participant, fires a set of operations defined by $\mathcal{R}(c_i, k, p_j)$. This function returns alternative sets of pairs $\{<p_l, n_l>\}$. A set of pairs $\{<p_l, n_l>\}$ indicates that in order for the constraint c_i to *tolerate* operation p_j on its kth participant, operation p_l must be performed on its n_lth participant for each pair in the set.

$\mathcal{I}$ is the impose function used to impose a new constraint c_i on a set of objects that may already participate in a network of satisfied constraints. The function $\mathcal{I}(c_i)$ returns alternative sets of pairs $\{<p_l, n_l>\}$. A set of pairs $\{<p_l, n_l>\}$ indicates that to satisfy c_i, operation p_l must be applied to its n_lth participant for each pair in the set.

5.2. Satisfaction Planning

Constraint satisfaction in the OTP approach is actually carried out in two stages: (1) planning a sequence of operations and (2) executing the operations. A set of constraints is satisfied incrementally by repeating the satisfaction process.

At the planning stage, when a new constraint c_i is added, it sends the operations prescribed by $\mathcal{I}(c_i)$ to its participants. Objects that are participants in other (satisfied) constraints in the network, notify them of the proposed operations. A constraint receiving an operation from one of its participants may send some other operations to its other participants as prescribed by its $\mathcal{R}$ function. A participant is *isolated* if it participates in only one constraint. An isolated participant can *absorb* any operation that maintains the constraint. A constraint *absorbs* an operation from a participant if the function $\mathcal{R}$ returns an empty set of operations. If this propagation of operations in the network terminates in a finite number of steps, the resulting sequence of operations is a plan for satisfying the new constraint in the context of the previously satisfied constraints. The second stage is the execution of the plan. Specific procedures are associated with the entries in the $\mathcal{R}$ and $\mathcal{I}$ tables to carry out the operations.

Geometric knowledge and the context information of constraints can be incorporated in the planning stage by means of geometric reasoning. There are two cases when alternative schemes of imposing or propagating operations exist and we should take the best choice. We can also take advantage of the degrees of freedom in choosing parameters of operations, using the context of satisfied constraints and geometric knowledge. For example, to satisfy a constraint l_1 *parallel* l_2, we can rotate l_1 by a certain angle θ around any pivot, but if the context is that l_1 *tangent* to another circle c_1, we would like to rotate l_1 around the center of c_1, so that we preserve tangency. As another example, to make l_1 tangent to circle c_1, in the context l_1 *intersecting* l_2, we would prefer to rotate l_1 around the intersection point rather than to translate l_1.

5.3. Propagation Loops

The propagation of operations during a satisfaction planning process may result in a loop which indicates the constraints involved cannot be dealt with in isolation. Global relaxation of the whole constraint network as used in THINGLAB [5] is a general way to solve the loop problem but it is inefficient. We present a solution using local relaxation which solves only those constraints involved in a loop and propagates the solution to the rest of the constraint network. A simple example below illustrates the loop problem.

The Pentagon Problem

Given five segments s_1, s_2, s_3, s_4, s_5 with fixed lengths, construct a pentagon on a plane. A simple OTP model for this problem is given below.

$\mathcal{O}$ = { *segment,point* }.
$\mathcal{C}$ = { *joint(segment,segment,point,point)* }.
$\mathcal{P}$ = { *link(segment,segment,point,point)* }.
$\mathcal{R}(joint(S_1,S_2,P_{e1},P_{s2}),link(S_3,S_1,P_{e3},P_{s1})) = \{link(S_1,S_2,P_{e1},P_{s2})\}$.
$\mathcal{I}(joint(S_1,S_2,P_{e1},P_{s2})) = \{link(S_1,S_2,P_{e1},P_{s2})\}$.

The constraint $joint(S_1,S_2,P_{e1},P_{s2})$ means that the starting point P_{s2} of segment S_2 coincides with the end point P_{e1} of segment S_1.

The operation *link*$(S_1, S_2, P_{e1}, P_{s2})$ moves segment S_2 such that its starting point P_{s2} coincides with the end point P_{e1} of segment S_1.

The pentagon can be constructed by a set of five constraints:

{*joint*$(s_1, s_2, p_{e1}, p_{s2})$, *joint*$(s_2, s_3, p_{e2}, p_{s3})$, *joint*$(s_3, s_4, p_{e3}, p_{s4})$, *joint*$(s_4, s_5, p_{e4}, p_{s5})$, *joint*$(s_5, s_1, p_{e5}, p_{s1})$}.

The first four constraints can be satisfied using the OTP approach without any problems. The fifth constraint *joint* $(s_5, s_1, p_{e5}, p_{s1})$ which tries to close the pentagon results in a loop. The impose function $\mathcal{I}$(*joint*$(s_5, s_1, p_{e5}, p_{s1})$) returns {*link*$(s_5, s_1, p_{e5}, p_{s1})$}. As in Figure 6, the movement of s_1 involves the movement of s_2, and the movement of s_2 involves the movement of s_3, etc., and finally, the movement of s_5 involves the movement of s_1: the propagation of the *link* operation forms a loop.

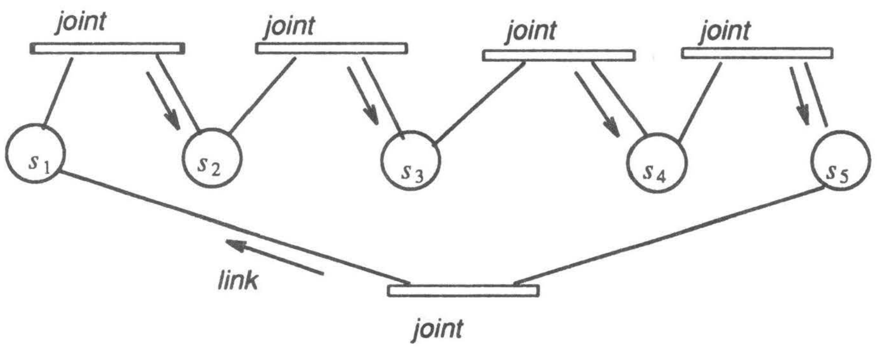


Figure 6

Local Relaxation

Local relaxation handles a set of constraints which cannot be satisfied in isolation. The main steps are i) detect a propagation loop in the current propagation phase; ii) translate the constraints involved in the loop into a system of algebraic equations; iii) pick up the additional dimensional equations from the sketch model that are necessary to make the set of equations solvable; iv) solve the system of equations numerically; v) assemble a sequence of operations according to the numerical solution, so that when the operations are carried out, the constraints in the loop are satisfied simultaneously; and finally, vi) propagate the assembled operations to the rest of network. We briefly explain the above steps through an example.

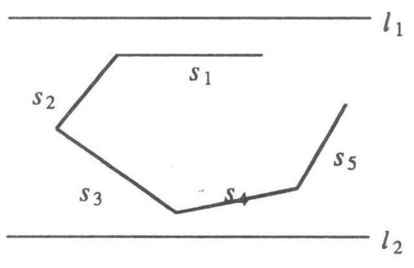


Figure 7

Consider the line segments in Figure 7. The underlying constraints are l_1 *parallel* l_2, l_1 *parallel* s_1 and four *joint* constraints that connect the five segments s_1, s_2, s_3, s_4 and s_5. To close the five segments, we impose a new constraint s_1 *joint* s_5 which results in a loop.

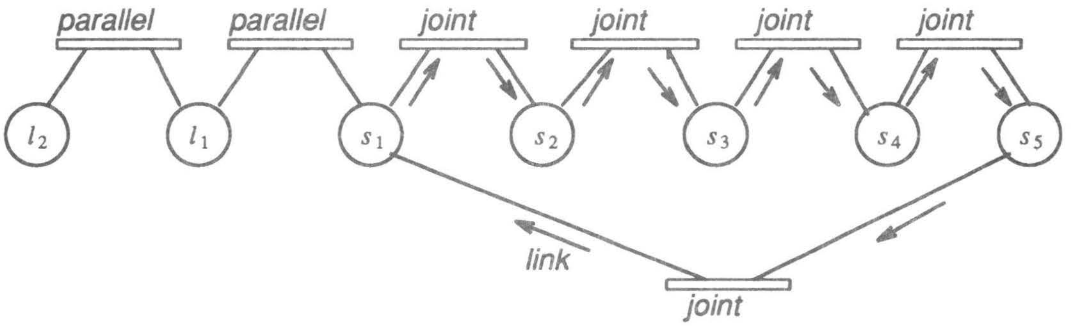


Figure 8

1. Detection of Propagation Loops

   The propagation of operations originates from satisfying a constraint. A constraint sends out operations to its participants either to satisfy or maintain itself. An object involved in the constraint receives the operation and propagates to other constraints it participates in. The underlying assumption is that the objects receiving an operation have not received any other operations during the current satisfaction planning process. If an object receives a second operation, it contradicts our assumption of operation. We call it a propagation loop.

   In this example, a loop is detected when s_1 receives the second *link* operation. The rest of the task is to find all the constraints involved in the loop. This is done by backtracking the propagation route. In the example, the five *joint* constraints are collected.

2. Translating Constraints into Equations

   The set of constraints (and objects) obtained from the above step is translated into a system of algebraic equations. The equations are in terms of coordinates of characteristic points or characteristics parameters of the participating objects. For example, constraints about lines are translated into equations in terms of coordinates of their end-points.

   l_1: (x_{11},y_{11}), (x_{12},y_{12})
   l_2: (x_{21},y_{21}), (x_{22},y_{22})

   joint$(l_1,l_2) \rightarrow x_{21} = x_{12},\ y_{21} = y_{12}$
   parallel$(l_1,l_2) \rightarrow (y_{12}-y_{11})(x_{22}-x_{21}) = (y_{22}-y_{21})(x_{12}-x_{11})$

   An initial set of equations are obtained from the constraint model. The resulting system of equations is usually under-constrained, because the constraint model generally does not specify the positions and orientations of all objects. We need to pick up additional constraints from the sketch model to make the system of equations solvable. This gives us some freedom in choosing additional dimensional constraints from the sketch model. For instance, we simply *fix* some objects to obtain a well-constrained set of equations. We can apply heuristic rules to select an appropriate set of additional constraints to 1) localize the effect of the new constraint, and 2) simplify equation solving. In this example, we would like to fix s_1 that maintains the parallelism between l_1 and s_1.

3. Solving the System of Equations

   The system of equations is solved using numerical relaxation methods. We use a method to represent and manipulate equations similar to the one used in variational geometry. The solution gives the coordinates of the characteristic points or parameters of the objects involved in the loop.

4. Assemble Operations From the Solutions

   The solutions of equations give the proper dimensional values of the objects that satisfy the constraints in the loop. Objects whose original positions are different from the solutions must be moved to their new positions by some operations. The effect of such operations may have to be propagated to the objects outside of the loop. So we assemble the operations according to the solutions and propagate the operations to the rest of the network. The assembling of operations depends on the context of the rest of network. When alternatives exist, we choose the operation which affects the rest of network less than others.

5. Propagate Operations to the Network

   Every one of the operations may propagate to the rest of the network. In this example, l_1 *parallel* s_1 is maintained because s_1 is fixed in loop relaxation and no further propagation is required. Otherwise, the operations on s_1 must be propagated to l_1 through *parallel* to maintain the constraint l_1 *parallel* s_1.

6. Constraint Management

6.1. An Overview of Oar

Oar is a system for object-oriented programming that combines the power of logic programming for expressing declarative facts about objects and their interrelationships, with the concept of message passing as the mechanism for triggering the imperative knowledge associated with objects [4]. In Oar, general declarative information about objects and their interrelationships determine an object's behavior. The details of manifestation of such behavior are expressed imperatively through message passing. The inheritance network in Oar is dynamic and fluid. The class membership of an object, and thus its behavior, can dynamically change as more facts about its properties and its relationships with other objects are introduced. Oar is suitable for highly dynamic applications where objects and their behavior must evolve through the course of a computation, as more information about them and their interrelationship is assimilated by a program.

Computation in Oar is carried out by two major components: an inference mechanism based on first order logic, and a imperative programming environment based on message passing. The two components are closely interrelated through the notion of objects. Message passing is the means by which objects use the computational resources of other objects. Message passing in Oar is associative and uses the inference mechanism to identify the receivers of messages and their methods. The inference mechanism itself is an object whose resources can be used by other objects through message passing. Some of the interesting features of Oar are presented below.

1. Combining declarative and imperative knowledge

   In most object oriented systems, declarative knowledge can be specified for only one purpose. This is to reflect a limited view of how objects are interrelated through a class/instance hierarchy. As in logic programming languages, in Oar, declarative knowledge of a general nature can be expressed. Such information can affect selection and activation of methods of objects.

2. Predicates as object properties

   In a language like Smalltalk, an object is a concrete entity whose structure is determined (and fixed) at the time it is instantiated, by its position in a predetermined class hierarchy. Thus, structural properties of objects, i.e., their attributes, are induced by this static class hierarchy, although their values may change dynamically. As in logic programming languages, there is no such distinction between structure and value in Oar. In Oar, an object is an abstract entity that satisfies a given set of properties. Properties of an object in Oar are the set of facts (well-formed formulae in a formal system based on first order logic) known about that object. The properties of an object include any well-formed formula that can be derived by the inference rules of the logical system, as well as the ones stated explicitly, i.e., as axioms.

3. Dynamic object hierarchy

A distinguished relationship among objects in Oar is the *isa* relation. This relation defines an object as a specialization of another. An *isa* relation between two objects can be defined as the logical consequence of some other facts. Thus, in Oar the equivalent of a class hierarchy is induced by the properties of objects. Because properties of objects can dynamically change as more facts are discovered during program execution, the object/class hierarchy in Oar is dynamic.

4. Prototype objects

   In Smalltalk, there is a distinction between an object and a class. The underlying philosophy of this distinction is the set representation of a class [10]. A class is a set of instances where each instance is an object. Classes can have subclasses. Subclass definition and creating an instance are two distinct operations in this approach. In Oar, a class is represented by one of its typical members, i.e. by an object. This approach is based on the representation of classes as *prototypes* [10]. In this approach, any object represents the class of which it is a typical member. Defining a subclass and creating an instance are both accomplished by the same mechanism: defining extension objects through the *isa* predicate.

5. Inheritance of properties

   Inheritance in Oar is associative. Propagation of properties through the object/class hierarchy (Oar's equivalent of inheritance in Smalltalk) is accomplished through a special inference rule: the *isa* inference rule (a different concept than the *isa* relation mentioned above). The combination of this inference rule and Oar's method definitions make message passing in Oar effectively similar to the delegation mechanism of actor-based systems [11].

6. Associative binding of methods with objects

   Associative inheritance of method properties leads to a flexible and powerful generalization of the binding of imperative knowledge with objects. In most object oriented programming languages, the binding of methods with objects is static and by programmers' decree. In Oar, this binding is associative and dynamic. For instance, It is possible to define a method as applicable to an abstract (class of) object(s) whose properties satisfy the preconditions of the procedure. Thus, any object whose properties satisfy the preconditions of such a method, will dynamically inherit the method. Furthermore, this ensures that the method is never invoked by an object that does not satisfy its preconditions.

7. Associative message passing

   Message passing in Oar is a generalization of message passing in conventional object oriented languages. To send a message, an Oar object specifies a message and a qualifier. The informal semantics of message passing in Oar is that the sender activates the specified method of all objects in the closure of the given qualifier. This generalizes the conventional message passing in two ways: first, there can be more than one receiver for a message, and second, the sender does not necessarily directly know the receivers beforehand.

Associative inheritance, dynamic object hierarchy, and associative message passing create alternative courses of action. When the message passing mechanism finds more than one receiver for a message, or more than one method for a receiver, they are taken as representing alternative courses of action. One of these alternatives can be selected nondeterministically, or they all can be pursued in parallel in a manner that is transparent to the program, through *branching*. The user program can also take control and sequentially send a message to all receivers as in a broadcast. For example, a user program may want to send a message to all objects whose color is red and let them print their names sequentially.

6.2. Mapping An OTP Model Into Oar

Object Types

Objects of an OTP model are Oar objects represented as object symbols. In Oar, there is no distinction between an object type and an instance of the type. An object type is a prototype object of that type.

The object types of an OTP model are represented as a set of object symbols reserved for the types of objects recognized by the system. An instance of a type can be declared using the **isa** predicate. For example, *line* is an object symbol representing object type line, a specific line l_1 can be declared as:

l_1 **isa** *line.*

Through the **isa** relation, l_1 can inherit all properties of *line*.

Constraint Types

Constraint types are also Oar objects, therefore represented as object symbols as well. Same as object types, there is no distinction between a constraint type and an instance of the type. A constraint type is a prototype constraint of that type.

The constraint types of an OTP model are represented as a set of object symbols reserved for the constraint types recognized by the system. An instance of a type can be declared using the **isa** predicate. For example, *parallel* is an object symbol representing constraint type *parallel*, a specific constraint p_1 can be declared as:

p_1 **isa** *parallel.*

Operations

An operation of an OTP model is implemented as a method procedure and is denoted by an object symbol which is the (message) selector for the method procedure. Operations are always associated with object types as their methods. The association is specified by the **method** predicate. For example, the association between the object type *line* and its method *rotate* which is implemented as the method procedure *line_rotate_c* can be specified as:

method(*line*,*rotate*,2,*line_rot_c*).

This definition states that when the object *line*, or any of its specializations that inherits this definition, receives a message whose selector is *rotate* and contains 2 parameters (a pivot and an angle), the method procedure *line_rot_c* is to be invoked to respond to the message.

An object type can have several method procedures for the same message selector. Selection of a specific method procedure to respond to a message can depend on the context. For example,

```
method(line,rotate,2,line_rotl_c):-
                                    tangent(line,circle).
```

states that when a *line* is tangent to a *circle*, the method procedure *line_rot*1*_c* is to be invoked to respond to a message whose selector is rotate and has 2 parameters. The procedure *line_rotl_c* is implemented to rotate a line around a circle and *line_rot_c* rotates the line around its start point. The selection of a method procedure can be based on by preconditions required by the method clause. For instance, *line_rotl_c* requires the line to be tangent to a circle. Association of the method procedures is determined by the Oar's inference mechanism dynamically at run time.

The general format for an operation method procedure is as follows:

```
procedure_name(client,selector,par₁,par₂,...,parₙ)
{
        check_operation_loop;
        perform_operation;
        propagate_operation;
}
```

Impose Function

The impose function is distributed to each constraint type as methods to satisfy the constraints. In our prototype implementation, the common selector for these methods is *satisfy*. Each constraint type has a method procedure that implements the impose function on this constraint type. The alternatives returned by impose function are implemented as different method procedures which are selected depending on the context. For example, the definitions

```
method(tangent, satisfy, 2, tangent_sat1_c):-
        constrained_by(Line,tangent), intersection(Line,Line1).

method(tangent, satisfy, 2, tangent_sat2_c).
```

states that *tangent_sat1_c* is invoked to satisfy an instance of *tangent*, if the line constrained by (the specific instance of) *tangent* is already constrained by an *intersection* constraint. This procedure rotates the line around the intersection such that the line is tangent to the circle. Otherwise *tangent_sat2_c* is invoked which simply translates the line to be tangent to the circle. The association of the method procedures and objects is dynamic at run time.

The general format of a method procedure for the method *satisfy* is:

```
procedure_name(client,selector,par_1,par_2,...,par_n)
{
        prove_by_theory; /* geometric reasoning */
        numerical_checking;
        parameter_computation;
        operation_sending;
}
```

Propagation Function

The propagation function is distributed to each constraint type as methods to maintain the constraints. In our prototype implementation, the common selector is *maintain.* that implements the propagation function on this constraint type. The alternatives returned by the propagation function are implemented as different method procedures specifying the context-dependent conditions for their selection. For example, the definitions

```
method(tangent, maintain, 2, tangent_mai1_c):-
        constrained_by(Line,tangent), intersection(Line,Line1).

method(tangent, maintain, 2, tangent_mai2_c).
```

states that *tangent_mai1_c* is invoked to maintain an instance of *tangent*, if the line constrained by *tangent* is also constrained by another *intersection* constraint. For example, to maintain the constraint under an operation that translates the circle away, the procedure rotates the line around the intersection to keep the line tangent to the circle. Otherwise *tangent_mait2_c* is selected which simply translates the line to be tangent to the circle.

The general format of a method procedure for method *maintain* is as follows:

```
procedure_name(client,selector,par_1,par_2,...,par_n)
{
        parameter_computation;
        operation_sending;
}
```

6.3. Geometric Reasoning

Geometric reasoning can be used in OTP to make the process of planning for constraint satisfaction more intelligent. As we observed in previous sections, the satisfaction planning process is rather local and simple-minded. More intelligent plans can be generated by considering the context of a constraint and its participating objects.

For example, suppose line l_1 is tangent to circle c_1. Imposing the constraint l_1 *parallel* l_2, can result in performing a rotation on l_1. Normally, rotation of a line would use its start point as the pivot. We can associate a different rotation method procedure with l_1 to use the center of the circle as the pivot for the rotation, so that the tangency is preserved.

> **method**(*line, rotate, 2, object_smart_rotate*):-
> *constrained_by(line,tangent).*
>
> **method**(*line, rotate, 2, object_normal_rotate*).

The context sensitivity intends to short-cut the operation propagation so that the possibilities of operation loops are reduced. In general, we try to localize the modification of the satisfied network and therefore reduce the amount of computation.

The explicit representation of geometric knowledge makes the reasoning about geometric constraints possible. The following example shows that implied constraints are satisfied without operational transformation.

If l_1 *parallel* l_2 and l_2 *parallel* l_3 are two satisfied constraints, imposing l_3 *parallel* l_1 is redundant, the system can prove this is an implied constraint and does nothing to satisfy it. The geometric knowledge used can be represented as:

> **method**(*parallel,satisfy,2,do_nothing*):-
> *constrained_by(L1,parallel),*
> *constrained_by(L2,parallel),*
> *parallel(L1,L3),*
> *parallel(L2,L3).*

Implied constraints always cause operation loops if they are not detected. Similarly, some contradictory constraints can be detected. Imposing l_1 *perpendicular* l_2 is a contradiction if we can show l_1 and l_2 are *not* perpendicular, for instance, show l_1 and l_2 are parallel.

> **method**(*perpendicular,satisfy,2,report_contradiction*):-
> *constrained_by(L1,perpendicular),*
> *constrained_by(L2,perpendicular),*
> *not_perpendicular(L1,L2).*

6.4. System Dynamics

1. *Constraint Satisfaction*:

   Constraint satisfaction is started by sending a message to a constraint object to invoke its *satisfy* method. The specific *satisfy* method procedure is dynamically selected by the system considering the context of the constraint. It sends proper messages to the constrained objects to invoke their operation methods. The operation procedures are also selected in a context dependent manner and change the objects properly and propagate the operations to other related constraints by invoking their *maintain* methods. The satisfaction process is done when the propagation finishes successfully or when a loop is detected.

2. *Loop Detection and Handling*:

   A special system object called *history* is responsible for detecting propagation loops. Same as other objects, *history* has methods that handle various cases. Two major methods are *record* which records an operation to be propagated into the history fact base and *loop_detect* which checks the fact base to see if a loop occurs.

When an operation method procedure of an object is invoked, it first sends a message to *history* to invoke its *loop_detect* method. If no loop is detected, the operation procedure performs the operation and sends a *recod* message to *history*. When a loop is detected, the loop handling process is invoked and the operation propagation stops.

Various specialized techniques can be used in the loop handler. In its most general form, the loop handler translates the constraints involved in a loop into a system of algebraic equations. Additional constraints can be picked up from the sketch model as needed to make the system of equations solvable. Using the solutions of the system of equations, the loop handler assembles a sequence of operations and sends them to the objects involved in the loop.

7. A Case Study

The following is a scenario that shows how a design problem is solved on our constraint system. The illustration is divided into steps that reflect the interactions between a designer and the system.

The Design Problem

Given two segments L, $L1$ with lengths l, l_1 and a circle C with radius r, design a mechanism which satisfies the following constraints:

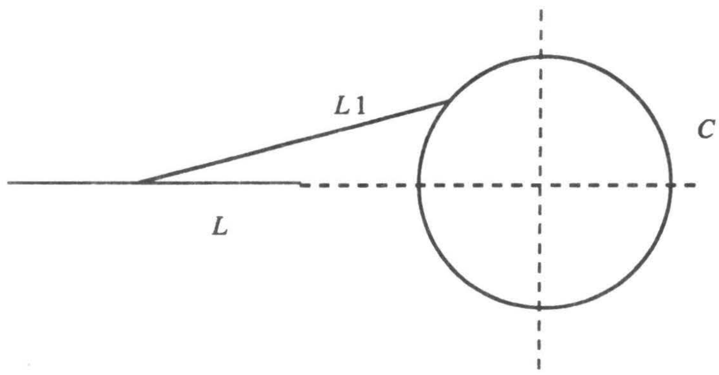
(1) one end-point of $L1$ is on L;

(2) the other end-point of $L1$ is on C;

(3) the center of C is on the extension of L;

(4) the end-point of $L1$ on C can move around the circumference of C while sliding the other end-point on L and maintaining the length of $L1$.

The solution to this problem involves finding an appropriate position for L relative to C.



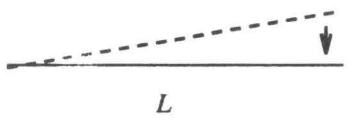
Denote the start point of a line L as $L_{startpoint}$, the end point of the line as $L_{endpoint}$ and extension of the line as $L_{extension}$. Denote the center of a circle C as C_{center}.

DESIGNER:

create an arbitrary line named L.
impose constraints: length(L) = l, horizontal(L).

SYSTEM:

scale L to length l.
the impose function for horizontal(L) returns:
rotate(L, $L_{startpoint}$, ϕ)
meaning to rotate L around its starting point by a proper angle ϕ such that L is horizontal.

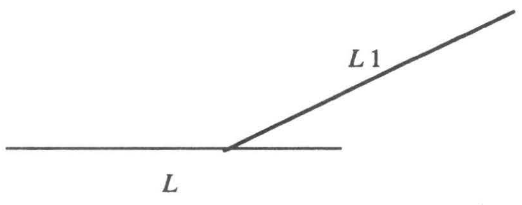


DESIGNER:

create another line named $L1$.
impose constraints: length($L1$) = l_1, coincide($L1_{startpoint}$, L).

SYSTEM:

scale $L1$ to length l_1.
the impose function for coincide($L1_{startpoint}$, L) returns:
translate($L1$, d_x, d_y)
meaning to translate $L1$ by a proper distance (d_x, d_y) such that its starting point is on L.

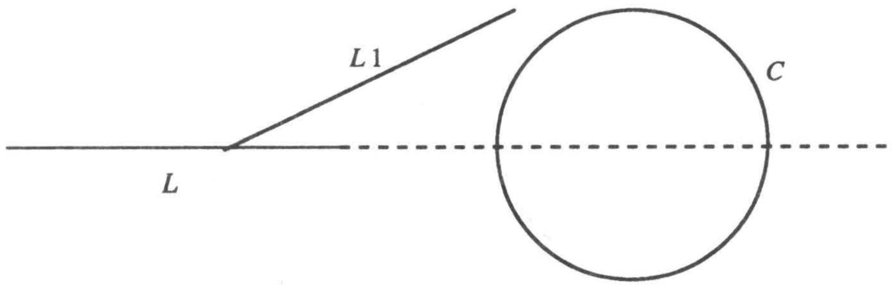


DESIGNER:

create a circle named C.
impose constraints: radius(C) = r, coincide(C_{center}, $L_{extension}$).

SYSTEM:

scale the radius of C to r.
the impose function for coincide(C_{center}, $L_{extension}$) returns:
translate(C, d_x, d_y)
meaning to translate C by a proper distance (d_x, d_y) such that the center of C is on the extension of L.



DESIGNER:

impose constraint: coincide($L1_{endpoint}$,C).

SYSTEM:

in the context of the constraint: coincide($L1_{startpoint}$,L),
the impose function for coincide($L1_{endpoint}$,C) returns:

```
if distance($C_{center}$,$L1_{startpoint}$)-radius($C$) $\leq$ length($L1$)
    rotate($L1$,$L1_{startpoint}$,$\phi$)
else
    translate($L1$,$L$,$d_{\min}$)&rotate($L1$,$L1_{startpoint}$,$\phi$)
```

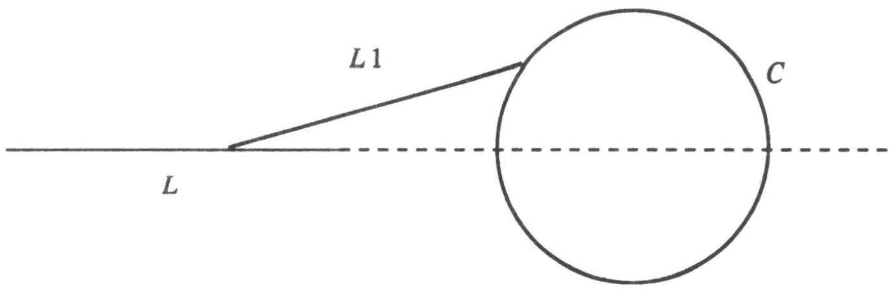
translate($L1$,L,$d_{\min}$) means translate $L1$ along L
by the minimum required distance.

for the first case, rotate($L1$,$L1_{startpoint}$,ϕ) is sent to the constraint coincide($L1_{startpoint}$,L) which decides it can tolerate this operation and no further propagation is required.

for the second case, translate($L1$,L,$d_{\min}$)&rotate($L1$,$L1_{startpoint}$,ϕ) are sent to the constraint coincide($L1_{startpoint}$,L).

depending on the amount of translation $d_{\min}$, either both operations are tolerated by this constraint, or a translate(L,L,$d_{\min}$) is propagated to L.

Assume it is the first case:



DESIGNER:

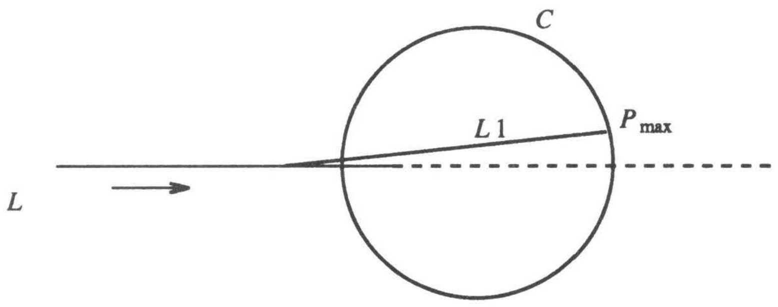
pick up a point $P_{\max}$ on C.
impose constraint: coincide($L1_{endpoint}, P_{\max}$)

SYSTEM:

in the context of the constraint: coincide($L1_{startpoint}, L$),
the impose function for coincide($L1_{endpoint}, P_{\max}$) returns:
translate($L1, L, d$)&rotate($L1, L1_{startpoint}, \phi$)

translate($L1, L, d$)&rotate($L1, L1_{startpoint}, \phi$) are sent to the constraint coincide($L1_{startpoint}, L$).

depending on the amount of translation d, either both operations are tolerated by the constraint, or translate($L, L, d_{\min}$) is propagated to L, where $d_{\min}$ is the minimum distance required for L to move in order to keep coincide($L1_{startpoint}, L$) satisfied.

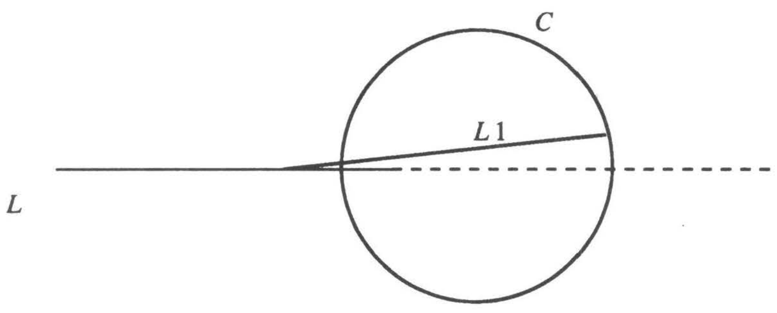


DESIGNER:

remove constraint: coincide($L1_{endpoint}, P_{\max}$).

SYSTEM:

remove coincide($L1_{endpoint}, P_{\max}$) from constraint set,
nothing is changed in the sketch.

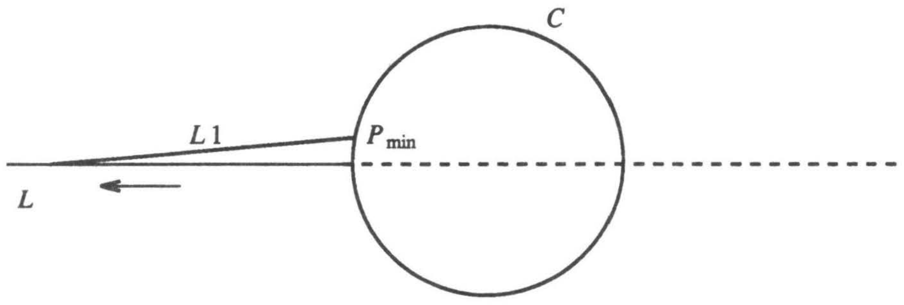


DESIGNER:

pick up another point $P_{\min}$ on C
impose constraint: coincide($L1_{endpoint}$,$P_{\min}$)

SYSTEM:

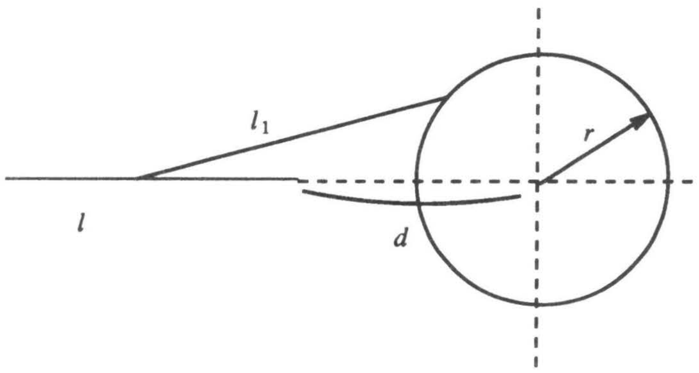
similar to imposing coincide($L1_{endpoint}$,$P_{\max}$).



We mention a few interesting observations about our case study:

- A designer can deal with incomplete information when working on a design. He may even finish a design without complete information to imply a unique solution. This capability is usually missing in other constraint-based systems. The systems based on the algebraic interpretation of constraints cannot solve this problem because some underlying constraints are only implicitly specified. For example, the following two constraints are necessary to imply a valid embedding:

$$l_1 \geq d + r$$
$$l_1 \leq d - r + l$$



- Qualitative constraints and those involving inequalities can be more easily satisfied if a designer can interact with the system. For example, pick a center for C *on* the extension of L, place a circle *above* a line, put a circle *inside* a box, etc.
- Reasoning about spatial relationships symbolically simplifies the computations and avoids problems with numerical methods.

8. Discussions

Some of the important characteristics of our constraint management system are summarized below.

- Two Tiered Model

  Incompleteness of information is inherent in design. Frequently, not enough information is available at intermediate stages of a design to derive a system of algebraic equations with unique solutions. Defaults and arbitrary fabrication of additional equations are commonly used to derive a unique solution in many systems. Coercing completeness in this manner has the drawback that once such additional constraints are added, they are often treated indistinguishably from the ''real'' constraints.

  In OTP, there is a clear separation between the intended constraints and the additional information necessary to render one of the possibly many solutions that satisfy those constraints. The system uses two models in parallel: a constraint model and a sketch model. The constraint model is almost always incomplete, reflecting only those constraints explicitly required by a designer. The sketch model is always complete, reflecting in the form of a concrete sketch, both the implications of the constraint model and all additional information necessary to drive a solution.

  In most systems, there is a tight correlation between the sketch and the set of constraints. We believe that while sketches are important tools for conveying relevant design information, not everything contained in a sketch should be taken seriously. Typically, most metric information and some topological relationships implied by sketches prove to be either wrong or irrelevant as a design evolves. Our design system assumes a looser binding between constraints and sketches than most other constraint based systems: sketches and constraints may be modified directly and independently. Changing a sketch or its underlying set of constraints may or may not entail a change to the other.

- Incremental Satisfaction

  Refinement of ideas during the course of a design often results in incremental addition of constraints. It is conceptually simpler to ignore existing solutions, start with the modified set of constraints, and try to find a solution to satisfy them. This approach suffers from three drawbacks. First, there is extra work involved in resatisfying the bulk of the constraints that remain intact after each incremental modification. Second, due to instability of some numerical methods and limitations of equation solving techniques, it may not be possible to solve the old equations in the presense of the new ones. Third, the solution found for the new set of constraints may not be related to the one for the old set of constraints in an intuitively meaningful way.

  In OTP, incremental addition of a constraint results in local modifications to the existing solution (sketch model). This can be more stable and computationally less expensive compared to constraint satisfaction systems based on algebraic techniques. The advantages become significant especially in the context of interactive CAD, where incremental growth and editing of a design's underlying network of constraints is predominant.

- Geometric Reasoning

  Most constraint systems are primarily based on the algebraic interpretation of geometric constraints, and lack the ability to reason about the abstract properties of constraints and design objects. In OTP, the higher-level semantics of geometric constraints are used at a planning stage to derive the local changes necessary for satisfaction of a constraint.

  The semantics of a satisfied constraint is expressed through the degrees of freedom that it leaves for its participating objects. The degrees of freedom of an object under a constraint are represented as a set of constraint-preserving operations. A new constraint is accommodated by a network of satisfied constraints through propagating the sequence of operations associated with the new constraint to the appropriate previously-satisfied constraints. Similarly, the semantics of satisfying a constraint is also expressed through a set of operations.

Constraint satisfaction in OTP uses symbolic reasoning to plan a sequence of operations that transforms an unsatisfied set of constraints to a satisfied set of constraints. It can use the higher-level semantics of geometric relations to avoid unnecessary computation. For example, knowing that tangency between a line and a circle is invariant under rotation around the center of the circle, an OTP system can avoid using the algebraic interpretation of tangency in appropriate contexts.

In systems based on the algebraic interpretations of constraints, sometimes the transitive closure of equations involving some given variables is used to localize solutions. This technique is purely syntactical and disregards the semantics of the context of constraints. As a worst case scenario, this technique may end up solving the whole set of constraint equations, whereas our approach can localize the changes (and equation solving) as shown in the following example.

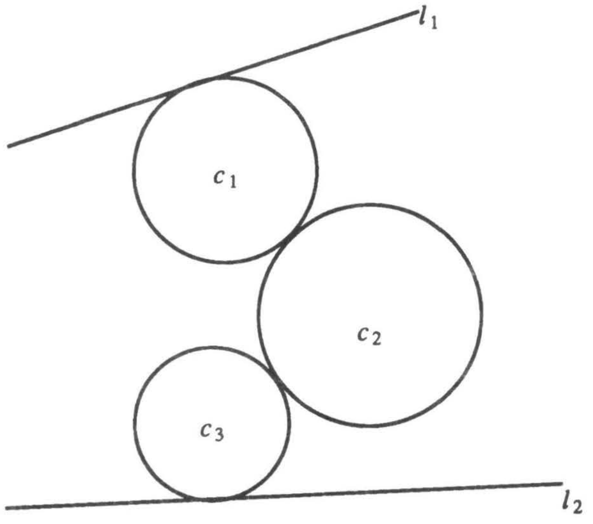


Figure 9

In the example of Figure 9, the constraints are l_1 *tangent* c_1, c_1 *tangent* c_2 $c2_1$ *tangent* c_3 and l_2 *tangent* c_3. To impose the constraint l_1 *parallel* l_2, the algebraic approach will solve the equations corresponding to the five constraints. Our approach will consider only a proper rotation of l_1 around the center of c_1 such that l_1 is parallel to l_2. The computation involves l_1 and l_2 only, and the only change is that of the slope of l_1.

- Qualitative Constraints

  The operational transformation approach used in OTP provides a convenient means for dealing with qualitative constraints. It is sometimes difficult to represent a constraint analytically, in terms of algebraic equations (e.g., symmetries). Furthermore, constraints involving inequalities and higher degree polynomials are sometimes difficult to solve. On the other hand, it is often possible to devise procedures for satisfying and maintaining a constraint using an appropriate set of primitive operations.

9. Conclusion

We presented a constraint-based design system based on the Operational Transformation Planning (*OPT*) approach. In OTP a constraint is satisfied by devising a sequence of operations that transform a given configuration to one where the constraint holds. The construction techniques of Euclid in his proofs of elementary geometry theorems are good examples of the operational approach to constraint satisfaction. The OTP approach is quite suitable for interactive design because of its incremental satisfaction of constraints and preference for local modifications. OTP tries to avoid transforming constraints into systems of algebraic equations, when possible. Thus, global computation, e.g. solving systems of equations every time a dimensional value is changed, is replaced by local computations incorporated into functional procedures. Furthermore, through geometric reasoning, OTP can avoid some geometric inconsistency problems. In our system, abstract information and embedding information of a design are separated and dealt with at different levels, The effects of a change to a design are computed locally through a sequence of well-defined operations. Therefore, OTP systems can be semantically more robust and computationally less expressive than systems based on the algebraic approach. Especially, OTP systems can deal with a wider range of geometric constraints, such as qualitative constraints which are difficult to incorporate into the existing CAD systems.

References

1. F. ARBAB AND J. M. WING, ''Geometric Reasoning: A New Paradigm for Processing Geometric Information,'' in *Design Theory for CAD, Proceedings of IFIP W. G. 5.2 Working Conference 1985 (Tokyo)*, ed. H. Yoshikawa, pp. 107-121, North-Holland, Amsterdam, 1986.
2. F. ARBAB AND B. WANG, ''Reasoning about Geometric Constraints,'' in *Preprints of the 2nd IFIP 5.2 Workshop on Intelligent CAD*, Cambridge, UK, September 19-22, 1988.
3. F. ARBAB, ''Examples of Geometric Reasoning in Oar,'' in *Intelligent CAD Systems 2: Implementational Issues*, ed. P. J. W. ten Hagen, T. Tomiyama, and V. Akman, Springer-Verlag, 1988.
4. F. ARBAB, ''Preliminary Report on Oar: a System for Objects And Reasoning,'' Technical Report, Computer Science Department, University of Southern California, November 1988.
5. A. BORNING, ''The Programming Language Aspects of Thinglab, A Constraint-Oriented Simulation Laboratory,'' *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 4, pp. 353-387, October 1981.
6. B. BRUDERLIN, ''Constructing Three-Dimensional Geometric Objects Defined by Constraints,'' in *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, pp. 111-130, ACM Press, 1986.
7. R. C. HILLYARD AND I. C. BRAID, ''Analysis of Dimensions and Tolerances in Computer-aided Mechanical Design,'' *Computer-Aided Design*, vol. 10, no. 3, Butterworths, June 1978.
8. F. KIMUARA, H. SUZUKI, AND L. WINGARD, ''A Uniform Approach to Dimensioning and Tolerancing in Product Modelling,'' in *Proceedings of CAPE '86, Second International Conference on Computer Applications in Production and Engineering*, pp. 165-171.
9. F. KIMUARA AND ET AL., ''Variational Geometry based on Logical Constraints and its Applications to Product Modelling,'' in *Proceedings of CIRP '87.*
10. H. LIEBERMAN, ''Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems,'' *special issue of ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 214-223, November, 1986.
11. H. LIEBERMAN, ''Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object Oriented Systems,'' in *3eme Journees d'Etudes Langages Orientes Objets*, ed. P. Cointe, AFCET, Paris, France, 1986.
12. R. A. LIGHT AND D. C. GOSSARD, ''Modification of Geometric Models through Variational Geometry,'' *Computer-Aided Design*, vol. 14, no. 4, Butterworths, July 1982.
13. R. A. LIGHT, ''Variational Geometry: Modification of Part Geometry by Changing Dimensional Values,'' in *Proceedings of Conference on CAD/CAM Technology in Mechanical Engineering*, MIT, March 1982.

14. V. C. LIN, D. C. GOSSARD, AND R. A. LIGHT, "Variational Geometry in Computer-Aided Design," *Computer Graphics*, vol. 15, no. 3, Association for Computing Machinery, August 1981.

15. J. R. ROSSIGNAC, "Constraints in Constructive Solid Geometry," in *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, pp. 93-110, ACM Press, 1986.

16. J. R. ROSSIGNAC AND ET AL., "Interactive Design with Sequences of Parameterized Transformations," in *Proceedings of the Second Eurographics Workshop on Intelligent CAD Systems*, Veldhoven, The Netherlands, April 1988.

17. D. SERRANO AND D. GOSSARD, "Constraint Management in MCAE," in *Artificial Intelligence in Engineering: Design*, ed. J.S. Gero, Computational Mechanics Publications, 1988.

18. I. SUTHERLAND, "Sketchpad, A Man-Machine Graphical Communication System," Phd thesis, MIT, 1963.

Paper Session *User Interface*

Aart Bijl

POINTER: **Picture Oriented INTERaction**

A Programme for ICAD/HCI Research

POINTER: Picture Oriented INTERaction

A Programme for ICAD/HCI Research

Aart Bijl
EdCAAD, Univ. of Edinburgh
20 Chambers Street
Edinburgh EH1 1JZ

March 1989

This paper outlines a general strategy for ICAD/HCI, providing a context for collaborative research projects. An instrumentalist approach to computer systems is advocated, together with a quasi-linguistic treatment of graphics, leading to systems by which users can construct and modify expressions of their own knowledge.

Experience of developing the MOLE system (Modelling Objects with Logic Expressions — Tweed *et al.* 1988),[1] and of its use by student architects and other researchers, has provided some pointers to human/computer interaction, HCI. This interaction has to reconcile human knowledge with partial and formal models of knowledge, as held in a representation scheme in a computer. It must give users control of operations on those models. When the users are designers, then this ambition presents profound and as yet unresolved problems.

In this paper, we start by making some fundamental observations about the strategy generally adopted for ICAD (intelligent computer aided design). This assumes the possibility of achieving correspondence between computational models of domain knowledge and a user's own knowledge, and that this correspondence can be valid even if it is not visible to designers. We then show how MOLE is based on a different strategy. This accepts that we cannot know to what extent any prior model of domain knowledge actually is the same as a designer's own knowledge. Instead, we see domain models as occurring in readings of expressions at the interface between a computer and a designer, and we see these readings as remaining the responsibility of the user. The role of MOLE is to support the production and modification of expressions that get passed between people.

The strategy illustrated by MOLE presents interesting user interface problems. The user has to know the logic on which the system is built, in order to exploit that logic in the course of constructing expressions of design knowledge. The aim must be to make logic visible, accessible and useful to users. Only then can we overcome the prescriptive imperative of orthodox ICAD systems on design practices.

This is a major objective. This paper outlines a programme for collaborative research aimed at this objective, for discussion among researchers. It seeks to stimulate responses from within the research community, leading to collaboration that will be motivated by interests in fundamental research. To serve this purpose, this paper links theoretical interests in computational problems with practical interests in using computers.

1 MOLE has been developed with support from the Building Sub-Committee of the Environment Committee of the UK Science and Engineering Research Council, and has been described at the two previous ICAD Workshops.

Orthodox ICAD

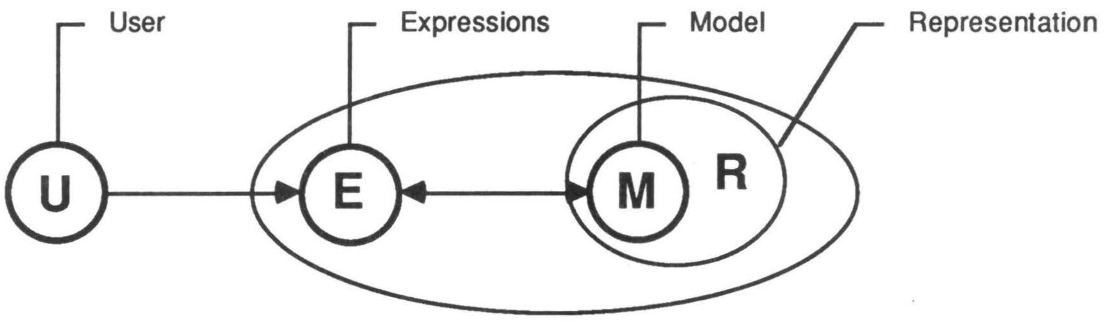


Figure 1: Elements of an ICAD system.

This is the usual schema for ICAD (and other computer systems) in which:

E is the form in which expressions are realised at the interface between a computer system and its user;

R is the representation scheme of the system, with associated facilities for operating on mappings (or interpretations) from and to E;

M is some contents in R, which may be regarded as a model representing a user's interest in using the system.

The central issue is what status and role can be attributed to such models, from a user's point of view? This question has to take into account the kind of being that users are, and it is dominated by the fact that our users are designers.

The usual answer, in the field of ICAD, is that these models correspond to domain knowledge. A model either represents some knowledge that has been prepared by someone else (an expert, or a programmer) and is thought to be relevant to some current activity of a user, or it represents a user's own knowledge which becomes evident in the course of such activity. The latter case is generally regarded as more ambitious and tempts us toward linguistic treatment of user behaviour, applied to users' expressions.

In both cases, the notion that a model can correspond to domain knowledge rests on an assumption that the model somehow replicates some part of a user. To fulfil the intention of this assumption, we find endeavours to rest the model on further logical models of the user and her or his world.

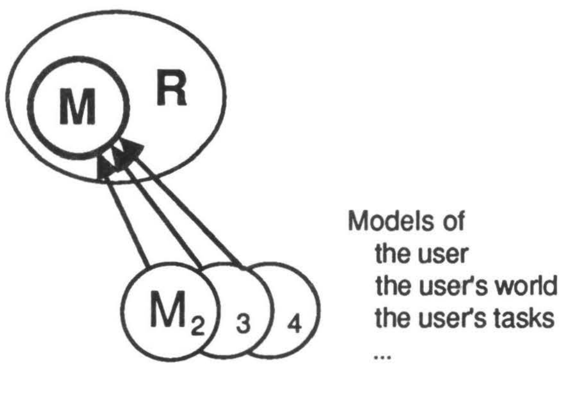


Figure 2: Systems resting on models of users.

Alternative Modelling Strategy

MOLE illustrates our attempt to explore an alternative assumption. We accept that we do not know whether any model of domain knowledge actually corresponds with that knowledge as it might otherwise be known within any user of the model. Basically, we accept that we cannot know. Our aim is to go as far as we can towards domain independent systems — whilst acknowledging that it is not feasible to entirely divorce formal treatments of expressions from domain knowledge. The role of MOLE, then, is to allow users to express their own models in the form of text and drawings. These expressions can be read as realisations of abstract models, and it is these realisations which are passed between different users and between a user and a computer.

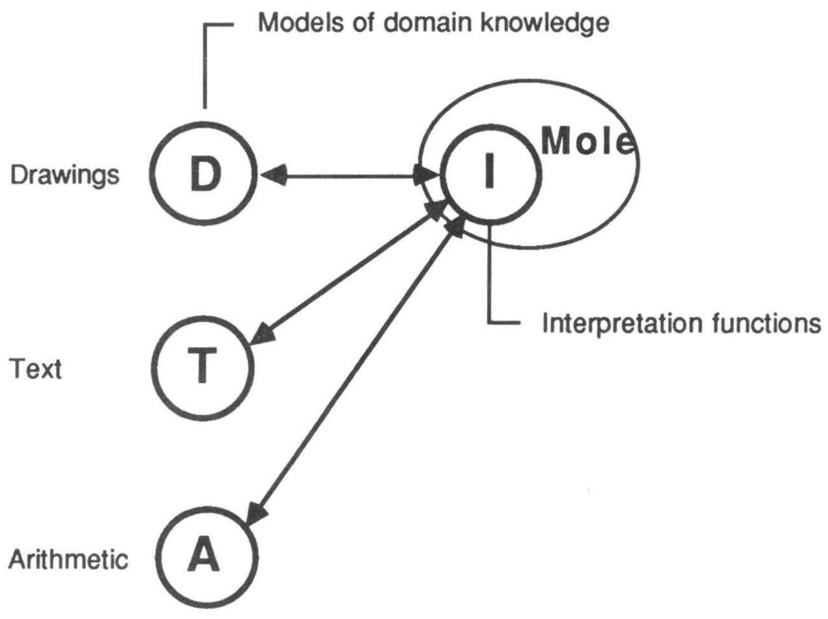


Figure 3: Models read from expressions.

In this schema:

- D is the form in which drawing expressions are realised, which presumes some drawing production machine based on a general structure for drawings and operations on drawings;
- T is the form in which text expressions are realised, which presumes some text production machine based on a general structure for text and operations on text (some formal syntax which can be used in conjunction with a computational sub-set of people's natural language);
- A is the form in which numeric expressions are realised, which presumes some arithmetic machine based on the logic of numbers;
- I refers to interpretation functions (or mappings), for translating expressions into logical 'meanings' and establishing interdependencies between drawings, text, and numbers — this presumes prior domain independent functions which can be employed by users when they build their own domain models.

Note that pictorial, textual and arithmetic expressions can all be regarded as specialisations of graphics, and it is generally only as graphics that they appear at the interface between humans and computers, on a computer display screen. These graphical forms receive symbolic (digital) representations within the computer, and symbolic logic is used to effect interpretations from one form to another.

Users' domain models in the form of drawings and text are represented within MOLE, in a logical form, and it is conceivable that different users will invoke each other's models from within MOLE. The difference in the schema presented by MOLE, compared with the more usual approach to ICAD, is that domain models serve as models only when users read them from realisations in the form of drawings and text. MOLE itself carries responsibility only for the consistency and coherence of its representation, and not for the consistency and coherence of domain knowledge. It is the users who are the modellers.

In this schema, abstract domain models are retained by users and the system provides support for realisations of these models. No domain specific models are held within MOLE in a manner that cannot be seen and modified by users. The interpretation functions on which the system is built, deals only with forms of expression and logical relationships between parts of expressions. The system includes *part of* and *instance inheritance* relations, plus *variables* and *conditions,* and *evaluations* (see later example). These relations and operations can be applied to parts of character strings, and to parts of line drawings (to values for angles, lengths, and kinds of attachments), and they can be used across text and drawings.

By invoking logical relations, users can define rules in the system, which establish connections between particular expressions in the form of text and drawings. These can be grammatical rules employed linguistically, or more simply they can be rules for identifying and linking expressions in a manner that users find relevant to their own domain knowledge.

User Interfaces

The problem which now emerges is how can a designer use a system such as MOLE — what is expected of the user? From experience, the following answer seems to be inescapable: the user has to be familiar with the logical relations on which the system is built. The user has to be familiar with logic in the abstract, in order to use the machine's implementation of logic to produce and interact with expressions, and to control the accumulation of expressions that is represented and stored within the system. The user has to know how the system's use of logical relations affects the consistency and coherence of the machine's representation of his or her design knowledge.

For practising designers, this is an awesome and unfamiliar responsibility. It entails a mode of thought that appears to be foreign to the activity of designing. In effect, we are asking designers to program their computers.

It is usually argued that this observation can be moderated in a number of ways. In programming their own machines, designers can expect to reuse their programs, recalling old programs, so that they do not always start with an empty machine. They can reuse programs that make the machine exhibit desired behaviours in producing and modifying expressions. Different designers might share each other's programs. However, while all this might be true, this speculation does not remove the fundamental requirement that the designer has to know and become familiar with the logic on which the system is built — otherwise the designer will not be able to modify existing programs to meet changing demands that come from the world of design.

More commonly, it is argued that other people should program computers for designers. Here we find the old distinctions between systems and application programs, and computer programmers and end-users — applications depend on users' knowledge being programmed in some prior way, within computers. The objection to this strategy is that it leads to prescriptive systems which compel users to work within the bounds set by application programmers

As a variant of this orthodox strategy, we have 'user friendly' systems — exemplified by the Macintosh's standard window/icon/menu interface, used in the same way by different application programs. Friendliness seems to refer to the surface behaviour of a system, evident in a graphical form which is attractive to users and constant across applications. This works for regular 'simple' applications but causes problems when users have particular needs, when they find themselves in contention with detailed decisions taken by application programmers. The friendliness of the interface then obscures the logic of the system from the view of the user, and results in highly prescriptive programs.

Lastly, we should note that even in the case of these friendly systems, users have to program their knowledge into these systems — they have to work within the limits of the procedures that constitute computer applications.

All these strategies do not remove the fundamental requirement that a designer has to know the logic of the system. It is said that people ought to be logical anyway and, therefore, for sensible people, this requirement ought not to be too onerous. Such a position assumes an equivalence between computational logic and ordinary human logic — an assumption that is difficult to sustain. Instead, we might see the problem of designing an ICAD interface as one of reconciling the difference between computational logic and the separate logic of users. This problem is profound if we accept that we cannot achieve a prior formal understanding of the logic of users, as is the case when users are designers. Research then needs to focus on the problem of how to make the logic of a system visible, accessible and controllable by users, so that designers can decide how to use the system to express their own design knowledge.

POINTER

POINTER is a proposed framework for a collaborative programme of research, arising out of experience of MOLE and earlier CAD systems, and more recent work on combined NL and graphical interaction with a computer's knowledge base. This programme is aimed at a mode of human/computer interaction, HCI, which will allow persons to interact with computational processes by means of *pictures*. The focus of this interest is on the *construction* of graphical expressions with *interpretations* to other (symbolic) representations. The purpose is to achieve system generated responses which *modify* graphical expressions, in a manner that will correspond with user expectations. For this purpose, we propose a quasi-linguistic approach to graphics, covering:

- meaning of graphical compositions in context of use;
- representation of that meaning in computational symbolic logic;
- use of an interactive interface to extend and change meaning.

This emphasis on pictures used in conjunction with words is crucial to the development of an acceptable HCI system for ICAD.

Market

The potential market for *POINTER* developments is very large, including the kind of computer users who currently use Macintoshes and PCs with graphics. These developments will be useful in design fields where graphics is used to express and manipulate information, making CAD applications more accessible and controllable by designers. More generally, these developments will be useful to managers and policy makers in commerce and industry, who want to be able to control their views of information held in computers and who find pictorial presentations of complex information more understandable.

POINTER developments, extending beyond the paradigms of Mac HyperCard and object-oriented programming, can be expected to contribute to the next advance in popular and friendly user interfaces.

Background

Present HCI systems that use pictures, as exemplified by Macintosh icons, offer graphic primitives which serve as commands or 'canned phrases'. Users cannot compose these primitives into newly articulated graphical expressions with interpretations. These systems also allow mixing of bit map images, but they 'know' of no further structure of images which could support interpretations to other representations of things that images depict. These limitations restrict users' graphical access to computational processes which might otherwise support representations of their own knowledge.

Early CAD systems (Hoskins 1977, Bijl *et al.* 1979) offered some facilities for constructing and interpreting graphical expressions, but these were highly domain specific. Attempts to achieve generality (Zdybel *et al.* 1981) have demonstrated potential, incorporating user models and semantic integration with the displayed images, but they have concentrated on graphical presentation. More recently, in the field of geometry modelling, there has been renewed recognition of the need for user control of concepts which relate graphical expressions to associated knowledge of materials, techniques and client specifications (Opas and Mantyla 1988, Requicha 1988).

A linguistic approach to graphics has been adopted for a large ESPRIT project, (P393) 'ACORD', which includes EdCAAD and the Edinburgh Centre for Cognitive Science among its partners (Klein 1987). Results are promising (Lee *at al.* 1986, Pineda *et al.* 1988, Szalapaj *et al.* 1988). That project covers formal treatments of predefined graphic objects which are used as system primitives with certain variable properties and relations. Now the aim is to progress towards user defined graphical compositions with user defined meanings.

More generally, an appreciation of graphics as a communication medium is relatively undeveloped in the fields of computer science, artificial intelligence, and cognitive science. Few cognitive psychologists have looked into the mechanisms of picture production and associated interpretation (Sommers 1984). How pictures can be used to convey varied kinds of information remains an important question, and new research needs to contribute to and stimulate interest in this area.

Role of Computers

As a starting point, we propose an *instrumentalist* approach to computer systems. Quite simply, this means that we look upon computers as instruments for doing things, without the implication that they *model* people or the things that people do. Computers can be regarded in the same way as we regard other tools, such as mechanical tools which are used because they enable us to do things. We don't think of those tools as needing to incorporate models of ourselves — they just do things that we find useful. Computers are no different. The ramifications of this position run deep, and only a few consequences will be considered here.

The thing that a computer does is perform operations on expressions, including the execution of mappings between different forms of expression, and the application of formal systems (such as arithmetic) to expressions. In all cases, these are expressions from persons and we can regard them as representations of what the persons know — but a computer does not know that they are representations.

By executing complex sequences of operations on many expressions, a computer can be made to look like it is behaving as a person. Thus we are tempted into a common confusion over the distinction between a representation and whatever it represents, especially if it is thought to represent human knowledge. We are tempted to try to make computers be more like persons, or make them do things more like persons do them, and make computers that can understand the behaviour of persons. We find ourselves embarking on a major enterprise to devise computational models of the world of people, and of persons in this world. However, even sophisticated computers remain just machines for executing operations on expressions. We should not reduce the impact of this fact by claiming that humans are no different (Bijl *et al.* 1989, Bijl 1989) — we may not know (in an overt sense) what expressions represent within persons, and we ought to accommodate the possibility that expressions and whatever generates them from within persons are different.

An instrumentalist approach to computer systems accepts them as tools, albeit highly sophisticated tools, which persons can use to construct their own expressions for purposes of communicating with other persons. This includes the possibility of constructing expressions that have the effect of modifying other expressions, and the possibility of persons gaining access to and using each other's expressions. The strong implication here is that persons using computers to construct expressions have to know the tools which they are using, and have to retain responsibility for their expressions. Persons have to know the behaviour of computers for purposes of constructing expressions, without necessarily knowing their inner operations which support their behaviour. Here we come to the essential focus of HCI, the study of inner operations of computer systems which can support behaviour appropriate to people's construction of expressions.

Fundamentally, this position is compatible with those of cautious AI advocates, but it focuses attention on how the role of computers can or should be viewed by users. Computers are seen here as tools offering logical inferencing mechanisms which people can use to construct their own models of their domain knowledge — these inferencing mechanisms can be similarly applicable to pictorial and written forms of expression.

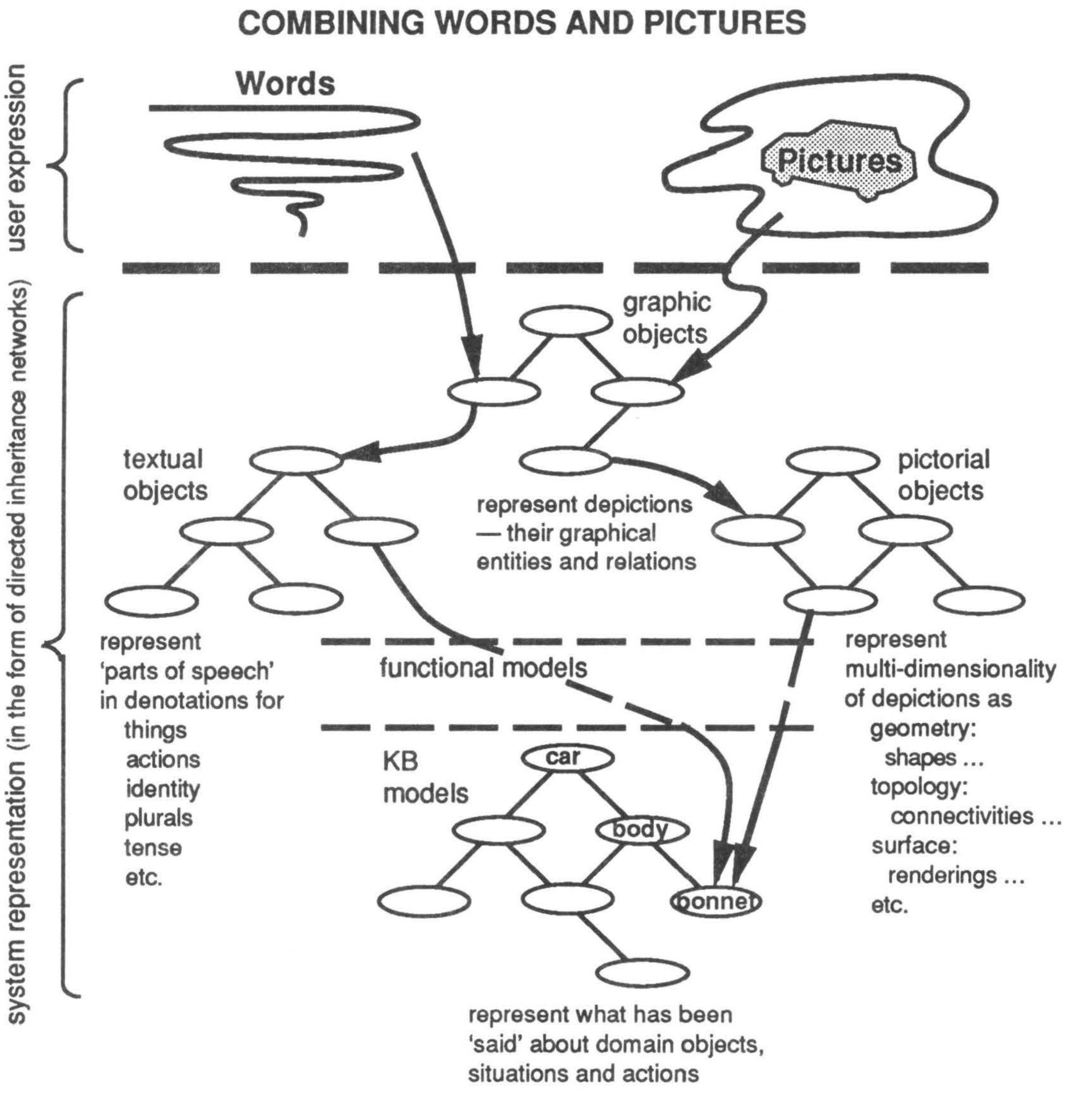
PROPOSED SYSTEM STRATEGY

Our proposed strategy for HCI, as indicated in Figure 4, is based on the observation that all expressions at the interface between a user and a computer, on a screen, are in graphical form. Graphics here comprises 2-D spatial arrangements of lines or other *graphic objects,* and includes text characters. A representation for graphics can therefore comprise mappings to *pictorial objects* defined in terms of their syntactic categories (i.e. possible readings of graphical compositions), and to *textual objects* defined in terms of similar categories for character strings. These objects can be 'known' in *lexica* of picture parts and words so that they and their associated relations can be used by further computational processes.

This grouping of text under graphics may appear strange — but we are interested in operating on spatial relations in compositions of pictorial and textual objects. We want to include the possibility of attaching significance to variations in depictions of words.

Linguistic treatment of pictures rests on *lexical definitions* for parts of pictures, which support *parsing* and *semantic representations* for pictorial compositions. The separate functionalities of pictures and words can then be combined in a system's composite representations of screen images. *Functional models* represent tasks that can be applied to interpretations from graphical images, such as generating 3-D spatial projections from 2-D pictures. They can receive and act upon pictorial and textual objects, and pass results to the system's *knowledge base (KB)*, as users' domain specific models which can be recalled in the form of graphical images on the screen.

Our focus on expressions sees them as representations of human knowledge, covering a spectrum from general knowledge of treatments of expressions as language, to domain knowledge described through use of language. A system that represents treatments of expressions, built on lexical entries (like Smalltalk objects, but oriented toward linguistic behaviour) with semantics known to users, can be used to construct and modify expressions of domain knowledge, the kind of expressions that are passed between people. The functionality of the system will then consist of its application of inferencing mechanisms to uses of lexical entries and, consequently, to users' graphical expressions of their domain knowledge. We see here an important distinction between the system's use of its internal resolution facilities which condition its representations of user expressions, and users' use of their knowledge about lexical entries when constructing expressions — the latter gives users control over the behaviour of the system.



COMBINING WORDS AND PICTURES
user expression
system representation (in the form of directed inheritance networks)
Words
Pictures
graphic objects
textual objects
pictorial objects
represent depictions — their graphical entities and relations
represent 'parts of speech' in denotations for things actions identity plurals tense etc.
functional models
KB models
car
body
bonnet
represent multi-dimensionality of depictions as geometry: shapes ... topology: connectivities ... surface: renderings ... etc.
represent what has been 'said' about domain objects, situations and actions

Figure 4: This is a tentative picture of the strategy outlined in this paper — and it is an example of a graphical expression which combines pictorial and textual objects.

The system has to make use of differentiations which it can 'read' in its representation for graphical images. This applies to differentiations in arrangements of lines depicting picture parts and to differentiations in strings of text characters. In the case of pictures, differentiations in images can be mapped to pictorial objects that might be defined in terms of:

classes of boundary conditions to 2-D spaces, such as lines (with angle and length) and polygons (with area ...);

connectivities between such objects, as topologies;

surface treatments of objects, as in colour and texture rendering.

Such dimensionalities need to be calibrated against (arbitrary but agreed) units of measure, for purposes of identifying values which can be passed from the image through the lexica and on to functional models.

A user will normally need to be aware of both pictorial objects and textual objects, by means of text and other depictions on the screen. The user will have to know enough about these objects in order to use them to construct expressions of her or his own domain knowledge.

The system's symbolic representation for uninterpreted images will not normally need to be known by users. However, we can expect that the structure of a representation, corresponding to the extents and connectivities of graphic entities in an image, will be evident in the depiction on the screen.

RESEARCH GOALS

The aims of *POINTER* can be supported by research on separate fronts, on separate aspects of this system strategy. Advances on each front can contribute to the success of a composite system for effective HCI. However, an advantage of the strategy which has been outlined is that no one system component should be critical to the realisation of other components — the strategy allows for piecemeal advances to be assembled into ever better systems. This possibility rests on the necessary simplicity of the system's control mechanisms, which must not be too 'clever' (i.e. must not presume too much about the intentions of particular users). The aim is to allow users to exercise control over expressions and the system's corresponding representations of their own domain knowledge.

Research contributions can be focused on the following topics, with the intention that advances should have general relevance to future developments of commercial systems.

Image representation:

Definition of basic graphic entities, properties and relations from which any images can be constructed — serving as a graphical modelling environment for graphical productions. The research problem here is to distinguish between the graphical modelling environment and pictorial objects that can be modelled. To achieve generality, we have to consider graphical productions as being domain independent. We need to know of graphical compositions as things-in-themselves, which can be mapped to pictorial objects.

Representation of pictorial objects:

Definition of the dimensionalities of pictures that can be read from graphical compositions, including geometries, topologies, and surface treatments. This definition has to be specific to particular dimensions but, for generality, it otherwise needs to be domain independent. Dimensions can be defined discretely, with their own demands on the graphic representation, and certain dimensions might be interrelated. More than one dimension might share some common instances of graphic entities — as occurs in the familiar phenomenon of multiple readings of depictions. Dimensions need to include calibrations so that values can be identified in graphical compositions. These syntactic parts of pictures form the content of a picture lexicon. Users' graphical expressions can then be mapped to lexical entries and values can be passed on to functional models for generating further semantic representations from pictures.

Representation of textual objects:

Definition of character strings and their roles as 'parts of speech' (seen by users in compositions of text, and numerals), in terms of logical entities and their relations. The representation scheme has to accommodate logical differentiations for the presence of things (noun phrases), actions associated with things (verb phrases), identity (anaphoric reference ...), plurals, tense, etc. (as possible syntactic categories of formal languages, including NL systems as computational sub-sets of human language). These syntactic parts of writing form the content of a word (or phrase) lexicon. Users' expressions can then be mapped to lexical entries and values can be passed on to functional models for generating further semantic representations from written compositions.

Functional models:

Definition of tasks that can be applied to pictorial and textual objects, in terms of grammars or rule systems which correspond with generally accepted knowledge about treatments of those objects. These include systems for generating logical representations of NL semantics from users' expressions, systems for generating 3-D spatial projections from 2-D pictures, and system for establishing interrelationships between NL and spatial semantics. For generality, any functional models for establishing particular associations between pictorial and textual objects need to be valid for whole application domains. Results from these functional models need to be in a form that can be accommodated in a KB.

KB models:

Definition of a general storage and retrieval environment which can accommodate values abstracted from users' expressions, via the lexica and processed through functional models — these are the values 'read' from the spatial differentiations exhibited in graphical productions. KB representations then are domain specific models resulting from a user's use of the system.

Artificial intelligence:

The system's ability to execute mappings between an image representation, textual and pictorial objects, functional models, and KB models can be thought of as representing human intelligence — but the representation itself has to be viewed as artificial intelligence (with the implications noted earlier under *role of computers*). This ability rests on the system's in-built representation scheme (see later) which might exhibit and amplify behaviour which otherwise is thought to result from human cognitive processes — the system then serves as an instrument which amplifies certain mental powers of people.

The system's representation scheme has to maintain consistency and effect coherence across the above system components. This is a major research ambition, but a modest achievement is likely to suffice for the development of practical systems which allow users a lot of control over system behaviour.

GENERAL SYSTEM ARCHITECTURE

The above topics fit into a general system architecture as indicated in Figure 5.

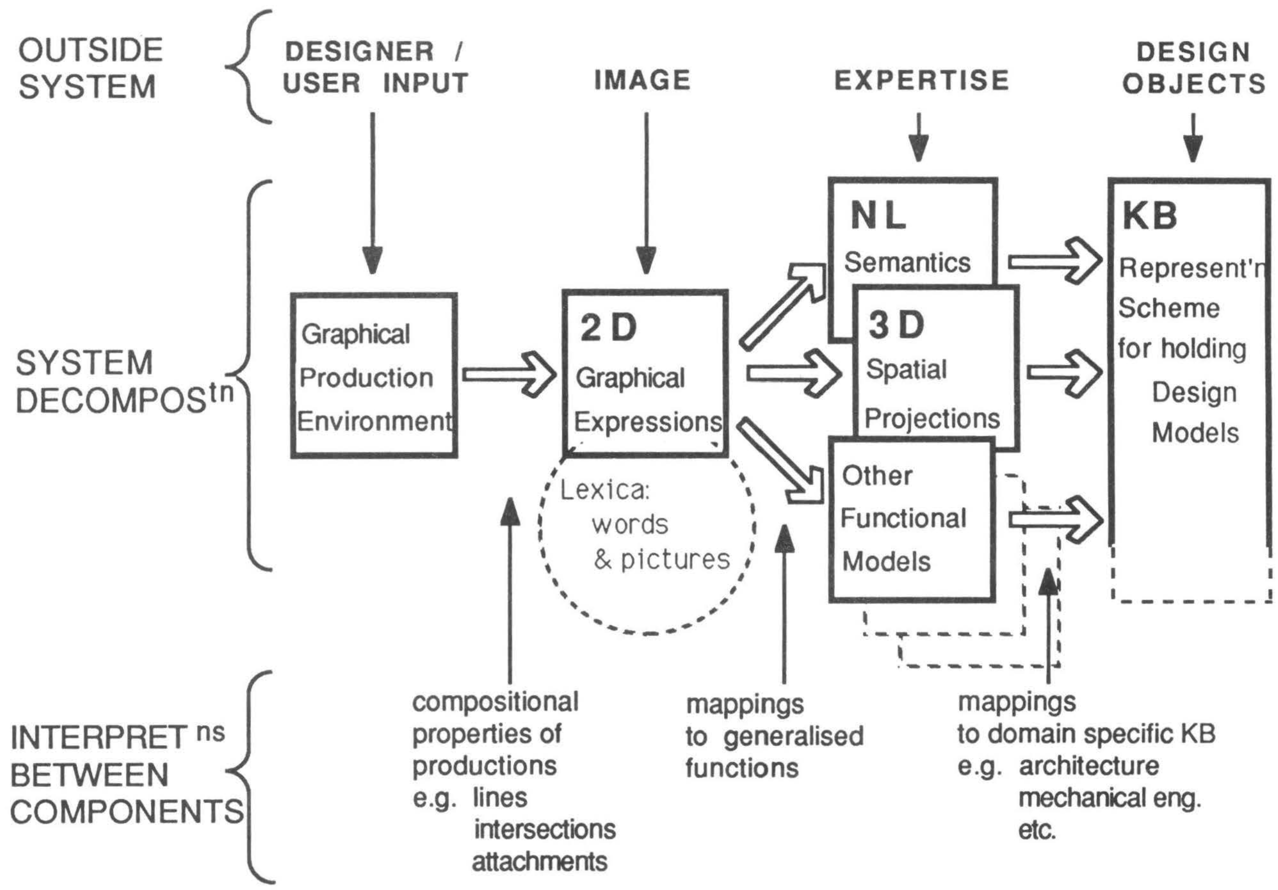

OUTSIDE SYSTEM
DESIGNER / USER INPUT
IMAGE
EXPERTISE
DESIGN OBJECTS
SYSTEM DECOMPOStn
Graphical Production Environment
2 D Graphical Expressions
NL Semantics
3 D Spatial Projections
Other Functional Models
KB Represent'n Scheme for holding Design Models
Lexica: words & pictures
INTERPRETns BETWEEN COMPONENTS
compositional properties of productions e.g. lines intersections attachments
mappings to generalised functions
mappings to domain specific KB e.g. architecture mechanical eng. etc.

Figure 5: A tentative general architecture for *POINTER* systems.

The main system components are a graphical production environment which behaves as a *graphics machine* for realising *graphical expressions* or screen images — these components, operating under the control of users, can be regarded as somewhat equivalent to people's normal practices in *constructing* images, whether or not they are using computers. People might use *lexica* for expressions in order to interpret them as words and pictures with associated grammar rules, to condition their further actions on expressions. Lexica support *parsing* of expressions to produce *semantic representations* in terms of further expressions that are acceptable to actions or functions that might be applied to a currently extant expression, or an image on the screen. Such functions are represented by further system components, as *functional models* corresponding to human expertise on formal treatments of expressions. Results from functional models are deposited in the system's *KB*, as users' domain specific models which can be recalled as images on the screen. These latter system components offer computational processes for constructing and modifying users' expressions and, for this purpose, the behaviours of these components have to be known to users.

The architecture of a system has to include provision for communication between components, to map entities into the forms that can be used by the different components, and to maintain logical consistency and coherence across all representations held within the system. This ability of a system has to rest on a general representation scheme and its associated control features, which will be outlined in the next paragraphs.

Representation scheme:

Construction of a knowledge representation scheme and the operations over knowledge representations needs to be guided by the general strategy for graphics outlined earlier. We want a representation scheme that can apply uniformly to all components of a system, covering graphic objects and derived pictorial and textual objects, plus functional models and KB models.

The representation scheme needs to include inferencing mechanisms of the kind now available in modern systems, such as those covering inheritance, defaults, updating, and constraint checking. These facilities must support relations between a graphical realisation of an image on a screen, and representations of depictions throughout the various components of the system, to its representations of users' domain knowledge. Relations also need to be supported in the reverse direction, to generate graphical realisations of words and pictures from existing states of representations within the system. Most critically, facilities of the representation scheme must accommodate the user's own input of expressions, in the form of words and pictures.

As a general computational strategy, we propose the use of DAG (directed acyclic graph) representations and the accompanying unification operations over them, which have already been applied in the fields of NL and graphics (Tweed *et al.* 1988, Lee *et al.* 1986). DAGs also are an appropriate choice because they are a natural representation underlying object-oriented programming. Moreover, we expect promising results from any study of the use of graph unification as a fundamental operation underlying graphical manipulation, knowledge representation (Ait-Kaci and Nasr 1986, Fargues 1986), and NL processing (Shieber 1986), which is aimed at an optimal computational model of interaction between these different representations.

For trial implementations, a graphics Prolog based representation scheme, MOLE (see later example), is available.

Control:

Control of system operations rests on distinctions between: control features of formalisms (as built into a logical representation scheme); control features of modelling components (as built into prior functional models); and user control of models used to modify expressions. The first entails representations of human intelligence built into machines, and this can benefit from AI techniques. The second and third cases entail people's intelligent use of the former, with the implication that users have to know the behaviour of formalisms (and prior modelling components) which they employ on their expressions.

Control expressions:

These are user actions aimed at operations which a user wishes to invoke within the system, with system defined interpretations: key hits, mouse clicks ... on characters, special symbols, menus, and icons.

EXAMPLES

The following two examples are included to illustrate some limitations of current picture oriented interaction, and the potential of *POINTER* developments. One is taken from EdCAAD's experience on the ACORD project (mentioned earlier), and the other from early experience of integrated CAD systems.

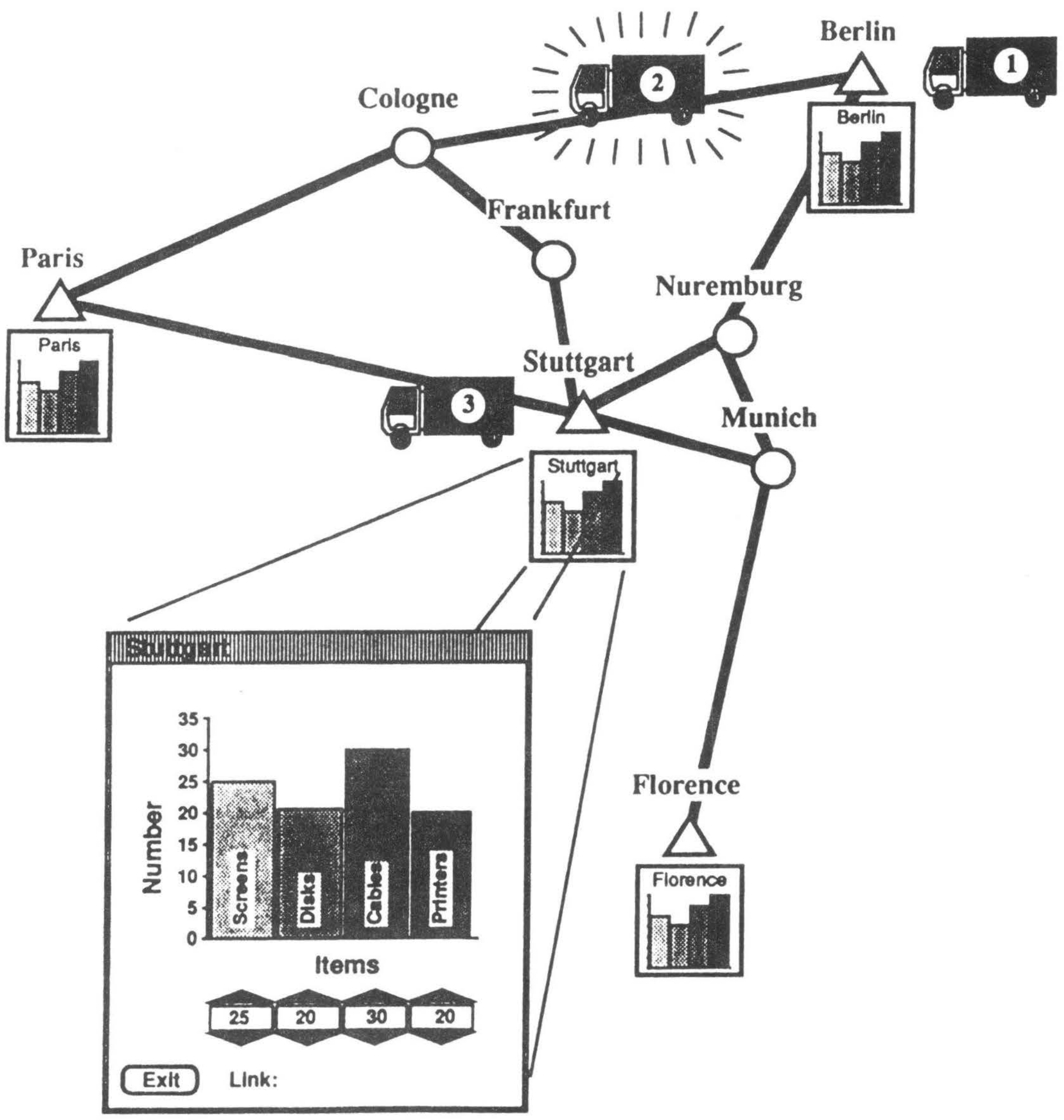
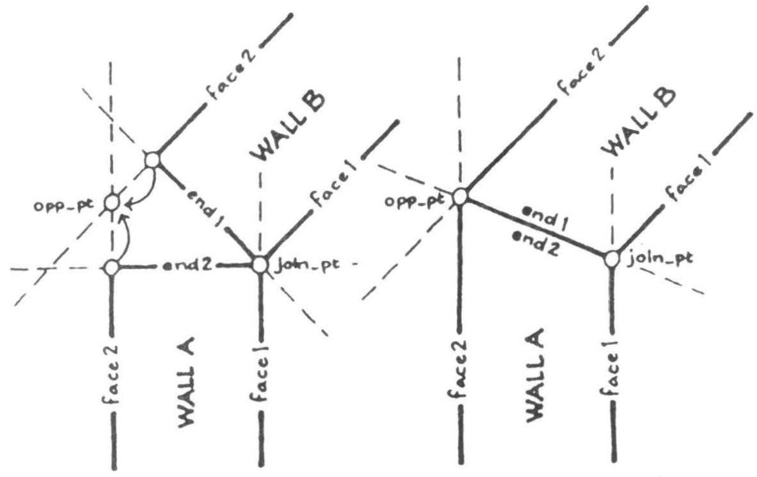
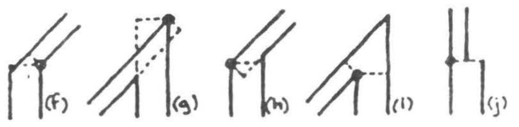


Figure 6: Map of transportation

The ACORD project has developed a prototype application representing transportation of goods, as shown in Figure 6. Graphics depict towns (as nodes) linked by roads (as arcs). Graphic symbols denote towns, and they are labelled with charts showing quantities of goods at each town. Lorries are denoted by further symbols which can likewise be labelled with charts showing their contents. These lorries can be placed at towns and they can be placed on roads or moved to other towns. When they are at towns they can load or unload goods. NL and direct manipulation of graphics are used to say what is happening, and consequences appear in revised charts and changed positions of lorries. This application treats graphical entities as pre-defined primitives with certain variable properties and relations, such as a lorry being *on* a road or *at* a town. In the normal course of using the system, a user cannot, say, decompose and redefine a symbol to denote a certain kind of lorry, or define relations for lorries that will account for one being away from any town and off a road (broken down?). In this respect, the development of graphics as a means of expression remains weak.





```
JUNC_END:
    [join_pt = fix_wall:join_pt,
     opp_pt:
        [conline1 = fix_wall:face*|{:conpt* = join_pt}:bearer,
         conline2 = join_wall:face*|{:conpt* = join_pt}:bearer],
     change:
        [fix_end = fix_wall:end*{:conpt* = join_pt},                JOIN_END_AB:
         join_end = join_wall:end*{:conpt* = join_pt},                  [fix_wall = WALL_A,
         move1 = fix_end:conpt*|{ = join_pt}:(opp_pt:* = *),             join_wall = WALL_B,
         move2 = join_end:conpt*|{ = join_pt}:(opp_pt:* = *)]].          join = ˉJUNC_END].
```

Figure 7: Context sensitive junctions

Figure 7 shows an example originally from CAD and reworked using MOLE (Modelling Objects with Logic Expressions — Krishnamurti 1986, Tweed *et al.* 1988, Bijl 1989). This system allows users to define descriptions of objects and functions that can be applied to them, and includes graphic entities as possible objects. MOLE includes general treatments of inheritance and updating in a context of parts hierarchies and kind relations, such that any object can be a sub-part and a super-part, and a sub-kind (child) and a super-kind (parent) of other objects, and descriptions can include conditional and replacement expressions. A system component used to produce graphical images becomes a user of the MOLE representation scheme (Szalapaj 1987, 1988).

The problem illustrated here is to join two context sensitive spatial objects, and form a junction that will establish spatial continuity. The expressions (lower part of figure) show a representation of the general case of this kind of junction, with join parts and change parts, and an instance in context (A and B). The diagram (upper part of figure) shows certain spatial properties of the objects before and after being joined. The diagram includes hierarchically structured graphic objects with names and spatial properties, down to the level of lines with angle values, and these are used in the representation. This representation can then instantiate or modify particular junctions (with different length and angle values), or it can reflect different instances that might be drawn graphically. The substantial problem which remains is that we lack orderly formal mechanisms for inter-relating representations of graphic objects with further representations of depicted objects, so that subsequent changes to either representations will have effects that can be known and anticipated by users.

CONCLUSION

This paper has outlined a major programme for HCI research. Its starting point rests on a distinction between human knowledge, within persons, and expressions as representations of that knowledge, external to persons. Humans use expressions to get at and share each other's knowledge, and we can do so because we are able to appreciate the distinction between overt expressions and our sense of knowing within each of us — a distinction that is evidently employed by designers, and managers and policy makers.

Computers do not sense this distinction. They are machines for performing operations on expressions and we can use them as tools for constructing and modifying our expressions. Computers remain just such machines, even when their operations on many expressions contributed by different persons make them look like they are behaving intelligently. The role of AI techniques has been identified with a system's inferencing mechanisms associated with its general representation scheme, within the system, which condition its surface behaviour as evident to users. These facilities have to maintain logical consistency and coherence across the system's representations of user expressions, and they condition system responses to user actions.

The focus for research on HCI then has to be on the possible representations within computers, of user expressions, so that a system's behaviour can be intelligible and controllable by users. This focus is central to all potentially practical computer systems, and it provides scope for a major programme of collaborative research projects.

REFERENCES

Ait-Kaci, H. and Nasr, R. (1986) LOGIN: 'A Logic Programming Language with Built-in Inheritance', *Journal of Logic Programming 3.*

Bijl, A. (1989) *Computer Discipline and Design Practice — Shaping Our Future,* Edinburgh University Press.

Bijl, A. and Pineda, L.A. (1989) 'Notions of Language and Design for Intelligent CAD' in Tomiyama, T. and Holden, T. (eds) *Intelligent CAD II,* North-Holland.

Bijl, A., Stone, D. and Rosenthal, D.H. (1979) *Integrated CAAD Systems,* EdCAAD Report for the Dept. of the Environment, University of Edinburgh.

Fargues, J., Landon, A.M.C., Dugourd, A. and Catach, L. (1986) 'Conceptual Graphs for Semantics and Knowledge Processing', *IBM Journal for Research and Development 30.*

Hoskins, E.M. (1977) 'The OXSYS System' in Gero, J.S. (ed) *Computer Applications in Architecture,* Applied Science, pp.343-391.

Klein, E. (1987) 'Dialogues with Language, Graphics and Logic' in *ESPRIT '87: Achievements and Impact,* North-Holland.

Krishnamurti, R. (1986) 'The MOLE Picture Book: On a Logic for Design', *Design Computing 1(3).*

Lee, J.R., Bijl, A. and Szalapaj, P.J. (1986) *The Graphics Component of the ACORD System,* EdCAAD/ACORD working paper.

Opas, J. and Mantyla, M. (1988) 'Introducing Manufacturing Knowledge into Intelligent CAD Systems', 2nd IFIP WG 5.2 Workshop on *Intelligent CAD,* Cambridge.

Pineda, L.A., Klein, E. and Lee, J.R. (1988) 'GRAFLOG: Understanding Drawings Through Natural Language', *Computer Graphics Forum,* 7.

Requicha, A.A.G. (1988) 'Geometric Modelling and Programmable Automation', IFIP TC5 conference on *CAD/CAM Technology Transfer to Latin America,* Mexico City.

Shieber, S.M. (1986) *An Introduction to Unification-based Grammar Formalisms*, CSLI Lecture Note Series.

Sommers, P. van (1984) *Drawing and Cognition,* Cambridge University Press.

Szalapaj, P. (1987) 'A Transformational Grammar for Line Drawings', int. symp. on *Fuzzy Systems and Knowledge Engineering,* Guangzhou/Guiyang, China.

Szalapaj, P., Lee, J. and Bijl, A. (1988) *The Semantics of Computational Systems that Depict Non-Graphical Information,* Esprit P.393 ACORD deliverable T3.6b.

Tweed, C. and Bijl, A. (1988) 'MOLE: A Reasonable Logic for Design?' in ten Hagen, P.J.W., Tomiyama, T. and Akman, V. (eds) *Intelligent CAD Systems II; Implementation Issues,* Springer-Verlag.

Zdybel, F., Greenfeld, N.R., Yonke, M.D. and Gibbons, J. (1981) 'An Advanced Information Presentation System' in *Computer Graphics 81,* pp.19-36.

Christine Giger, Michael Lutz, Luiz Ary Messina

An Intelligent NC-Programming-System as a Significant Extension of Intelligent CAD-Systems

An Intelligent NC-Programming-System as a Significant Extension of Intelligent CAD-Systems

Christine Giger, Michael Lutz, Luiz Ary Messina

Technische Hochschule Darmstadt
FB Informatik
FG Graphisch-Interaktive Systeme
Wilhelminenstr. 7
D - 6100 Darmstadt
Federal Republic of Germany
Tel: 06151-1000-57
Telex 4197367 agd d Telefax 06151-1000-99
E-Mail giger@zgdvda.uucp

Abstract: Concerning intelligent CAD and NC systems, we focus on two points which seemed to be important. First, we try to specify a data structure for storing a design part together with its geometrical and technological attributes, in order to achieve the possibility of interaction between CAD and NC systems. Second, we offer a significant tool (line diagrams) for the graphical representation of knowledge, based on an object oriented approach. This tool is used to support a user in searching for variant parts or NC programs which are stored in a knowledge base.

1. Introduction

When dealing with the design process of a product, we must not suppress the aspects of the manufacturing process. Knowledge of the latter is often very important to the designer, because it may influence (for example) the shape or material of the part.

Unfortunately there are very few systems available which support an NC programmer in laying down the steps for the manufacturing process on numerical controlled machines (NC machines). Usually these systems are not on such a high level of development as most of the CAD systems are today. Besides, interfaces between CAD and NC systems can only be used for transmitting geometry (most of the time there does not exist any interface at all), which is not enough regarding the above mentioned purposes.

Therefore, our aim will be to design systems which "keep in mind" the manufacturing process while dealing with design and vice versa. This is not as difficult as it seems to be. Many problems (for example variant finding, natural language explanation, consistency checking, etc.) may require similar solutions for both parts of the system (CAD and NC). Since the knowledge of the NC process often seems to be more complex than the knowledge of the designing process, we want to focus on developing an intelligent NC system in this paper. Nevertheless, we always keep in mind that there must be a preparation for all concepts on the side of the CAD system. Besides, all the knowledge of the manufacturing process must also be available for the designer, perhaps with little modifications in its presentation on the screen (the designer will probably need more explanantion than an NC expert).

The most important aim now is to design a system which will be accepted by the majority of NC programmers who are accustomed to use procedural languages to produce NC code and for those who should learn the NC process of turning, drilling, milling, etc. For this purpose, systems must be able to offer at least as much functionality as procedural languages; but nowadays one should take advantage of Computer Graphics and Artificial Intelligence (AI) to meet some requirements of modern systems.

2. State of the art

Nowadays CAD systems are used as tools for the fast production of drawings for machining purposes, architecture, or VLSI design. Regarding CAD systems for machining purposes, their output is not more than simple (2D) drawing segments in the worst case, in the luckiest case we get a 3D description of the designed part. But interfaces such as IGES for example can only be used for transmitting geometry. Technological information (surface qualities, material attributes, etc.) which is usually a part of the semantics of an ordinary drawing, cannot be transmitted via a standard interface to an NC system at the moment. The other way round, the designer must have in his mind at least some knowledge of the manufacturing process. There is no possibility in getting support from the CAD system concerning NC requirements.

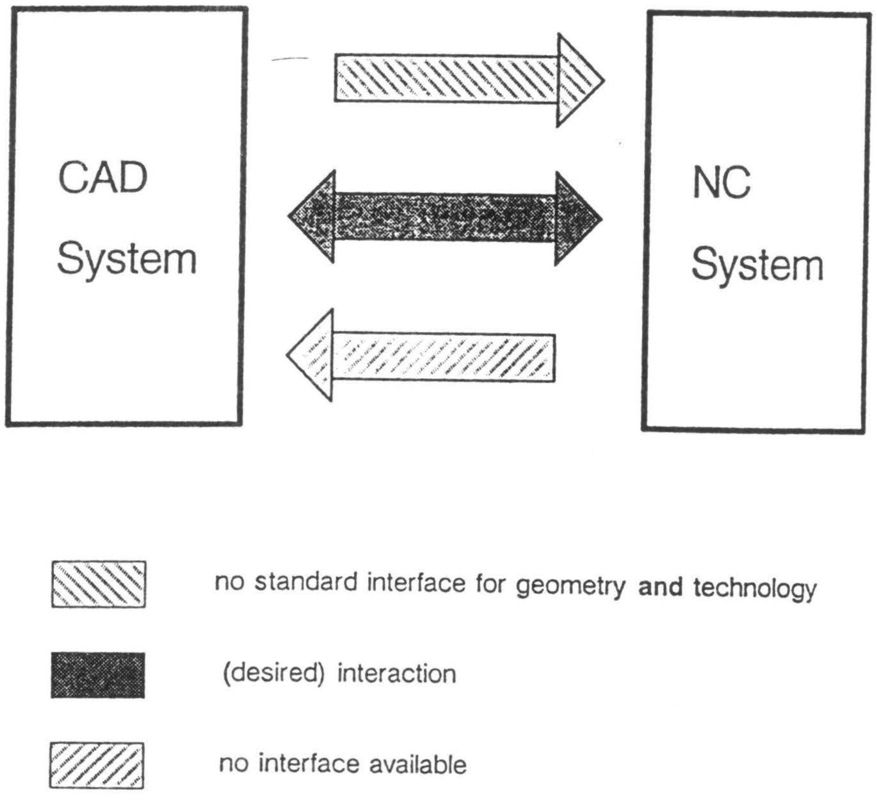


Figure 1: Connections between CAD and NC System

NC programming systems are used to produce NC code for different kinds of NC machines. Available systems can handle technologies such as turning, drilling, milling, grinding, flame cutting, and eroding. Systems usually offer tools such as an interactive graphics user interface, libraries encluding technological information, and (sometimes) the possibility of simulating parts of the machining process. Nevertheless, knowledge of the machining process on numerical controlled machines is essential to the user. Besides, he does not get any help in laying down the sequence of steps in the programming process, no support in choosing tools, machines, etc. Most of the time the user has to make decisions on the basis of his experience. It is difficult to get transparency about the number and kind of NC programs that are already stored in a systems library or data base. Usually the interface to this data base is a kind of "on-line catalogue" and the user has to know the special keywords (numerical or alphanumerical strings) for searching in the catalogue.

When every step of the machining process is completely defined by the user, the NC system produces a machine independent code which afterwards is adapted to a special machine with the help of a post processor. The resulting NC code can be stored in a data base or immediately output on a punched tape.

3. Requirements on intelligent systems

Nowadays all kinds of systems should have user interfaces that can be used by people without special education in computer science. Systems should act as an intelligent assistant; they must be able to help users to get quick and correct solutions of their problems. For this purpose the following requirements have to be met by the (NC) systems.

1) *Graphics tools*
Modern techniques for interaction, such as controlling with icon menus, mouse-input and window managing, must be available. This is necessary to ensure fast communication with the system.

2) *Flexibility*
Fast adaption and extension of the user interface must be possible. The user should have the possibility to influence the sequence of steps in the dialogue. Sometimes it may be useful to allow the user to change parts of the system layout.

3) *Easy comprehension*
The dialogue has to be designed to help the user solving his problems. The user must not be burdened with properties of the dialogue.

4) *System help*
For every step and situation in the dialogue system, help must be available.

5) *Representation of knowledge*
System answers must be presented in a form that users expect. Experience and usual expressions of the user's field of work should be integrated in the system.

6) *Stability of the dialogue*
The dialogue must not be inconsistent. Reactions of the system should be similar in comparable situations.

7) *Transparency by graphical representation*
   Most of the information in the system should be presented to the user with the help of pictures.
8) *Easy detectability of errors*
   Errors have to be avoided by the system as far as possible. If an error occurs which cannot be handled by the system itself, an error message should be sent to the user. Besides, the system should offer help to the user, so that he can easily detect the error.
9) *Simplification of repetition*
   The user must have the possibility of repeating certain sequences using different parameter values. In this case the usage of procedural languages can be avoided.
10) *Structure of knowledge base*
   Consistency checking, extension and manipulation of the knowledge base needs a problem oriented, modular, uniform structure of the data base.

4. Proposals for improvement

4.1. Interactive Graphics

In order to offer specialized help, the system must have a knowledge base. The way, in which knowledge is presented to the user, has a strong influence on the acceptance of the system.

Since graphical representations are easily understandable, knowledge visualization as a tool of communication in intelligent, interactive user interfaces is appropriate. The basic aim is to replace the menu technique by the use of graphical representations (semantic networks, line diagrams, etc.) as the elements of communication.

These graphical representations reflect the knowledge and the dependencies between knowledge according to the context and the notion of the user. Hence, the sequence of dialogue steps takes semantic aspects of the user into consideration. Furthermore, these graphical representations are a valuable support for creating, structuring and manipulating the knowledge base.

In order to get a good performance, geometrical information of graphical representations has to be included in the knowledge base. This means that a basic geometrical structure has to be added for appropriating parts of the knowledge base.

4.2. Variant programming

The amount of time necessary to design a part and to produce an NC program with the help of a CAD/NC system can be reduced by creating and manipulating variant programs interactively. The user should be able to carry this out without learning a procedural language. This can be achieved by using methods which are introduced in the section "Interactive Graphics".

The basic idea of finding a variant is the use of one of the well-known classification systems for contours of parts ([11]). The contour of every part is composed of contour elements. Every contour element is represented by a number and a certain position in a numerical string (see figure 2). In ordinary CAD/NC systems users can search for such a string or parts of the string if they want to find a variant. This method requests a lot of knowledge of the classification rules used in the system. Therefore most of the users think it is it easier to design a part (or NC program) completely new than to look for similar parts in the system.

	Part Class (a)	Outside Shape (b)	Inside Shape (c)
1	$L/D \leq 0.5$	smooth; w/o form elements	w/o drill hole; w/o cut through
2	$0.5 < L/D < 3$	unilateral increasing; w/o form elements	unilateral increasing or smooth; w/o form elements
3	$L/D \geq 3$	unilateral increasing or smooth; thread	unilateral increasing or smooth; thread
4		unilateral increasing or smooth; functional cut	unilateral increasing or smooth; functional cut
5		multilateral incr.; w/o form elements	multilateral incr.; w/o form elements
6		multilateral incr.; thread	multilateral incr.; thread
7		multilateral incr.; functional cut	multilateral incr.; functional cut
8		functional cone	functional cone
9		moving thread	moving thread

Figure 2: Example for a classification of turning parts with length L and diameter D

Our idea is to use an object oriented data structure for representing (variant) parts in the system. Technological attributes as well as contour elements are represented as attributes ("methods" in Smalltalk e. g.) of the stored objects. Rules of inheritance, which are father-son relationships, are based on the classification rules mentioned above. Non standard data bases are available (see [1], [8] e. g.) which can be used for this purpose. The advantages are obvious: users can define the desired parts in their own (natural language) expressions and the stucture of the stored knowledge (here variants) is pretty good understandable to a user. Nevertheless, there are still some disadvantages: representing object oriented data structure on a screen usually means showing a tree structure.

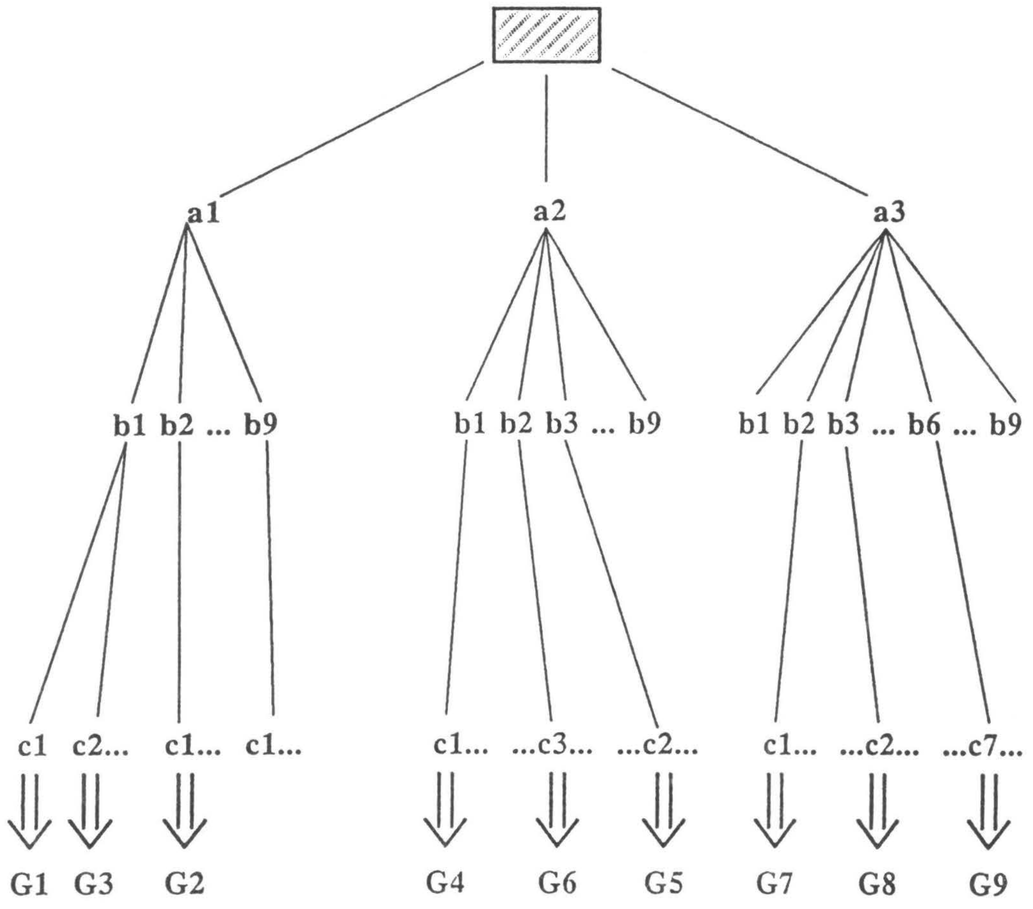


Figure 3: Usual graphical representation of the classification scheme in figure 1 (with classified parts G1 - G9)

Now imagine a user searching for an object with attributes a1, b2, c2. Of course, in the end he wants to see all objects with attributes a1, b2, c2 which are stored in the knowledge base. If he specifies the attributes in this order there is no problem for us to show the corresponding part of the tree to him. But what is to do if he does not remember the attributes in succession?

There are two possibilities: Either we force him to specify the attributes in a certain order or we must store in the data base all possible structures and relationships, which means storing a lot of redundant information. Since both possibilities do not seem satisfactory to us, we concentrate on a slightly different approach: line diagrams. The benefit is that we still have an object oriented approach, but attributes do not have to be specified in a certain order if there are no semantic reasons (consistence rules) for this. Besides, it is always visible which attributes can still be specified by the users in every step of the specification process.

Therefore, the user specifies the contour elements or other attributes with the help of line diagrams until the shape of the part or all steps of the machining process (NC program) are defined. While the searching process is controlled by the user with line diagrams, the actual state of the process must be indicated. For example, a window can show the composition of the already specified contour elements. The shape of a part which corresponds to a variant can be adopted as an icon in a special variant icon menu. As a result we can easily define, present, and choose variant programs.

5. Variant programming and line diagrams

5.1. Principles of variant handling

As we stated before, the most important aspect for an NC-programmer to accept the system is, that he has the possibility of working efficiently and effectively with the NC-programming system. For this purpose we allow the user to define variants by laying down the shape of the part and the corresponding steps of the NC-process interactively. Then the system is able to store the variant and shows this to the user by adopting the shape as an icon (for example). The problem which occurs here is, that probably hundreds of variants have to be stored in this way. Therefore, we must offer help for searching variants.

We solve this problem by parameter controlled searching and as a result we eliminate successively unfitting variants with the help of line diagrams. As search parameters we use geometrical attributes of the part to be manufactured.

The resulting number of possible variants is then presented to the user. For deciding which of those variants matches his special problem, he can get additional help from the system. For example, the system can indicate which parts of the NC-program might be influenced when certain values of parameters of the variant are changed.

Then the system can automatically present some tools which may be used for machining the part. The criteria for choosing these tools are

- geometry of the shape of the tool and the part,
- material of the tool and the part,
- strain of the tool,
- fastening of the part,
- smoothing or cutting a part.

If it is not possible to get a deterministic choice of tools, then line diagrams can also be used to present a certain amount of possible tools and their characteristics (attributes).

5.2. Line diagrams

In a knowledge base, information about available tools, materials, variants and connections between geometrical shapes and corresponding NC-programs are stored. For easy manipulation we need a clear, uniform concept for the knowledge base structures and their visualizations. Decisions must be made transparent and knowledge must be illustrated by the usage of graphical representation. A very powerful tool for this purpose is the application of the formal concept analysis.

The formal concept analysis turns the traditional, philosophical concept theory into a set-theoretic model. The basic data structure of the formal concept analysis is called context and is defined as a triple (G, M, I), where G is a set of objects, M a set of attributes and I a binary relation between G and M that specifies the correspondence between the objects and attributes.

We say $g\, I\, m$ to indicate that the object $g \in G$ has the attribute $m \in M$. As an example figure 4 shows a context that describes a classification of part contours, where the binary Relation I is given by the crosses. A cross in the row of the contour g_i and the column of the classification attribute m_j means that the contour g_i is characterized by the attribute m_j.

	a1	a2	a3	b1	b2	b3	b4	b5	b6	b7	b8	b9	c1	c2	c3	c4	c5	c6	c7	c8	c9
G1	x			x									x								
G2	x				x								x								
G3	x			x										x							
G4		x		x									x								
G5		x				x								x							
G6		x			x										x						
G7			x		x								x								
G8			x			x								x							
G9			x						x										x		

Figure 4: The part contours G1 - G9 are classified with regard to the classification scheme in figure 1

A pair (A / B) is said to be a concept of the context (G, M, I), if $A \subseteq G, B \subseteq M$, A consists of all objects of G, which meet all attributes of B and B consists of all attributes of M which are met by all objects of A. A is called the extent and B the intent of the concept.

A concept of the contour classification context is given by $(G1, G2, / a1, c1)$ (see figure 4). Every contour $(G1, G2)$ meets all the attributes $(a1, c1)$ and vice versa all contours which are characterized by the classification attributes $(a1, c1)$ are given by $(G1, G2)$.

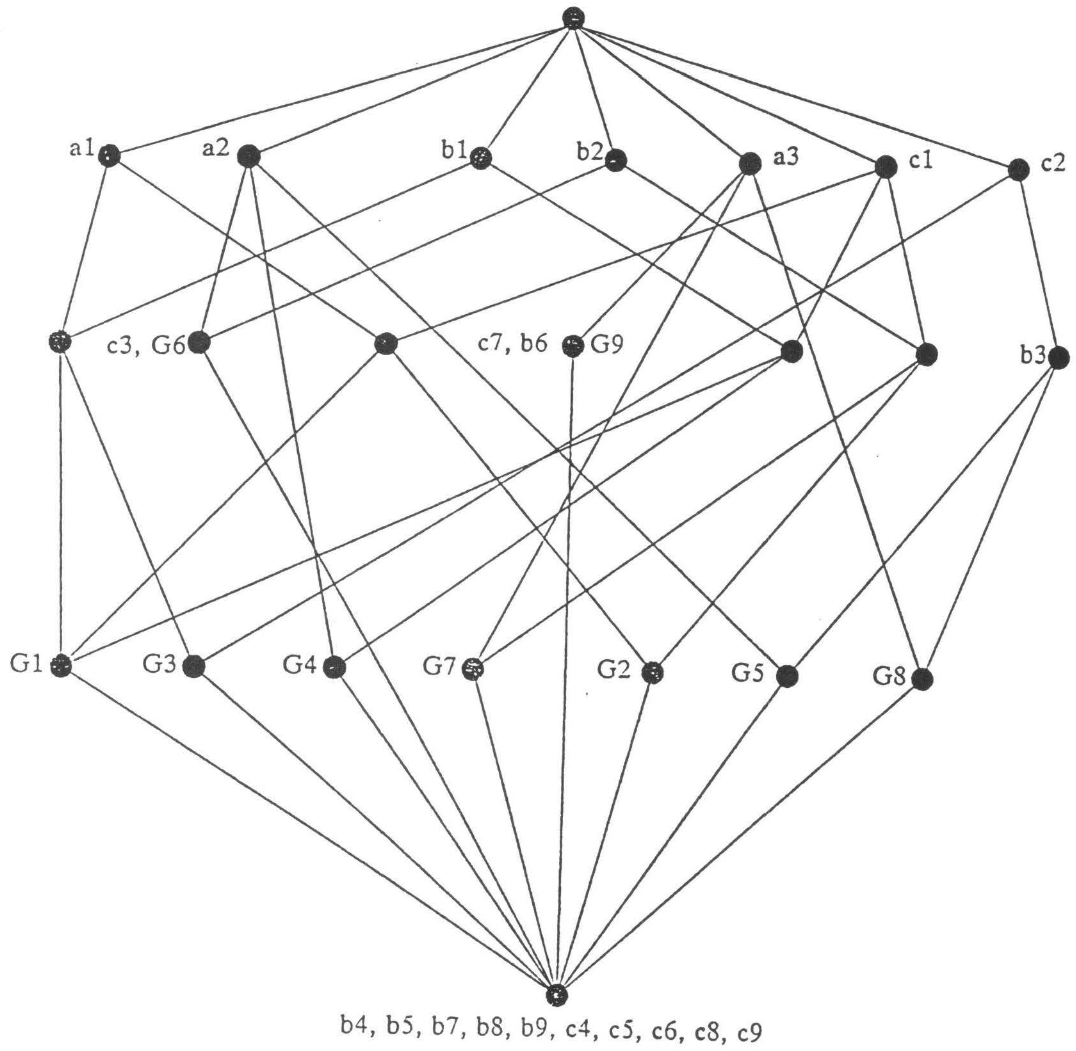
The hierarchy of contexts is captured by the definition

$$(A_1 / B_1) \leq (A_2 / B_2) :<=> A_1 \subseteq A_2$$

for concepts (A_1 / B_1), (A_2 / B_2) of (G, M, I).

In this case we call (A_1 / B_1) a subconcept of (A_2 / B_2) and (A_2 / B_2) a superconcept of (A_1 / B_1). For example $(G1 / a1, b1, c1)$ is a subconcept of $(G1, G2 / a1, c1)$ from the context contour classification (figure 4).

This hierarchy is a natural classification through attributes and makes clear the dependencies of the context. For each context the line diagram is an appropriate possibility to visualize the correspondent hierarchy of concepts. The line diagram (figure 5) shows the hierarchy of concepts for the context contour classification. Each vertex of the line diagram represents a concept of the context. Two concepts (A_i / B_i) and (A_j / B_j) are connected by an upward running line segment starting at (A_i / B_i) and ending at (A_j / B_j), if (A_i / B_i) is a subconcept of (A_j / B_j) and there is no concept of the context which is a superconcept of (A_i / B_i) and a subconcept of (A_j / B_j).



G1 - G9: 9 different parts (for turning)

a1 - a3: attributes which characterize the class of the part

b1 - b9: attributes which characterize the outside shape of the part

c1 - c9: attributes which characterize the inside shape of the part

Figure 5: Line diagram representing the context in figure 4

In order to get a clear arrangement of the line diagram, the vertices are not labelled with the concepts of the context. In spite of that the vertices are labelled with the names of the objects and the attributes in such a way, that the context itself, the concepts and the hierarchy of the concepts can be seen in the line diagram. The object has the attribute *m*, if the vertex which is labelled with *g* is labelled with *m*, too, or if there is an upward running polyline that starts at the vertex labelled with *g* and ends at a vertex labelled with *m*. The extent (intent) of a concept which is represented by a vertex *v* consists of all objects (attributes) with which vertices are labelled that can be reached with downward (upward) running polylines starting at *v*. The hierarchy is represented by polylines. For example, the vertex in figure 6 labelled with *a*1 represents the concept with the extent *G*1, *G*2, *G*3 and the intent *a*1.

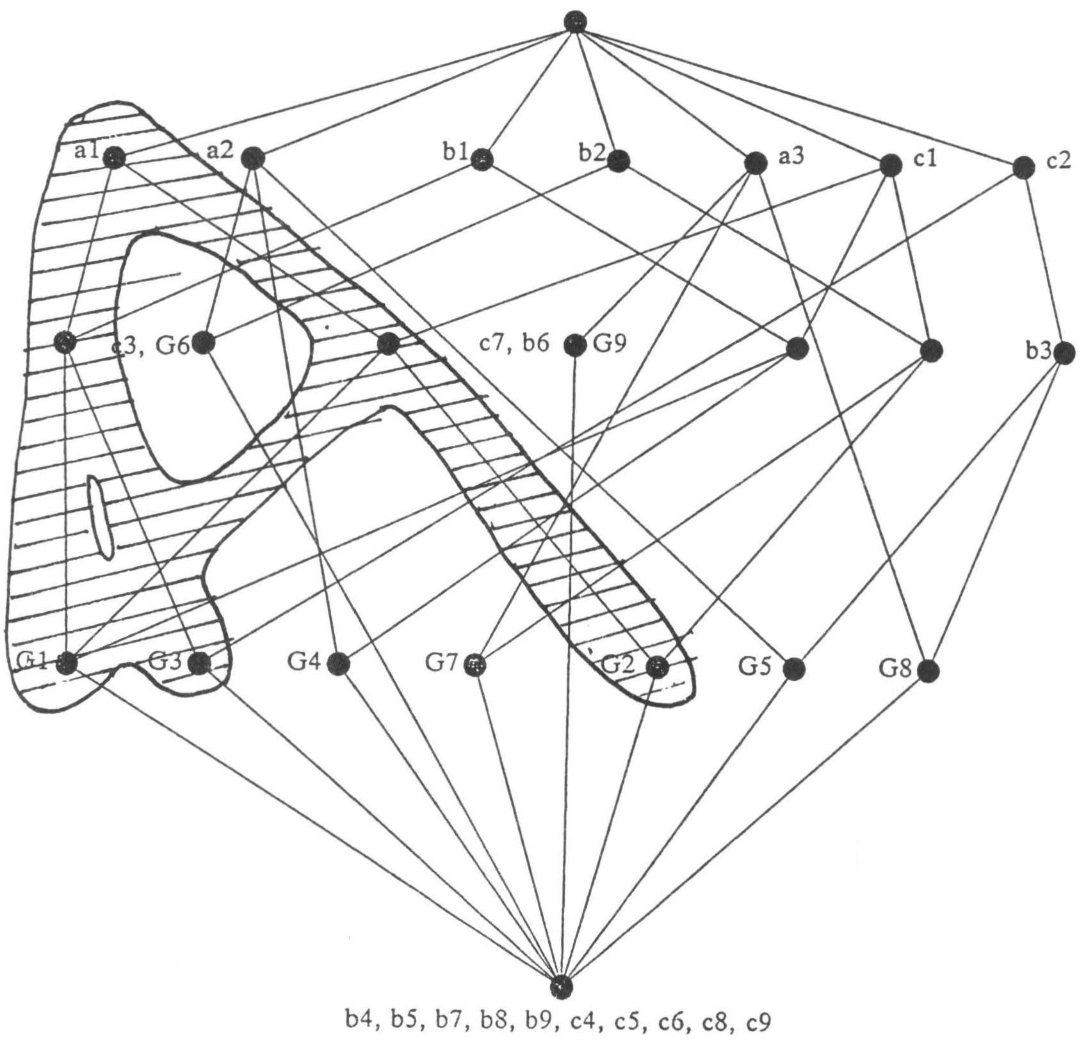


Figure 6: Marked concept (G1, G2, G3 / a1). This means if we choose the attribute a1, we get the information, that every part G1, G2, G3 meets a1, and all parts which are characterized by a1 are G1, G2, and G3.

The line diagram of the contour classification context is a structure to facilitate the search of variants. The processing specification of classification attributes, which characterizes the desired variant, corresponds in the line diagram with a downward running polyline, the steps of which are the concepts determined by the already specified attributes. In the course of the interactive search process the last reached concept is labelled by a rectangle surrounding the corresponding vertex (or highlighted in any other way). At the beginning of the search process the greatest concept of the context (represented by the vertex at the top of the diagram) is labelled by a special symbol. No attributes are specified yet and all variants are in competition with each other. The already chosen attributes can be shown in a different window displaying the contour elements corresponding to the attributes.

Therefore, the formal concept analysis is a valuable method for building expert systems. Here is a summary of the advantages of the methods of formal concept analysis:

- based on a mathematical theory,
- a clear, uniform concept for knowledge base structures and their visualizations,
- modular structured knowledge base by contexts,
- individual visualization of knowledge by line diagrams which makes decisions transparent and illustrates knowledge,
- no alternative questions, all attributes compete with each other.

6. References

[1] T. Batz, P. Baumann, D. Köhler: A Data Model Supporting System Engineering; Proc. of the 12th Int. COMPSAC, Chicago, Oct. 1988

[2] H. Baumeister et al.: Smalltalk-80; Informationstechnik - it, 29. Jahrg., Heft 4/1987, 241-251

[3] P. Bernus, P. J. W. ten Hagen, P. Veerkamp, V. Akman: IDDL: the language of a family of intelligent, integrated and interactive CAD systems (IIICAD); Second Eurographics Workshop on Intelligent CAD Systems, 1988

[4] W. Budde: Arbeitsablauf- und Werkzeugermittlung für die Drehbearbeitung. Ein Beitrag zur Automatisierung der Fertigungsplanung; Dissertation, RWTH Aachen, 1970

[5] W. Eversheim, A. Diels: Neutraler Technologiebaustein für die Drehbearbeitung (Automatische Bereitstellung von INFOS-Schnittdaten bei der NC-Programmierung); Rechnerunterstützte Konstruktion und Planung, Carl Hanser Verlag, München, 1988

[6] A. Goldberg, D. Robson: Smalltalk-80; The Language and its Implementation; Addison-Wesley, Reading, 1983

[7] A. Goldberg: Smalltalk-80; The Interactive Programming Environment; Addison-Wesley, Reading, 1983

[8] D. Köhler, T. Batz, P. Baumann: Modellierung und Darstellung graphischer Datenstrukturen in PRODAT; GI Fachgespräch: Non-Standard Datenbanken für Graph. Anwendungen, 1988

[9] M. Lutz: Computergraphik von Begriffsverbänden; Forschungs- und Arbeitsbericht, FG Graphisch-Interaktive Systeme, TH Darmstadt, 1987

[10] R. Opferkuch: Auswahl und Ausbau eines NC-Programmiersystems unter besonderer Berücksichtigung der CAD/CAM-Verbindung; Werkstattstechnik 78(1988), 101-105

[11] H. Opitz: Die richtige Sachnummer im Fertigungsbetrieb; Girardet Taschenbücher, Technik Bd. 2, 1971

[12] U. Pilland: Tendenzen in der NC-Technik; Werkstattstechnik 78(1988), 299-304

[13] Prospero M.J., Messina L.A., Towards the construction of graphical interfaces on the basis of geometric models, Proceedings of Eurographics'86, Lissabon, North-Holland, 1986

[14] Prospero M.J., Estilo declarativo na Programaçao Grafica Interactiva: analise e avaliaçao sobre sistemas em Prolog, Dissertationsarbeit in Vorbereitung 1988

[15] B. Veth: An Integrated Data Description Language for Coding Design Knowledge; First Eurographics Workshop on Intelligent CAD Systems, 1987

[16] R. Wille: Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts; Preprint, TH Darmstadt, 1981

[17] R. Wille: Bedeutungen von Begriffsverbänden; Preprint, TH Darmstadt, 1987

7. Appendix

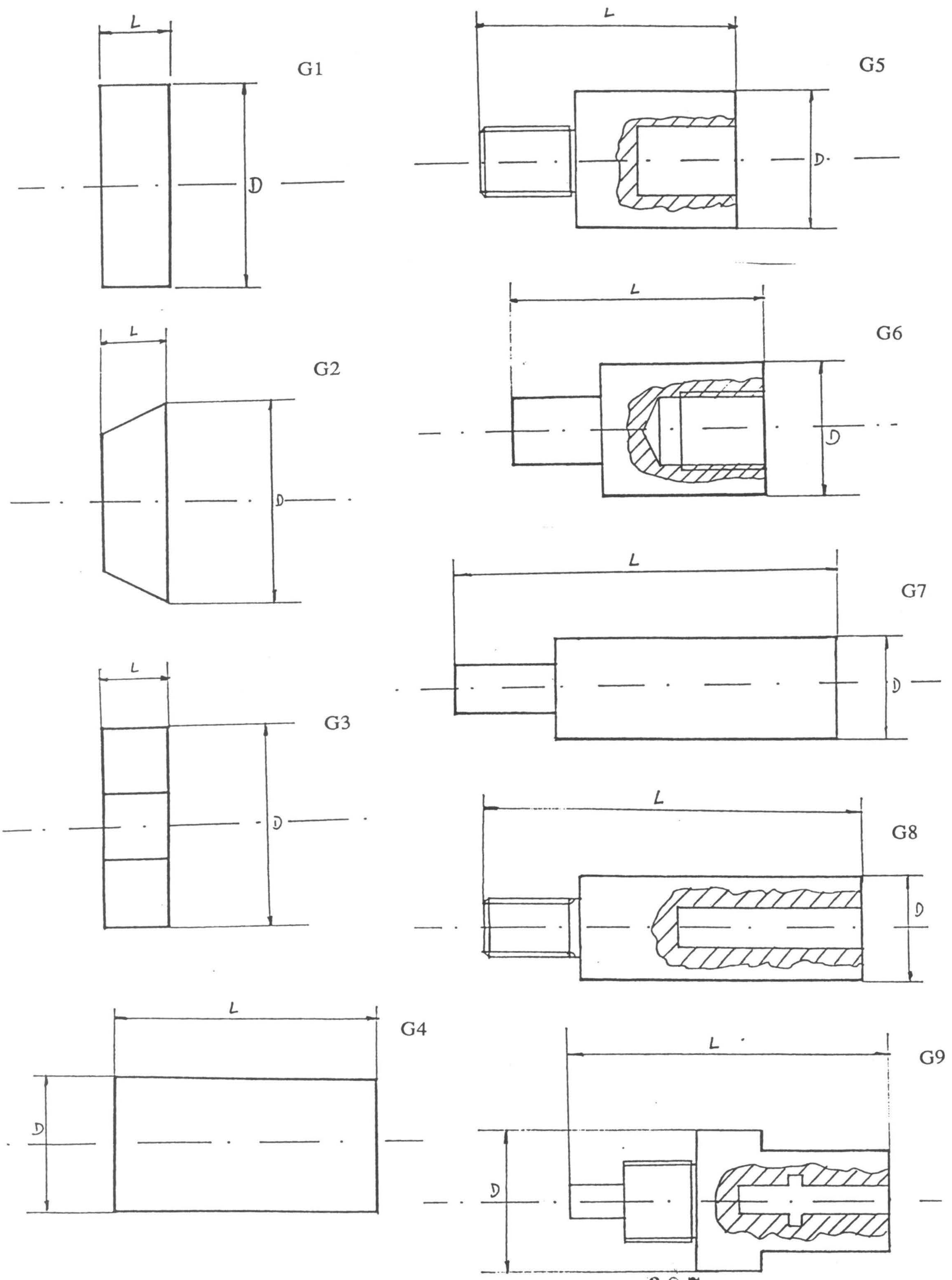


Figure 7: Shapes of the parts mentioned in the examples before

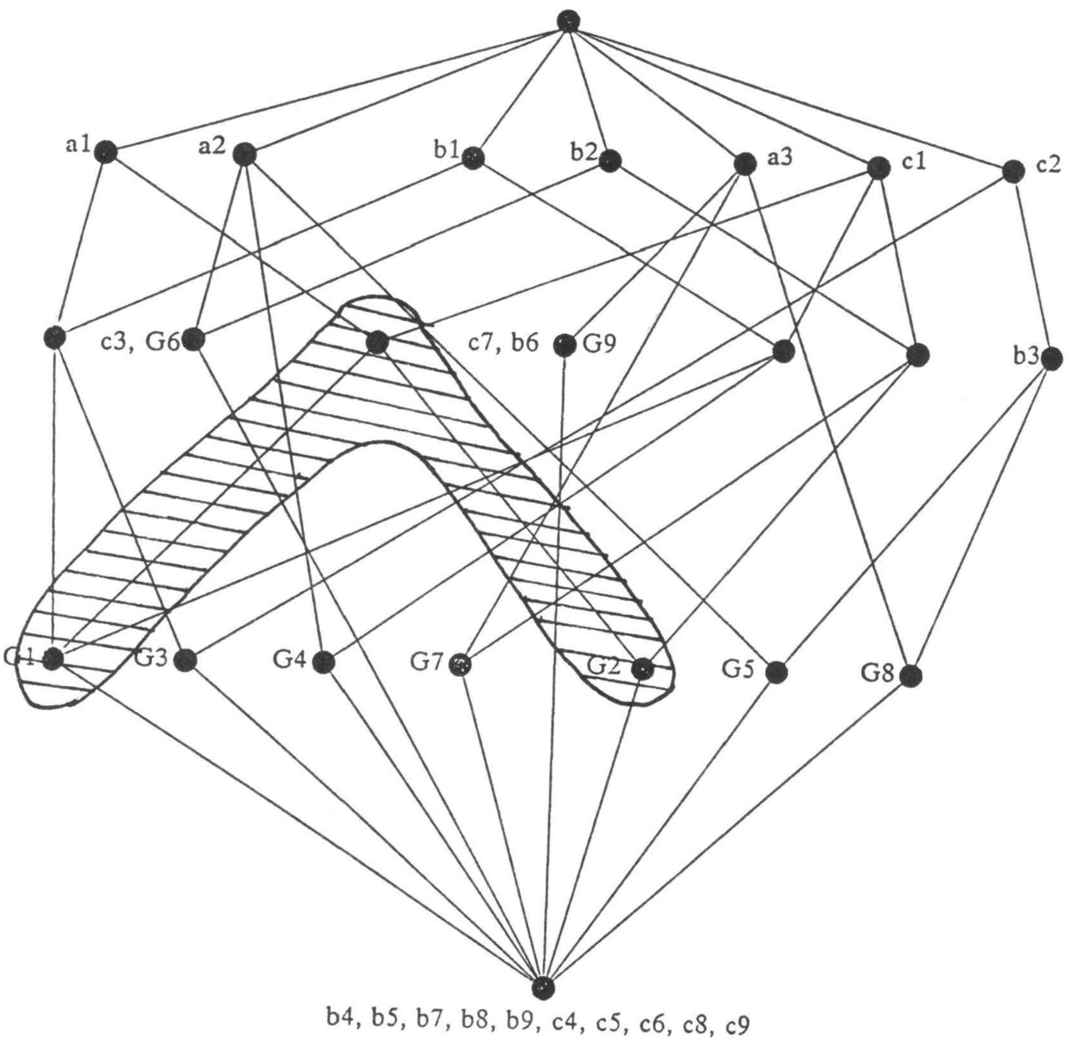


Figure 8: Marked concept (G1, G2 / a1, c1). It doesn't matter if we choose a1 first and c1 next or vice versa, the output (graphical representation of the result) is the same. Please remember the tree structure in figure 3, where we were forced to choose first a_i , then b_i , and last c_i .

Zsófia Ruttkay, Paul J.W. ten Hagen

Intelligent user interface for intelligent CAD

INTELLIGENT USER INTERFACE FOR INTELLIGENT CAD

Zsófia Ruttkay

Computer and Automation Institute,
Hungarian Academy of Sciences,
P.O.B. 63, H-1502 Budapest Hungary

Paul J. W. ten Hagen

Centre for Mathematics and Computer Science,
Department of Interactive Systems
P.O.B. 4079, NL-1009 Amsterdam The Netherlands

Abstract The user cannot fully exploit the capabilities of an intelligent CAD system unless the user interface is equally powerful. In the paper we list the general and design-related criteria of intelligent communication. The emphasis is on the semantics of the inputs/outputs in context of the communication, the design process and the design constraints. We give a proposal how to extend a traditional user interface with a monitor component in order to meet the requirements of intelligent communication. The proposal can be an implementational model for the communicational tasks of intelligent CAD.

1. INTRODUCTION

1.1 User Interface Issues of Intelligent CAD Systems

Attempts to improve CAD systems by applying techniques from the domain of artificial intelligence have addressed almost every aspect of CAD. This paper is concerned with ways to make the user of an interactive CAD system aware of such improvements. It is obvious that increased capabilities must become accessible through the user interface in order to be useful. Providing access to the various new CAD system functions is far from trivial, in fact it poses several sofar unsolved problems.

In order to describe the size of the problem we will describe some of the most important issues in intelligent CAD and the corresponding user interface issues that follow from them.

An ICAD system should be knowledgeable in several ways:
- It maintains different models of the object being designed, such as, functional, structural and geometric models.
- It represents the evolution of the design, the partial designs and alternatives in the various design stages.
- It handles domain specific knowledge for instance about constraints among design elements.

Logic and object oriented programming methods provide a more powerful symbolic computational aid to provide such a variety of abstract views on a model. They also can provide better means to represent the relations between all these variants of the design object. Moreover, this can be done in one integrated system.

Nevertheless, these systems still require the user to be in control. The CAD system is a (slightly) better assistant, provided that the user can easily express his commands. An important function of the system with this more complex representation of the design object, is to automatically select relevant information. There is always a limited capacity to visualise information, hence no resources are to be wasted. Moreover, showing relevant information is a sure sign that the system understands the user.

These characteristicts of Intelligent CAD (ICAD) raise novel user interface issues. They all are concerned with presenting (more) semantic information in addition to illustrating the design object. This immediately results in a richer dialogue language, because everything that can be

presented can also be discussed.
Below we will list a number of these issues. Their relation with the improved functionalities given above is quite obvious.

Concerning the topic of conversation:
- Means must be provided to define and present relations in and between models.
- Data items must be annotated with qualifiers such as age, importance and precision. The user interface can be made responsable for providing some of these automatically.
- The interface must be able to handle several design object presentations, either simultaneously or sequentially.

Concerning the flow of the dialogue:
- Embedded sub-dialogues and multiple threads in dialogues must be supported. For instance, the system can enter an extra dialogue for maintaining consistency.
- An ICAD system is used by expert users. They must be allowed to tune the system to their way of working. An untuned system is (by definition) what a non-expert user gets.
- The system must be capable of visualising the consequences of user inputs, including the reasons, e.g., domain specific reasons.
- The user interface must be able to record history, in particular to explain when and why decisions were made.

Concerning the language of communication:
- The mapping between CAD operations and user language may change as the dialogue (and the design) evolves. The mapping can be many to many.
- The vocabulary and presentation methods may also change.

In this paper we will investigate how these requirements can be met by primarily looking at the user interface itself. This is of course not the only aspect to be considered. However, the result of our analysis might be that if the CAD system is capable of improved functionality, than the user interface can convey it.

1.2 A Separate User Interface Component.

Application programs which are heavily dependent on user interaction, nowadays have a separate component for the user interface. Such a component can be based on a specialized system, called the user interface management system. It embodies a method to abandon poorly designed ad hoc user interfaces. It provides a rich set of basic interaction techniques, including window management functions.

A user interface management system (and the separate interface module) provide a number of advantages which will be heavily exploited to get an intelligent interface, e.g. an interface which can also use AI techniques.

- A clear model of interaction. All user interface functions are present in one environment. Hence, all combinations are in that environment as well. Thus a selective set can be provided for a given task, providing consistency in language across tasks, etc..

- Device independence. Every basic interaction technique can be choosen according to the given hardware. Above this basic level all device specific properties are shielded away.

- Interaction style. To some extend the interactions can be shaped according to individual user (group) requirements. For instance each can make its own screen lay out arrangements.

In the case of ICAD systems there is one major capability to be added to the user interface. It is the ability to access and process semantic information from the aplication directly in the user interface, for no other purpose than to moderate the interactions themselves. Thus, syntax and semantics get coupled visibly and more directly.

This capability will form the basis for many improvements in the interface itself. It constitutes a bridge between the application and the user interface. The reason for this bridge is twofold:

- In order to decide what is to be displayed, both the application status and the user inputs have to be considered simultaneously.
- Maximum directness of interaction also requires simultaneous interpretation of both user input and application status.

In the next sections we will first outline what kind of functionality can be improved by or provided by this capability, either from the point of view of general user interface properties or from the point of view of the ICAD functionality. It is up to the builders of ICAD systems to achieve the goals of more intelligence in the CAD interface exploiting the bridge function. This is similar to the view how to use a user interface manager. It provides means for building good interfaces, but guarantees nothing for any particular interface build with it.
After this overview an extension of the user interface manager will be introduced. For the sake of clarity all new functionality associated with the bridge function will be provided in yet another separate module called the user interface monitor.

The topic of our investigation is a user interface, which provides the functionalities required by the communicational tasks of intelligent CAD systems. We will refer to such a user interface as an intelligent one, in contrast to a traditional user interface. In Chapter 2 we discuss the criteria of intelligent communication in general, then in Chapter 3 the additional ones stemming from the functionalities of intelligent CAD systems. In Chapter 4 we describe our proposal for an intelligent user interface. Finally, in Chapter 5 some examples are given to present the intelligent functionalities.

2. General criteria of intelligent communication

2.1. Efficiency

The user interface should be able to 'understand' the user requiring of him the least possible input. The information not supplied by the user should be automatically added by relying upon

- common-sense application knowledge (e.g. defaults), or
- the content of one or more previous inputs (e.g. the line width to be used can be the latest one, or can be derived from the environment of the line to be drawn).

The user interface should be co-operative in two ways:

- it should be active in understanding the user's inputs 'a posteriori',
- it should anticipate the user's intentions and offer input tools and tokens which are most likely to be used by the user.

2.2. Directness

The user would like to have a feedback indicating how his input has been understood. Not only the syntactic correctness of the given input should be echoed, but the input should be interpreted in the context of what has been specified so far, and the possible consequences should be visualized (e.g. grouping, generating further outputs or modifying previously defined ones). The direct feedback from the user interface has two advantages:

- the application is relieved from certain semantic checks and updates,
- the user can rely upon the information supplied by the direct feedback while giving the input.

2.3. Deviations from the syntax

An intelligent user interface – similarly to communicating humans – should be permissive and motivated to understand, with the ability to cope with syntactically not perfect inputs. On the lexical level, typing errors or the misuse of input devices and the intended usage should be detected. On the level of syntax and semantics, the correct and relevant inputs should be identified. Decision should be made if a syntactically not correct input should be rejected, or it should be forwarded to the design system to attempt to interpret it.

2.4. Adaptation to the user

The user of a computer system may have other preferences of communication style/media as well as domain knowledge and experience than the system at hand. The user interface should presuppose the user to be of a certain type, but it should be able to adapt itself to the user. The characteristics of the user can be either asked for, or can be inferred from the flow of communication. The user interface could be tailored with ease to the needs of a given environment (e.g. accepted vocabulary and conventions in communication and design).

3. Typical communicational requirements of intelligent CAD

3.1. Different aspects of the same design in parallel

The aim of an intelligent CAD system is to support the designer - or a group of designers - throughout the entire design process. Typically, at a given stage of the design the user will address only one aspect of the design directly. He may, however, be interested in the consequences of these actions for other aspects or parts. The user interface is to provide and maintain additional views either on the basis of built-in design process knowledge or on the basis of user-directives.

3.2. Alternative designs

A similar multiple-view mechanism is needed for presenting alternatives. However, the context for alternatives is entirely different from the previous one: the same view of different candidate design alternatives should be provided. A user interface should have the capability of differentiating between common and specific features of alternatives, and support the handling of alternatives (e.g. comparing them, exchanging values between them, supplying simultaneous values).

3.3. Partial designs

In course of the design process, partial designs exist all the time. The design process is not guided strictly by the CAD system, the user has freedom in elaborating a design, so it cannot be predicted which variables will be given by the user and in what order. Means should be provided to visualize all the possible incomplete designs using the method given for the presentation of the complete one. Those parameters for the visualization which are not available because of certain design variables have not been bound yet, should be supplied by the user interface. The user interface could use a strategy for adding the missing parameter values, such as:

- using default design variables only for presentational purposes, and compute the parameters accordingly,
- using a specific parameter value to visualize if the corresponding design value has not been given.

The user interface should differentiate between a default value used for the visualization of an unspecified variable, a default provided by the design system and a value which has been given by the user or the design system.

3.4. The validity of variable values

Intelligent CAD systems aim at conforming to the practice of human designers by allowing explanatory usage versus decision making. A variable instantiation can be temporal, for the purposes of 'what - if' explorations or should be taken as a design decision. In the latter case, the value specified is either allowed to be modified by the user or by the design system or has been specified for once and all. The user interface is expected

- to provide different visual feedback for information with different status, and
- to define the status of a variable by reasoning from the status of other variables and the stage of the design (e.g. at the preliminary design stage certain variable instantiations should be treated as suggestions).

3.5. Approximate variable values

Human designers frequently use approximate values and vague statements to specify, evaluate and modify a design (e.g. 'bright kitchen', 'slightly corrosive environment', 'about this size'). The user interface should be responsible for the visualization of approximate values and vague statements. The problem of providing parameters for visualization is similar to the one mentioned in connection with partial designs. The user interface should treat the approximate values as constraints for for the precise value to be given later on.

3.6. Presentation of constraints and relations

A new requirement of the user interface is to support the presentation of -

possibly dynamically changing – constraints, in order to inform the user about the feasible inputs at a given stage of the design. This functionality provides two aims:

- to aid the user to give semantically correct input, by indicating the actual range constraint for the input being given,
- to inform him about the source of the constraint for the input in question, that is the relation which will not hold if the input is out of range.

The user interface should infer that a constraint must either be fulfilled, or can be violated at the expence of modifying the value of some other variable. The design status and the value status of the variables in the relation in question can be used for this purpose. This support should be interwoven with the inputs by the user, and should be presented in an efficient way, preferably using the medium and form of presentation of the user's input.

As the user knows about the input value expected, it can be assumed that he will give a correct one, and if he should not, then he is doing it on purpose. So the power of the user interface should be in giving information dynamically rather than in being armed with value checks.

3.7. Explanation, modification in context

When using a CAD system, the user would like to gain information about why certain variables have a given value, how the current design has been generated. The user also would like to have support for modifications: when changing one of his previous inputs, all the implied changes should be automatically generated.

A related problem is to indicate conflicts: the user should be informed about the previously defined variables in conflict, and request one of them to be modified. The strategy for choosing the variable to be changed can be based on the recency, design and value status of the variables in question.

In order to fulfill these tasks, the user interface should store the history of the inputs/outputs, and should maintain how they are related. It has to have a mechanism to select those previous inputs which are relevant to a question and to give the answer in a form digestible for the user.

3.8. Identification of parts

Experimental applications of intelligent CAD have drawn our attention to the problem of the identification of parts – generated by the user or by the system. The user should not be forced to use the identification method of the design system – e.g. codes, internal names –, but the user interface should be responsible for user friendly names for new instances: variants of an instance, further instances of a given type, or an instance derived from other instances (by e.g. geometric operations, assembly). It is a common practice of the human designer to identify entities intensionally, that is by giving the relations which should hold for the entity. So the user interface has to rely upon two sources in generating names:

- a given vocabulary for types, attributes, values, and instances (e.g. room, size, small),
- strategies to generate or choose names according to the intensional description of objects (e.g. the living room is the biggest one).

In addition to name generation, the user interface should be able to identify objects by the names and descriptions supplied by the user. The user interface should provide access to objects by – even partial – description of variables, interpreting it as search patterns. A rich choice of patterns should be allowed (e.g. the rooms with one window, the room without balcony). By this access mechanism the user is not forced to refer to an object by the same, unique name all the time.

4. Proposal for an intelligent user interface

Our aim is to provide a user interface powerful enough to support the communicational requirements we have discussed. These requirements have the following consequences for the model to be applied:

- The user interface should form a *separate module*; the user and the design system should communicate exclusively via the user interface.

- The user interface should have some understanding of the conversation between the user and the system, not relying upon syntax only. Specific and more overall *application domain knowledge* and some *discourse knowledge* should form the basis for the understanding capability.

- The routine communicational tasks should be carried out by a traditional user interface. In this way an 'intelligent' user interface can be considered as the *extension of a traditional one*, which is very convenient from both the technical and the conceptual points of view.

Here we do not discuss the problems related to devices, interaction techniques and human factors. It is inevitable to examine thoroughly which are the best means to present additional aspects of the communication over and above primary value exchange. Presumably much effort will be needed to offer friendly tools for 'sloppiness'. Theese tools should not – and probably, could not – be slavishly adoptated from traditions in human communication or in an engineering discipline. A new consensus should be reached with regard to exploiting fully the arsenal of presentation and interaction techniques (e.g. use of different media, colours, highlighting, blinking, animation). As other issues are in the focus of this paper, for demonstrational purposes we will use simple and somewhat 'ad hoc' means – namely: colours , line-type and line-width – to indicate the different aspects of communication.

4.1. The basic concept: a traditional user interface guided by a monitor

We propose an experimental environment to characterize the facilities of an intelligent user interface more precisely. Our model also demonstrates how an intelligent user interfaces can be implemented fulfilling the three criteria above. Our model consists of two components:

- the **service user interface**, which is a traditional user interface to accomplish the routine communicational tasks,

- the **monitor**, to 'intelligently' supervize the operation of the service user interface.

The monitor – relying upon additional information over and above the value of input/output variables – supervises the functioning of the traditional user interface and interferes in certain situations. The cases when the monitor should and could interfere are of two kinds:

- the traditional user interface 'asks for' the help of the monitor (e.g. in the case of erronous data),
- the monitor identifies a situation when it can interfere.

As a result of the monitor's interference, data for the service user interface are modified or generated, and the interaction continues in a way different from what would be prescribed by the traditional user interface alone.

4.2 The Formal Model

4.2.1 The service user interface

The service user interface is based on the Dialogue Cell system [3,5]. This is user interface management system which allows for three-dimensional graphics objects to appear as tokens in dialogues. The basic notions of the service user interface are the units of communication acts described by so-called Transaction Cells. A transaction cell reads and writes symbols while active and ultimately generates its result symbol.
The symbols communicated by a transaction cell are exchanged with named processes after having established a communication with them. A special set of these processes are the slave transaction cells which have been activated by this (parent) cell. Slave cells are deactivated when their parent is deactivated. Some of the descendant cells are basic input and output processes to directly communicate with the user.

A transaction cell can be characterized as a three component unit:
- transaction cell name, and symbol type;
- activation and connection rule;
- read/write grammar rule.

The transaction cell name is an external name to be used by all other units (including the parent) who wish to communicate.
The activation rule specifies all descendant transactions that need to be activated, as well as already active processes with which a communication link must be established. It also precisely describes when during the active life time of the transaction cell these links must exist. Note that this constitutes the basic mechanism to specify when a user is allowed to perform what action. The activation rule allows for simultaneous as well as sequential connections.

The read/write grammar rule is a regular expression specifying in what order reads and writes may happen. It allows for sequential as well as parallel reads and writes. The specification of the reads and writes is entirely symbolic. The actual values being communicated follow automatically from the real time values produced by the various processes including the basic input output processes.
A mapping from, say, input to an internal representation, which is communicated further, is also specified through a write-read sequence to/from the mapping function-process. Equally the visualisations occuring during interactions are taken care of by graphics processes connected to a transaction cell.

In this way we have a model for the user interface service component which abstracts from all process details as long as a number of abstract communication functions exist, being: read, write, activate, deactivate, connect and disconnect.
In figure 1a an 1b a number examples of transaction cells are given. The syntax is almost self-explanatory.

Without going into further details, we list the components of the service user interface (fig. 1.):

- the **symbol pool**, where the symbols to be supplied or supplied but not having been 'consumed' by a cell are stored;
- the **library of communication cells**, where the definition of communicatuin cell prototypes are given;
- the **pool of active cells**, to describe the present status of the communication;
- the **scheduler**, responsible for:
  - assigning parameters and symbols to communication cells,
  - updating both the symbol pool and the pool of active cells,
  - handing over certain symbols to the monitor.

4.2.2. The monitor

The monitor is a rule-based system which interprets the symbols in the context of:
- the *history of the communication*, that is what has been told, and in what way,
- the *semantic constraints* prescribed by the application domain.

The monitor has the following components:

- The **symbol working memory,** where information on symbols requested or having been supplied is stored. While in the symbol pool data fitting into the scheme of the traditional user interface is present, in the symbol working memory additional information about certain symbols of the symbol pool is registered. The entries of this working memory are of the following form:

WME = < symbol_name, V, VS, DS >

where: - symbol_name is the unique name of the symbol,
- V is the value of the symbol,
- VS ::= requested | default | approximate | precise,
- DS ::= explanatory | suggested | confirmed.

The value status, VS indicates whether the value for the symbol is to be given, or it has been supplied either by the user interface as a default or by the designer as an approximate or precise value. The design status, DS indicates that the value is either for exploratory purposes only, or is to be stored as a variable value for a design alternative. In the latter case it is either allowed or not to alter the value as the design proceeds.

- The **symbol history,** where the symbols consumed by a communication cell are stored, in the same form as the entries of the symbol working memory.

- The list of **design relations,** expressing design knowledge either as default values for certain symbols or relations describing the feasibility of the design. The relations are of the following forms:

DEFR= < default, symbol_name, DV >
DEFR= < default, symbol_name, CF >
DISR= < rel_name, SL >

where: - DV is the default value for the symbol,
- CF is a function of symbols,
- rel_name is the unique name for the relation,
- SL is a list of symbol names.

For each symbol, the relations which have the symbol as one of the variables can be listed. If each symbol in a relation has a value, then the relation can be checked. If all but one of the symbols in a relation have been given a value, then a value or a constraint for the free symbol can be supplied.

- The set of **monitor rules** of the following form:

RULE= < CONDS ; ACTS>

```
where  – CONDS ::= COND [,CONDS ]
       – COND  ::= symbol_working_memory_pattern |
                   history_pattern |
                   relation_pattern
       – ACTS  ::= ACT [,ACTS]
       – ACT   ::= add_delete_symbol | handover_symbol |
                   handover_parameter
```

The left-hand side of the rules decides about when to interfere, while the right-hand side prescribes what to do. Besides updating the symbol working memory and the symbol history, symbols and parameters can be handed over to the symbol pool of the service user interface. The patterns and actions as well as the inference engine responsible for rule selection and firing are discussed in [9]. The rules extend the traditional user interface functionalities in three ways:

- by generating symbols which would be required of the user,
- by supporting information exchange between cells other than the producer and the consumer of a symbol,
- by providing parameters for cells to visualize other aspects of the communication than data echange.

5. A design example

The task is to design a rectangular dining table for a given number of persons. We concentrate on the design of the board. The user is expected to give the number of persons first, and then to give the dimension of the edges of the board. Neither of the edges should be shorter than 60 cms. The only design constraint dealt with on the user interface level is that there should be at least 50 cms allotted to each person.

This design task is a very simple one, all the same many communicational requirements arise which demand an intelligent user interface: the human designer makes approximations, several trials, probably looks at alternatives, and finally elaborates one of them by giving the precise, final dimensions. He also uses intensional references and different media.In the followings, examples will be given to demonstrate, how some of these characteristics can be supported by the proposed intelligent user interface.

The following symbol names are used for simple or compound symbols:

- *corresponding to design variables*

pers:	the number of persons; even positive integer.
length:	the length of the table in cms; integer.
width:	the width of the table in cms; integer.
table:	the table being designed, (length, width).

- *for presentation entities*

p1, p2, ... :	points in the coordinate system of the service user interface; pair of reals.
line1, line2, ... :	lines; pair of points.

- *for visualization of presentation entities*

line-width:	thin | thick
line-type:	continuous | free-hand | dotted | dashed

The application domain knowledge is given in the form of:

- *relations expressing design constraint*

```
<edge, length, width, pers > :[length/50] + |[(width - 100)/50]|  ≥ pers/2
<minwidth,         width >    :  width ≥ 60
<minlength, length>           :  length ≥ 60
```

- *defaults*

```
<default, length,  max (100, max(60, pers *25)) >
<default, width, min(100, max(60, pers*25)) >
```

About the interaction techniques: the **length** and **width** symbols – according to the design variables– can be given either by typing in values to indicated places of the screen, or by drawing one of the edges of a rectangle. X The hierarchy of communication cells is explained in fig.2. The cells use parameters for visualization of symbols according to the conventions in fig. 3.

The service user interface provides that the communication should be syntacticly correct, both concerning the interaction techniques to be used and the type and range check for input values. The current range constraint for **length** and **width** – provided by the monitor – is used as parameter for the appropriate cell, which provides visual feedback accordingly.

5.1. Default for a symbol requested

In order to supply a default value when a symbol is requested, the service user interface reports to the monitor if a symbol is requested, in addittion to putting the request on its own symbol pool. The monitor has the following rules to supply the default – if exists – for the traditional user interface:

```
if < X requested > and <default X, Def >
  then write_to_symbol_pool ( <X, Def>, <line-type,dashed>,
                                         <line-width, thin>)
        and remove (< X requested >)
        and add (< X Def default suggested >)

if < X requested >
  then remove (< X requested >)
```

When the symbol **pers** is requested, no default will be supplied, the user has to give it. Let's suppose he has given 6. When the symbol **length** is requested, the default value of 150 will be computed and registered to the symbol pool with values for the **line-type** and **line-width** parameters to be used for the visualization of the **length** symbol by the traditional user interface. The same happens for **width.** The drawing generated accordingly can be seen in fig. 4.

5.2. Indicating constraints

Whenever a symbol is requested, the monitor looks up the design constraints in which the symbol appears as a variable. Knowing the value and design status of all the symbols, the monitor filters out those relations, which have only the requested symbol as unspecified. The actual constraint for the requested symbol is computed, and the constraint is visualized.

In our example, the monitor provides the following constraints for the **width** symbol, assuming that **pers** is 10, and according to what has been specified so far for **length**:

length value	no value	200 (approximate or precise)
suggested	width≥60, compulsory	width≥150, non compulsory
confirmed		width≥150, compulsory

The compulsory constraints must be respected by the user in order to produce a feasible design, while a non compulsory one can be violated, at the expense of some other symbols in the constraint which have been suggested, but not confirmed yet.

Whenever a constraint is computed, the monitor hands over it as a parameter for the service user interface. The communication cell that will be activated to produce the symbol will use these parameter values for visualizing the constraint as long as the symbol has not been given. In the case of a compulsory constraint, the service user interface will force the user to give a value not violating the constraint (see fig. 5).

5.3. Re-naming as error recovery

A common source of errors is that the user tries to give inputs in a way not acceptable by the user interface. The user has to conform to the restrictions of a traditional user interface. In our case, the service user interface expects a horizontal line to visualize the **length** rather than the **width**. That is when the user draws a horizontal line, the scheduler hands it over to the define-length cell. If the user – either on purpose or by mistake – does not fulfill this expectation, then the service user interface can do nothing but report the error.
Let us suppose, that the user has drawn a horizontal line. The scheduler of the service user interface hands over the line symbol produced to the **define-length** cell, which assigns 150 to **length**. Then the user draws a vertical line, which is consumed by the **define-width** cell. It turns out

that he has given the value of 170 to **width** (fig. 6.a). The **define-table** cell compares the two values, and detects the error. But instead of forcing the user to re-draw the table in the right position, the monitor interferes and changes the cast of the lines. The horizontal line will be picked from the symbol history, and will be handed over to the service user interface with the additional information that is should be consumed by the **define-width** cell. The vertical line from the symbol history will be handed over as when the symbol **length** will be requested, and will be assigned to the **define-length** by the scheduler. As a result, the appropriate names will be assigned to the edges (fig. 6. b.).

5.4. Naming based on intensional definition

The problem we have discussed can be solved in a more adequate way as well: The names used to communicate with the user are assigned to the symbols by the monitor, and handed over as character strings to be visualized by the service user interface. There are rules of the monitor responsible for the name assignment and generation. In our case, let us suppose that the service user interface uses internal names to identify graphical symbols. The monitor has rules to translate these names to names digestable to the user, and vice versa. The semantics of a user-friendly name is given in terms of relations that should hold.
A name should be registered only when either used by the user or requested as a chracter string to be consumed by a cell. A name is not used earlier than the necessary information to bound it is available. Dynamic changes in names – according to modifications of the design – can be maintained by the monitor.

In our case, the length could be defined as the dimension of the longer edges of the rectangle. The modified communication cell hierarchy is given in fig 7, and the communication taking place in fig. 8.

References:

1. Gudes, E., Bracha, G.: GCI – A Tool for Developing Interactive CAD User Interfaces, Software – Practice and Experience, Vol. 17(11), 1987. pp 783-799.

2. Guedj, R. A.: The evolution of style of interaction, Proceedings of CAD'86, London, 2-5 Sept. 1986. pp. 1-6.

3. ten Hagen, P.. J. W., van Liere, R.: A model for graphical interaction, Centre for Mathematics and Computer Science Report CS-R8718, Amsterdam, 1987.

4. ten Hagen, P. J. W., Tomiyama, T. (eds.): Intelligent CAD Systems I, Theoretical and Methodological Aspects, Springer-Verlag, 1987.

5. Van Liere, R., ten Hagen, P.. J. W.: Introduction to dialogue cells, Centre for Mathematics and Computer Science Report CS-R8703, Amsterdam, 1987.

6. Pfaff, G. E. (ed.): User Interface Management Systems, Springer-Verlag, 1985.

7. Proceedings of the Second Eurographics Workshop on Intelligent CAD Systems II, Implementational Issues, Veldhoven, The Netherlands, April 11-15. 1988.

8. Proceedings of the IFIP W. G. 5.2. Workshop on Intelligent CAD, Boston, USA, October 1987.

9. Ruttkay, Zs.: Multi-media Presentation in CAD Systems, Proceedings of the Second Eurographics Workshop on Intelligent CAD Systems II, Implementational Issues, Veldhoven, The Netherlands, April 11-15. 1988. pp. 69-92.

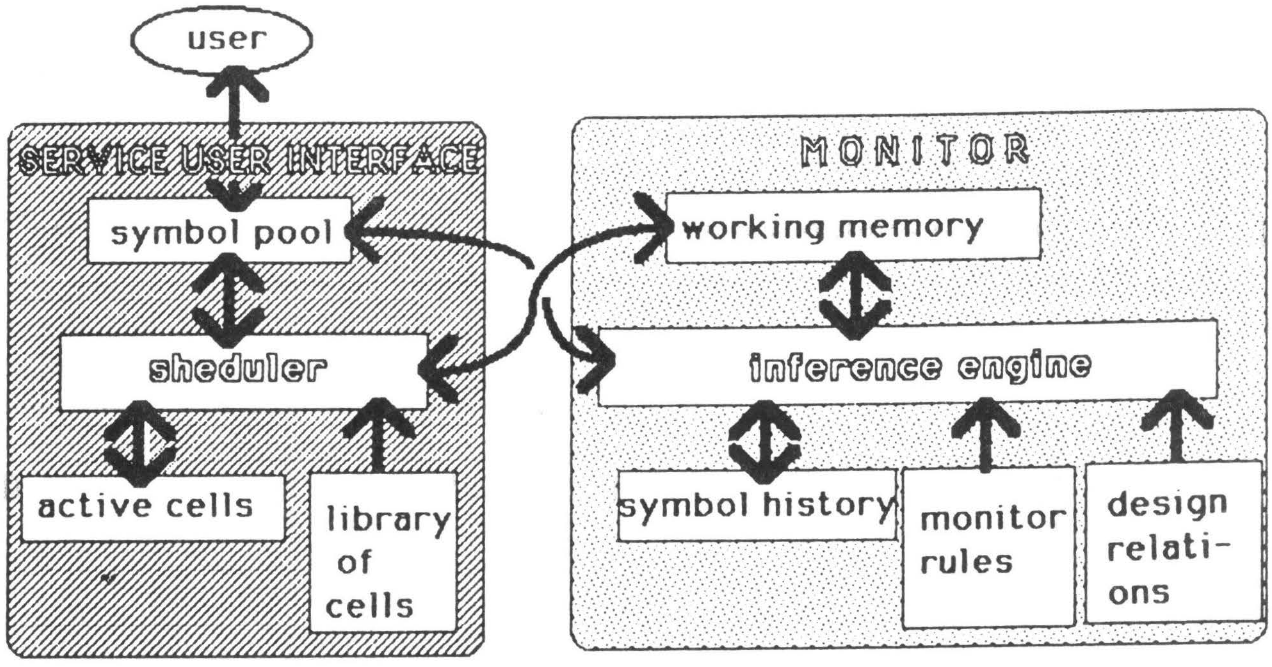


Fig. 1.

The components of the proposed intelligent user interface

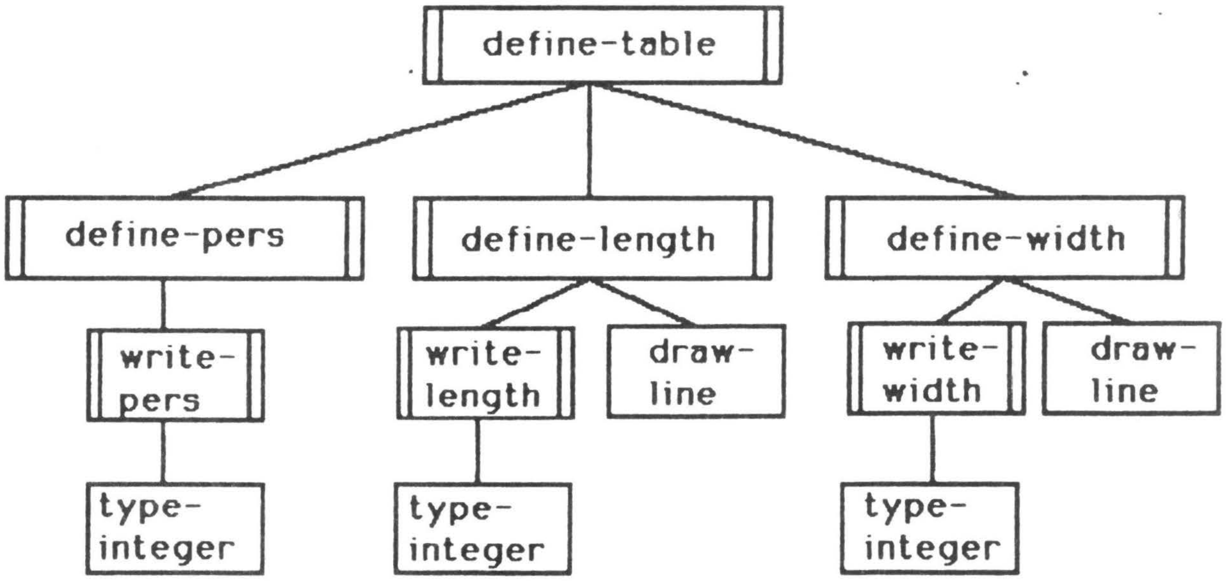


Fig. 2.

Hierarchy of communication cells used for the example

information on a variable	line-type		line-width	colour
value status:				
default	dashed	--------		
approximate	free-hand	▬▬		
precise	continous	———		
design status:				
suggested			thin	
confirmed			thick	
constraint:	dotted	············		
non compulsory		············▷		
compulsory		············▶		
conflict				red

Fig. 3.

The meaning of parameters used for visualization

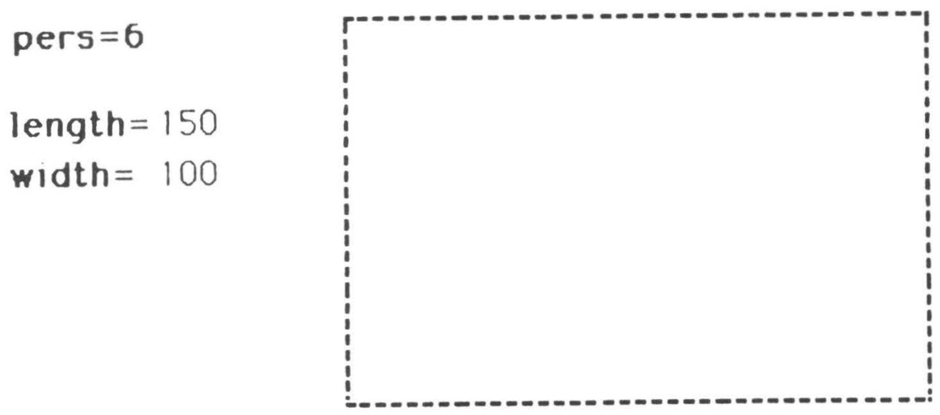


Fig. 4.

A default table supplied by the user interface

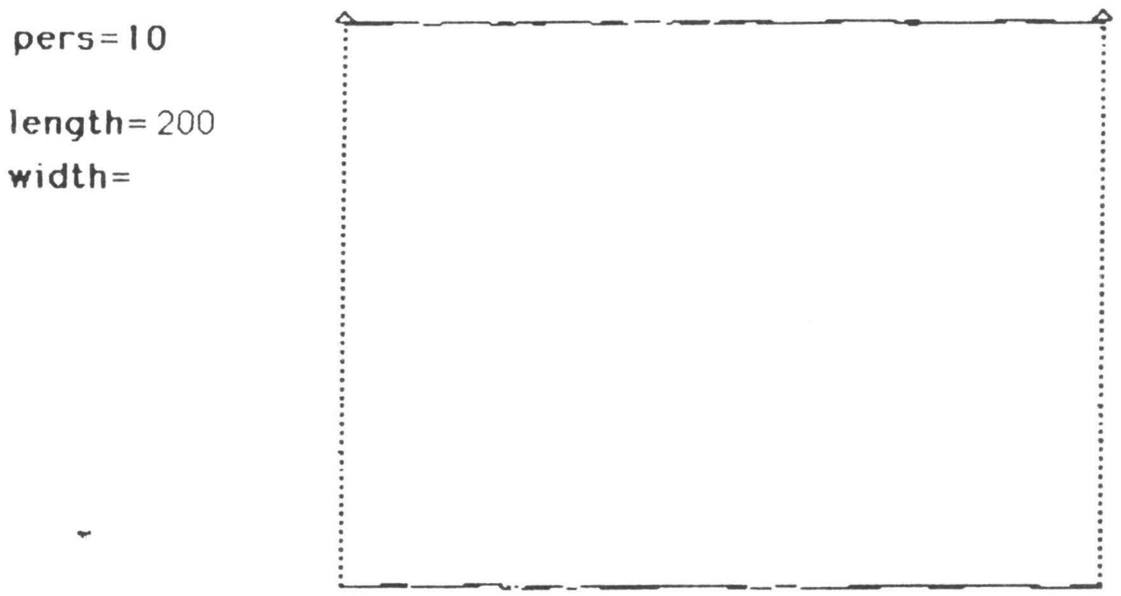


Fig. 5.a.

Visualization of a non compulsory constraint for width

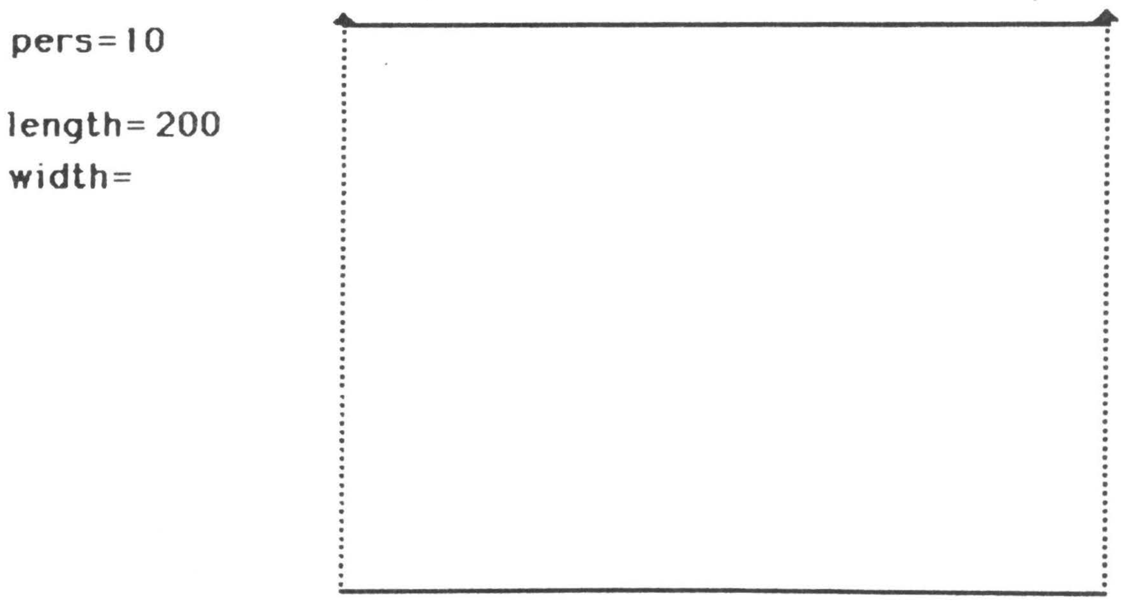


Fig. 5.b.

Visualization of a compulsory constraint for width

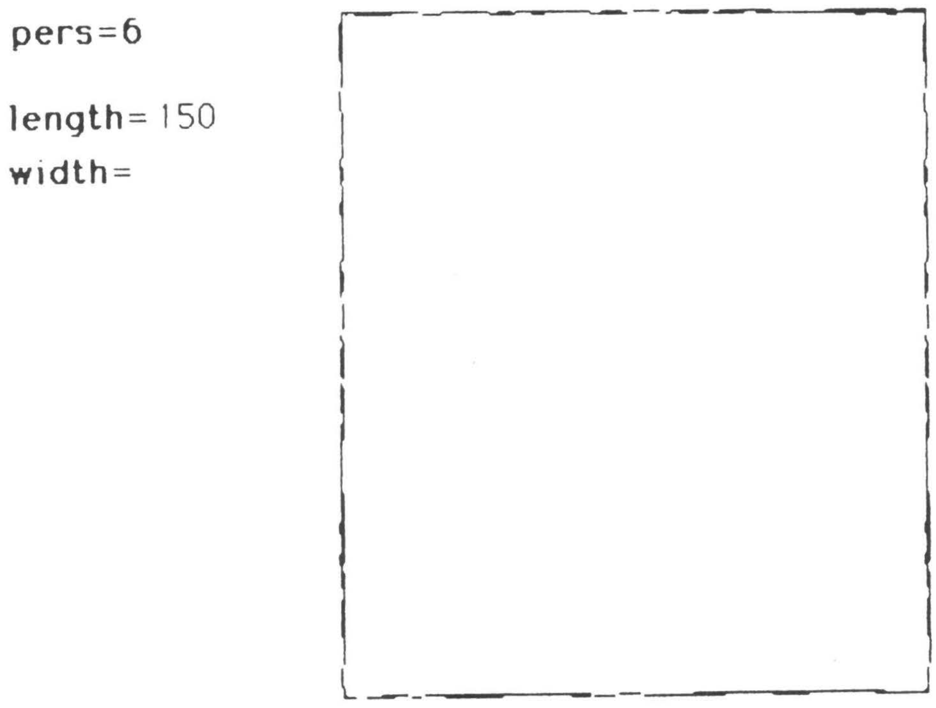

```
pers=6

length= 150
width=
```

Fig. 6.a.

Erroneus use of built-in names

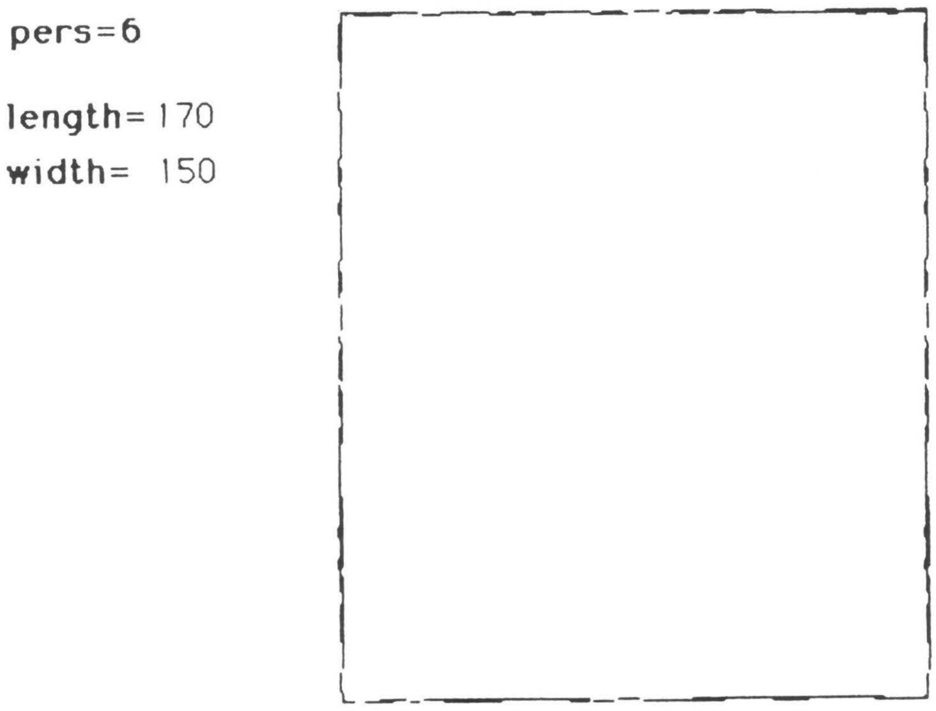

```
pers=6

length= 170
width=  150
```

Fig. 6.b.

Error recovery by re-naming

Full papers, not presented

Varol Akman, Wm. Randolph Franklin,

and Bart Veth

Design systems with common sense

EXTENDED ABSTRACT

full paper to be distributed

DESIGN SYSTEMS WITH COMMON SENSE (EXTENDED ABSTRACT)

Varol Akman, Wm. Randolph Franklin, and Bart Veth

Dept. of Computer Engineering and Information Sciences, Bilkent University, P.O. Box 8, 06572 Maltepe, Ankara, Turkey.

Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, Troy, N.Y. 12180, USA.

Dept. of Interactive Systems, Center for Mathematics and Computer Science (CWI), Kruislaan 413, 1098 SJ Amsterdam, T he Netherlands.

Precisely 30 years ago, John McCarthy wrote a landmark article titled "Programs with common sense" (1) . He proposed that a program is said to possess common sense if it automatically deduces for itself a sufficiently wide class of immediate consequences of anything it is told and what it already knows. Since then the area of Commonsense Reasoning has considerably flourished to assume the position of (arguably) the most exciting area of AI, cf. (2-7) for some recent representative works.

On the other hand, recent research in CAD has also shown a visible interest in the intellectualization of design. This is due to the general belief that the ultimate aim of CAD is the automation of the several knowledge-intensive activities performed today by highly-specialized, expensive, and error-prone experts. A common view goes like this: "Right now there are several designers who know a little about all facets of engine design, but there is no individual who could effectively desing an entire engine. Eventually, however, it may be possible for one person, using a collection of expert systems, to do a considerable amount of the engine design process " (from LOGIC, a publication of Control Data Corp., Spring 1987).

In this paper, we aim to accomplish two things: (i) identify the relevance and importance of commonsense reasoning in design, especially mechanical part (machine) design, and (ii) formulate the architecture and the capabilities of an "intelligent" CAD system which embeds common sense (in some sense). The importance in CAD of commonsense reasoning about the physical world has been first recognized (to our best knowledge) by Tomiyama and Yoshikawa (8). Our work is in the spirit of their suggestions and has been rather parenthetically published in some of our past articles (9-11) . This paper, on the other hand, is our first attempt to establish a basis for answering the following question:

What knowledge of everyday physics (viz. mechanics) does a design system need in order to predict the consequences of its (and its user's) actions while carrying out a design?

We should probably remark that we are aware of the difficulty of the problem we are dealing with. To quote Marvin Minsky "Common sense is not a simple thing. Instead, it is an immense society of hard earned practical ideas --------of multitudes of life-learned rules and exceptions, dispositions and tendencies, balances and checks " (from THE SOCIETY OF MIND). Therefore we shall follow McCarthy's advice ("In order for a program to be capable of learning something, it must first be capable of being told" (1, our emphasis)) and first try to sketch the basic features of a knowledge representation language for writing down commonsense knowledge. Since we shall, rather religiously, choose classical logic (along with some non-standard extensions) as our starting point, we shall concentrate on what to be codified in our system rather than how to reason with the codified knowledge. (This is clearly in accord with the logic programming paradigm.) A preliminary version of this language has already appeared in (9) (and later (10)); a precise version is currently under development at CWI, Amsterdam.

Clearly the proposition that one has a "theory" to deal with design is rather pretentious. We are aware of the fact that design is a "mysterious" activity which is currently done in its entirety only by human designers. Yet, to quote John Lansdown:

"In broad terms, most people would accept that designing is a cyclical process in which concepts are devised and then tested against some criteria of performance, cost or appearance. The tests: logical, physical, or just intuitive, lead to the concepts either being incorporated into the design or being rejected. In any event, the testing process gives rise to the formulation of new concepts and, importantly, then to new criteria for testing. The whole of designing thus is governed by what Ernst Gombrich calls "schema and correction" -------- almost a trial and error process where experimentation precedes correction which in turn leads to further experimentation".

We appreciate the difficulty of identifying and incorporating all the planning, heuristic, and intentive knowledge that good designers have. Nevertheless, we believe that even in the vague domain of design where any kind of formalization would probably look superficial, a formal outlook is the only way to do scientific research. We see logic as the essential framework of this formal outlook, since it is precise and unambiguous with a well-understood semantics that connects the formulas and the real world that they talk about.

There are a number of key notions that we believe an intelligent CAD system should have in order to show an appreciation of naive physics. Central among them is abstraction ------mapping of a situation to a new situation in which some attributes are ignored. (When we map a small sliding object to a point mass this is what we are doing. Similarly, we map a surface to a line segment, etc.) In the same vein, we regard the construction of physical representations that contain fictitions, imagined entities (such as forces and momenta) and the use of physical laws and principles (such as the law of conversation of energy, or the superposition principle) as the primary constituents of a theory of physics problem-solving. This is somewhat different from several workers' approach in the field who expect to find these constituents in more physchologically-oriented research, e.g. mental models (7) . In a nutshell, our naive physics should still be based on physics.

Our presentation will focus, in a decidedly theoretical way, on the formalization of design. We'll especially explain (i) the usefulness of logic in design, and (ii) treat in length the importance of naive physics and qualitative reasoning as fundamental theories about design objects and the physical processes.

Keywords: Commonsense reasoning, naive physics, qualitative physics, knowledge representation, logic, intelligent CAD, design.

Acknowledgment: We thank Ozay Oral, Mehmet Baray, Tetsuo Tomiyama, and Paul ten Hagen for their moral and intellectual support.

Note: This extended abstract has been prepared for the Third Eurographics Workshop on Intelligent CAD Systems (Practical Experience and Evaluation), Hotel Opduin, Texel, The Netherlands, 3-7 April 1989.

REFERENCES

1. John McCarthy, "Programs with common sense" in Semantic Information Processing, Marvin Minsky (ed.), MIT Press, Cambridge, Mass., pp. 403-418. (Originally appeared in Mechanization of Thought Processes, Vol. 1, Proc. Symp. National Physics Lab, London, 24-27 Nov. 1958, pp. 77-84)

2. Jerry R. Hobbs et al., Commonsense Summer: Final Report, Report No. CSLI-85-35, Center for the Study of Language and Information, Stanford University, Stanford, Calif., 1985.

3. Yoav Shoham, Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence, PhD diss. (also Report No. YALEU/CSD/RR 507), Computer Science Dept., Yale University, New Haven, Ct., Dec. 1986.

4. James G. Schmolze, Physics for Robots, Report No. 6222, BBN Labs, Inc., Cambridge, Mass., Sept. 1987.

5. Daniel G. Bobrow (ed.), Qualitative Reasoning about Physical Systems, MIT Press, Cambridge, Mass., 1985.

6. Jerry R. Hobbs and Robert C. Moore (eds.), Formal Theories of the Commonsense World, Ablex, Norwood, N.J., 1985.

7. Dedre Gentner and A.L. Stevens (eds.), Mental Models, Erlbaum, Hillsdale, N.J., 1983.

8. Tetsuo Tomiyama and Hiroyuki Yoshikawa, "Extended general design theory" in Design Theory for CAD, H. Yoshikawa and E.A. Warman (eds.), North-Holland, Amsterdam, 1987, pp. 95-130.

9. Bart Veth, "An integrated data description language for coding design knowledge" in Intelligent CAD Systems I: Theoretical and Methodological Aspects, P. ten Hagen and T. Tomiyama (eds.) Springer-Verlag, Berlin, 1987, pp. 295-313.

10. Varol Akman et al., "Knowledge engineering in design", Knowledge-Based Systems, Vol.1, No.2, 1988, pp. 67-77.

11. Varol Akman and Paul ten Hagen, The Power of Physical Representations, Report No. CS-R8819, Center for Mathematics and Computer Science, Amsterdam, May 1988.

Meurig Beynon

Definitive programming as a framework for design

Definitive programming as a framework for design

Meurig Beynon
Dept of Computer Science, University of Warwick, Coventry CV4 7AL, UK
Telephone: +44 (203) 523089 Telex: 31406 COVLIB G Fax: +44 (203) 461606
E-mail: wmb@uk.ac.warwick

Introduction

Research on design support systems has traditionally been concerned with two themes. One is the development of tools with powerful problem solving abilities in restricted problem domains. The other is with developing better interfaces between the designer and the system. These two concerns are complementary. Making tools more and more sophisticated and capable of autonomous action is intended to enhance the role played by the computer in the design process. Building more intelligent user interfaces aims to assist the designer in articulating a design. The challenge is to build systems that allow the designer to exploit the computational power of modern computers without becoming locked out of the design loop in the process.

Trends in recent research reflect the need to address this fundamental problem. The IIICAD (Intelligent Integrated Interactive CAD) system, as introduced in [19], aims to provide "an integrated infrastructure for problem solving tools together with a smart user interface". What is involved in building such an integrated infrastructure? Establishing common data representations that allow a simple integration of CAD systems and AI tools is not enough. The system components have to operate in a manner that can be intelligently controlled by the designer. The IIICAD system proposes to solve this problem through integrating the design knowledge associated with the tools, and introducing an intelligent supervisor to control the system on the basis of this knowledge.

This paper approaches the problem of realising the ideals of IIICAD in a different way. The difficulties of integrating powerful computational tools within a system that accommodates the human designer are interpreted as reflecting the inadequacy of current methods for representing computation in an interactive environment. It is argued that by choosing a better programming paradigm for describing interactive computation, it will become possible to make complex computational processes intelligible to the designer to a far greater degree. This will enable the designer to interact more closely and directly with the tools.

In understanding how to apply new principles to interaction within a design support system, it is important to clarify their potential role and scope. There is clearly a place for tools whose computational activity is opaque to the designer. It may be neither appropriate nor practical to equip the designer to intervene in the technical computations performed by tools, e.g. in numerical computation, or routine catalogue searching. But there will also be computational tasks within the system in which the designer has to be centrally involved, and where the raw application of computing power is no substitute for human insight and intuition. Much of the activity associated with the construction and validation of a design object is of this nature. It is in supporting this interaction that an appropriate representation for computation is crucial.

The problem of representing the computation carried out by the system to the designer in a useful manner is fundamentally linked to the choice of the underlying programming paradigm. This paper develops proposals made in previous papers [4,5,6], further exploring a computational model that exploits a definitive (definition-based) programming paradigm. Definitive principles are particularly well-suited to expressing computation in which there is interaction between user and computer. They are intended to give the designer means of directly effecting state transitions in a computation, of setting up autonomous computational processes to transform, simulate or realise a design object, and of suspending and intervening in such computational processes where appropriate.

The four sections of the paper respectively concern the motivation and basic concepts of definitive programming, the proposed application of definitive principles to design support, some simple illustrative examples of their use, and a brief comparison with alternative approaches.

§1. Definitive programming: motivation and basic concepts

What will the design environments of the future look like? It is to be hoped that the designer will be able to build a representation of the design object, to simulate its behaviour, and to customise the design environment itself. The latter involves the interactive modification of the design tools to assist the processes of flexible representation and effective simulation. Present techniques aimed at addressing these issues are diverse and hard to integrate: the representation of design object will perhaps be in form of information stored in a sophisticated data base; its behaviour will be described by means of qualitative reasoning expressed in terms of inference mechanisms; adaptation of the design tools will typically involve editing the specification of a piece of software.

From the perspective of this paper, the evolution of the design object as the design develops, the changing state of the design object as it is transformed in simulation, the modification of the design interface to suit the designer's present needs are all regarded as computations that unfold during the design process. Each is a computation which the designer has to comprehend, and be able to direct accordingly. They are computations in which the concept of a current state is strongly represented, and in which the course of the entire computation cannot be preconceived.

The development of programming paradigms has been heavily influenced by the need to hide computation from the user. Surely the designer does not need know the elaborate computations required in displaying a geometric object, or to understand the procedure by which a particular data item is retrieved from a catalogue. It is sometimes essential for clarity to use declarative abstractions that describe what is being computed without reference to the computational recipe to be used. But it is hardly to be expected that the programming principles that hide computation are also appropriate for interactive computation in which cooperation between the user and the computer is required. This is the principal motivation for a novel programming paradigm based upon the use of definitions: definitive programming. This section provides an overview of definitive programming, and introduces the concepts to be discussed and illustrated in later sections.

Definitive programming is a state-based programming paradigm in which the central abstraction is the *definitive system*. A definitive system is a family of variables such that the value of each variable is either specified explicitly, or is defined by a formula in terms of constants and other variables. The values of the variables in a definitive system can be seen as determining the current state, and a definitive system as representing a possible way in which the current state can be changed. The potential state transitions represented by a definitive system are associated with the redefinition of one or more of the explicitly defined variables.

In interpreting a definitive program, it may be useful to think in terms of a state transition model. Though the current computational state is strictly speaking represented by a set of values, it is often appropriate to represent this via a system of definitions. As explained above, such a system of definitions expresses latent state transitions associated with the redefinition of explicitly defined variables within the system. Following the conventional method of presentation of a state-based machine model, the computational state (as represented by the set of values associated with variables in a definitive system) can be conceived as a node of a graph, and the possible transitions from this state (as represented by possible changes of explicitly defined variables within the definitive system) as edges directed outward from this node. The combinatorial graph specified in this way will be very hard to conceive in all but the very simplest cases, but provides a faithful representation of the rich computational structure associated with a definitive program. Since several independent transitions associated with a single definitive system can be non-interfering, there is also the possibility of composite state transitions that in effect correspond to concurrrent execution of several redefinitions. Such a model of concurrent computation underlies the abstract definitive machine described below.

In the simplest form of definitive programming, the transition from state to state is entirely under the control of the user. The user generally describes an initial computational state by formulating appropriate definitions, and subsequently changes the state by redefining variables. Such a style of programming subsumes the underlying principle of the spreadsheet. A definitive system may be used to define the profit as a function of costs and sales, for instance, so that the expected profit changes according to the predictions for costs and sales. Definitive systems can be formulated for other applications, when the values are not merely scalar, and the operators not merely arithmetic. The values and the operators that can be used to combine values in a formula constitute an *underlying algebra*. By choosing the underlying algebra appropriately, a wide range of applications can be addressed [5]. It is possible to use a definitive system to describe the screen layout, for instance, so that updating the screen involves redefining appropriate parameters.

The use of a single definitive system over a sufficiently rich underlying algebra itself provides an expressive programming paradigm. In a functional language such as **miranda** [21], the user can introduce powerful operators, together with variables of sophisticated higher-order types that can be defined by very general formulae. A **miranda** script is a definitive system that determines the transition from one **miranda** environment to another associated with changing the values of explicitly defined variables in the script. Referential transparency is a central principle of **miranda**, and editing the script is outside the scope of the functional programming paradigm. Within the **miranda** system, the effect of redefining explicitly defined variables in the definitive system can only be imperfectly realised in other ways. One approach is to omit such variables from the script, and to obtain the values of other dependent variables through function evaluations with different parameters. Another approach that potentially represents state more faithfully involves introducing variables whose values are histories.

A definitive system is a significant generalisation of a functional programming script. Editing the script falls within the definitive programming paradigm, and the redefinition of a variable is interpreted as a transition from one computational state to another. Redefinition permits dynamic modification of the environment for evaluation. In the simplest cases, this entails redefining an explicitly defined variable, but it can also serve to set up a definitive system to represent new transitions, or to add new definitions. This effectively means that the state transition model above is enriched by introducing other transitions corresponding to the introduction of new variables, and to the redefinition of implicitly defined variables. Note also that variable redefinitions can refer to the values of variables in the current state. This generalisation may appear to be a matter of convenience where user-computer interaction via a definitive system is concerned, but has wider implications.

From the functional programming perspective, perhaps the most unusual aspect of developing definitive systems in the manner described above is that the conventional distinction between constructing and executing a program is blurred. In introducing new variables and definitions, the user is incrementally specifying an environment that can be explored through variable evaluation after the manner of functional programming, but may exercise an additional freedom to interpret a redefinition as a state change within a single environment, rather than as a new choice of environment. From the definitive programming perspective, as will be illustrated in the examples that follow, the distinction between transitions from one computational state to another has to be understood with reference to privileges of agents. Pure functional programming is a restricted form of definitive programming in which the user establishes a particular computational state by editing a script, and authorises no agent to change this state subsequently.

Definitive programming on the above pattern is limited to identifying sequences of state transitions that have a useful interpretation e.g. via the observed behaviour of an object. General-purpose programming demands much more than user-driven sequences of state transitions can provide. In modelling the behaviour of a complex object, the state transitions that are permissible will depend upon context. In a complex application, several different kinds of state changes occur concurrently. The computational model underlying general-purpose definitive programming is supplied by the "abstract definitive machine (ADM)". In this model, the user is generally one amongst many agents that can perform state transitions by redefining parameters within a definitive system. Reference to values of variables in the current state is essential when specifying agents in the ADM.

The ADM has been described in detail elsewhere [8], and only its most relevant features are considered here. During the execution of the machine, the computational state is represented by sets of definitions and actions D and A respectively. The sets D and A change dynamically as the program executes, through redefinition in D as prescribed by actions in A, and through the introduction or deletion of definitions and actions. A program for the abstract definitive machine comprises a set of entities, each of which specifies a set of definitions and a set of actions that are to be instantiated or deleted as a whole. To initiate execution, an appropriate set of entities is instantiated. Execution then proceeds in a sequence of machine cycles in which several actions may be performed concurrently.

Each action in A takes the form of a guarded sequence of redefinitions and/or instantiations or deletions of entities. The guards in actions are boolean expressions in the variables that appear in D. On each machine cycle, these guards are evaluated in the context supplied by D. Provided that there is no interference, the corresponding systems of redefinitions and entity reconfigurations are performed in parallel. The most significant forms of interference occur when a redefinition results in circularity, or leads to the evaluation of a variable that is currently directly or indirectly being redefined. The definitions in D play a similar role to the definitions underlying a multi-user spreadsheet - they record the current state of the system as established and modified through actions. The actions in A correspond to the user dialogue actions. The ADM supplies a framework within which many different agents can act concurrently by redefining explicitly defined variables in a definitive system. Though the fundamental concurrent computational model is synchronous, it is possible to simulate asynchronous activity within the ADM by introducing appropriate control entities (c.f. [7,8]).

Definitive programming has many potential merits for design support:

1) it provides a state-based programming paradigm;
2) it allows agent privileges and actions to be explicitly modelled;
3) it supports rich techniques for data representation and presentation.

The remaining sections of the paper examine the prospects for applying definitive programming principles to the motivating design support issues introduced at the outset. For this purpose, the computational tools within the design support system that are not directly accessible to the designer will determine the data types and operators in the underlying algebra. This means that, below a certain level of abstraction, computation in the design support system is conceived in conventional functional programming terms. How far the computation within the system is recast in definitive programming terms will reflect the extent to which the designer can penetrate the system usefully. For the present, even the degree of designer involvement conceived below may prove impractical. In principle, the intelligent intervention of the designer could be valuable even in some low-level computations, e.g. in resolving problems involving geometric singularities for purposes of graphical display.

§2: Definitive principles and the design process

How are definitive programming principles intended to be used for design support? This issue will be discussed abstractly in this section, with cryptic reference to the problems of designing simple mechanical objects such as doors that will be considered in greater detail in §3.

Three aspects of the design process in which interaction with the designer plays an essential role can be identified:

1) representation of the design object itself, and validation of the design

The designer typically has to construct a model of an object to be manufactured, to simulate its behaviour in the intended application under reasonable assumptions, and to confirm that this behaviour is consistent with that required. This part of the design process involves a cognitive component that only the designer can supply.

2) representation of the designer's concepts of the design object, and their development

To aid the designer, it will be necessary to impose some conceptual structure upon the design object, for instance, in the form of parametrisations that allow features to be conveniently modified. The appropriate form for these parametrisations cannot be preconceived, since they reflect the peculiar characteristics of the object, and the designer.

3) representation of the design object by and to the designer through the computer interface

The designer will need to be able to adapt the protocol by which data about the design object is presented dynamically, and to determine the manner in which feedback about the current form and status of the design object is given. These will depend upon the specific design problem, and must be guided by the imagination of the designer. A standard repertoire of input and feedback techniques can support only very limited applications.

Amongst the three, 1) is of course the primary concern, from which 2) and 3) are derived. A good designer will choose parametrisations of design objects to suit the intended function of a design object, and modify these to reflect the results of simulation. To work effectively, a designer will need to construct environments within which different parametrisations of a design object can be conveniently formulated and flexibly invoked, and within which the necessary processes of validation can be reasonably faithfully and efficiently performed.

At any stage in the design process, the current status of the system with respect to 1), 2) and 3) will have to be represented. The principal end-product of the design process will be the representation 1), but there may be a significant role for variant designs associated with 2), and for an interface 3) that provides a realisation for the abstract model described in 1).

Following [5], all three aspects of the design process considered above will be addressed using definitive principles. ADM models will be used to express the current status of the system with respect to 1), 2) and 3). The same paradigm for representing states and transitions applies in all three contexts, viz the configuration of a definitive system to suit a proposed transition followed by appropriate changes of explicitly defined variables. As has been explained above, the designer will be privileged to set up these models to suit the purpose of the design problem, and in this sense will in principle be free to intervene in all three. The agents that typically initiate changes in state within these models have a different significance however, as will now be explained.

Definitions and actions that represent the design object

The model of the design object, as produced by the designer, will be an ADM program that describes characteristic relationships between parts of the object, and expresses the ways in which these can change as the state of the object changes. For a very simple object, the ADM model might consist of a single definitive system, together with actions that express the conditions under which explicitly defined variables can be changed (e.g. the conventional door). For more complex objects, a program that makes fuller use of the capabilities of the ADM is required (e.g. the door that closes automatically under the action of a spring). Transition within the design object model is to be understood with reference to the intended use of the object, so that the ADM actions will correspond to changes in state that might be initiated by design object daemons representing agents acting in the application (e.g. a horse closing the upper half of a stable door).

Constructing a design object model involves a cognitive process, whereby the designer formulates definitions and actions to represent the observed or intended relationships between components of a physical object (e.g. describing the locus of the lock of the door as its aperture varies). It cannot realistically be a comprehensive model of the behaviour of the design object (e.g. several people using a door). The validity of a design object model can only be assessed, to some extent, by analysis and simulation. The ADM model of a design object provides the basis on which appropriate simulations can be built. It is intended to provide a computational model within which the designer can intervene to resolve conflict if necessary (e.g. to specify a priority for door users).

Definitions and actions that assist the designer in developing the design object model

Determining the form of the design object model is choosing a representation for the design, and impinges upon design support only in so far as some representations are better than others for purposes of analysis and simulation. Central to the designer's view are the methods within the design support system that make it possible to redesign an object i.e. to change the object in ways that are outside the scope of normal use. These will be represented by ADM models that most closely represent the designer's perspective on the design object, in which the definitions are expressly under the control of the designer. The simplest design object manipulation models are parametrised objects: definitive systems that describe several different designs depending upon the choice of certain explicit parameters. Other definitive systems will establish relationships between specified objects at different levels of abstraction (e.g. ensuring that all the doors to a building follow the same pattern), and can be used to maintain appropriate constraints (e.g. to relocate the hinge of the door whilst maintaining the correct geometry for the door cavity in the wall). Actions in this context might serve to transform the designer automatically from one framework for manipulation to another (e.g. switching from a context that relocates a hinge without changing the dimensions of the door, to one in which the width of the door is changed), or to impose constraints upon the designer (e.g. automatically revoking redefinitions that violate a specified constraint).

Definitions and actions that support the realisation of design object

Definitive principles are particularly well-suited for establishing direct connections between one data representation and another. The need to represent the design process to the designer effectively will be met by regarding the interface to the designer as itself a definitive system whose value is specified in terms of the internal ADM models described above. The definitions and actions that comprise this design object interface model form an essential part of the user-interface to the design support system. The designer is intended to adapt this model to suit the specific design task, but it will be preprogrammed to a much greater extent than the internal ADM models, and will generally perform most transitions through autonomous redefinition. The role of the design object interface model is to ensure that the definitions that describe the current screen layout properly reflect the nature and status of the current interaction. For instance, if the behaviour of the design object is being simulated, a set of definitions defining the external representation of the design object in terms of the internal representation will be established. If the designer is currently entering new information about a design object, a set of definitions to describing the appropriate screen layout, to include for instance menu format and status, will be set up.

In such a view, the designer's role is the development of an ADM program that combines several computational strands, resembling the musical counterpoint between independent voices. Closest to the machine level are the activities that describe the screen manipulation and feedback to the designer. Within the design application, there are definitions to represent the design object, and actions to be performed by the design object daemons in simulation. There is the program that the designer develops in order to manipulate and transform the design object in the process of design itself. Whether the approach to design represented by such an ADM program effectively meets the requirements of IIICAD depends upon the scope for dynamic intervention and modification given to the designer. It may not be possible for the designer to suspend and redirect a simulation, or to modify the way in which a design object is presented, for instance. The goal of present research on definitive principles for design is to investigate how concurrent execution of these computations can be supported in such a way that the designer can effectively monitor them and intervene as appropriate.

§3: Illustrating the use of definitive principles

Suppose that an architect is designing a door to suit a particular building. The context into which such a door might be placed can be described in an existing prototype system thus:

```
line      n1 = [NW, Lframe]
real      tablelength
line      n2 = [Rframe, NE]
point     NW
point     NE = NW + 6*doorwidth
int       k = 1 + tablelength/doorwidth div 1

real      doorwidth
point     Lframe = NW+k*doorwidth
point     Rframe = NW+(k+1)*doorwidth

monitor   k>5
```

The above definitions are intended to assist the architect to relocate the door so that there is sufficient room for a table to the left of the door frame, and to keep the width of the door in constant proportion to the length of the wall to which it belongs. The definitions enable the architect to express the essential relationships even though the position of the wall is as yet unspecified. Note that these definitions are for the benefit of the designer: they will determine values that are fixed when the door is in use. Monitoring the condition k>5 means ensuring that a message is displayed as and when the value of k exceeds 5. A monitored condition might be implemented by establishing a variable whose value is a string defined by the formula:

```
    if k>5 then "Warning: k>5\n" else ""
```

and setting up a variable whose value was a window in which the values of all such monitoring variables was concatenated. This illustrates the nature of the definitions used to support the implementation.

Certain characteristics of the door, such as its dimensions and type, are to be determined by the architect, and thereafter fixed. Choosing these characteristics constitutes door design. To determine whether a particular design is appropriate, the architect will need to simulate the door in use. For this purpose, the architect requires a computational model of how the door will behave. A simple definitive specification of a conventional door, as it might appear on an architectural plan, is:

```
    real   width = doorwidth
    bool   open
    line   door = [hinge, lock]
    point  hinge = Rframe
    point  lock = hinge + if open then {0, -width} else {width,0}
```

In interpreting such a definitive system as a design, it is not enough just to consider the current values of the variables and the relationships between them. It is essential to appreciate the status of each definition, and take account of the ways in which particular agents can change the system state. In general, some definitions will reflect constraints imposed by physical laws. The definitions relating door, lock, and hinge reflect the physical characteristics of the particular type of door chosen by the designer. Other definitive systems would be characteristic of a revolving or a sliding door. Definitions may also reflect rules otherwise imposed upon the design, as when the width of the door is pre-specified. Within the framework of such constraints, definitions can be freely chosen by the designer to ensure that the the system exhibits an appropriate behaviour. The position of the door is determined by where the hinge is located; this cannot be changed in normal use. In simulating typical use of the door, only the value of the variable 'open' can be changed. Declaring the circumstances under which particular variables (such as the variable 'open') can be redefined by the users of the system is a part of the design.

Agent protocols are generally implicit in design. It is obvious to the architect that the orthodox room user cannot change the width of the door. Because definitive methods allow state changes associated with design and simulation to be represented in the same manner, they highlight the need for protocols. In the above example, "opening the door" and "moving the hinge" cannot otherwise be distinguished. (Compare a conventional simulation program in which "moving the hinge" would involve editing the program, and "opening the door" supplying further input.) A user protocol may also be needed to complement a complex design, as when a designer formulates rules for the use of the design object.

In order to express the design in terms of what agents can do, and when, a designer will wish to organise actions by agent, specifying the preconditions enabling each potential action. (This is the approach that is being developed in the notation LSD [7].) A simple agent action (such as opening the door) will correspond to changing the value of an explicitly defined variable in the context of a definitive system. A possible protocol for the door user might be:

```
    not open -> open = True
    open -> open = False.
```

A more subtle protocol permits the door to be locked by another agent:

```
    not locked and not open -> open = True
    open -> open = False.
```

There are many variants of these protocols, as when a door automatically locks on closing for instance.

A protocol does not comprise actions directly suitable for interpretation in the ADM. An action in the ADM is to be executed when its guard is true: during a simulation an agent performing actions permitted by its protocol may choose between several possible options, and perhaps perform no action at all. The door protocol above gives no information about when a user is likely to open the door, nor what should happen if one users tries to open the door at the same time as another tries to lock it. The total concept of a design object is open-ended, and cannot be divorced from its intended use. The application of definitive principles permits the representation of a design object by an ADM program that encapsulates the characteristic behaviour of the object (e.g. how the door responds to being opened and closed). Such a program can then be used as a component in simulations of the object in typical use (e.g. the door being opened and closed by several users in a concurrent system). These principles are more fully illustrated by the next example.

Suppose that an architect wishes to specify a stable door, in which the upper part of the door can be opened independently of the lower. Such a door could be described by the definitive system:

```
bool   uopen
bool   lopen
line   udoor = [hinge, ulock]
line   ldoor = [hinge, llock]
point  hinge
point  llock = hinge + if lopen then {0,-width} else {width,0}
point  ulock = hinge + if uopen then {0,-width} else {width,0}
```

In this model, a typical user can independently change the values of the boolean variables 'lopen' and 'uopen'. In practice, the stable door might be designed so that the upper door overlapped the lower. The variables 'uopen' and 'lopen' would then be constrained to satisfy

$$\text{lopen} \Rightarrow \text{uopen}.$$

Note that this constraint itself does not fully specify how the stable door should respond to movement of the upper and lower doors. It is consistent with the supposition that the stable door behaves like a conventional door, for instance.

The protocol for using the overlapping stable door is more complex. It must express the fact that if both upper and lower doors are closed, and the lower door is opened, then the upper door also opens. At this stage, it is then possible to close the lower door whilst leaving the upper open. This can be expressed by a protocol of the form:

```
not uopen  ->  uopen = lopen; lopen = true
uopen      ->  lopen = false
```

This protocol is to be interpreted as asserting that the definition

uopen = lopen

pertains whilst the lower door is being opened from the context in which both upper and lower doors are presently closed. Such a protocol provides the basis of an ADM program simulating the operation of the door.

As a more complex illustration of how a design object is described by an ADM program, consider the implications of attaching a locking device to the stable door so that the upper and lower doors become "ganged" (i.e. locked together). In this case, the user protocol includes the action:

```
uopen == lopen and not ganged ->  ganged = true
```

that has more significant implications than previous redefinitions. Whilst the doors are unganged, 'lopen' and 'uopen' can be independently controlled as described above. Once the doors are ganged, the definitive context within which actions are performed is changed radically. Where before there might have been a variable 'open' with a value partially specified by the definition

```
open = if lopen==uopen then uopen else @,
```

the variables 'lopen' and 'uopen' are now more appropriately defined in terms of 'open', via:

```
lopen  = open
uopen = open,
```

and the variable 'open' is subject to user control. In yet another model, the doors might become ganged after an appropriate locking device was set, on the first occasion on which the condition 'uopen == lopen' was satisfied.

The development of ADM models of design objects for purposes of simulation is as yet at an early stage. Constructing ADM models to simulate objects such as the stable door in use is a subject of current research beyond the scope of this paper. For a fuller discussion of some of the issues, the interested reader can consult [8,9], in which an ADM model of a small system comprising two blocks connected by a string moving in discrete steps under the control of independent agents is described. It remains to be seen to what extent the construction of such models can be simplified through the application of definitive principles, but there are some indications that the knowledge of data dependency implicit in the ADM model for action can assist in detecting interference between concurrent actions, and allow the designer to intervene to resolve them. For instance, in the prototype blocks simulation referred to above, the fact that moving the right block to the right when the string is taut invokes a definitive system in which the position of the left block is defined in terms of that of the right, and vice versa, enables the ADM to recognise interference between the actions. In the context of this paper, it is particularly significant that the simulation is gracefully suspended when such conflict arises, so that the user can be consulted to decide upon the appropriate action, and the simulation resumed.

§4: Prospects for applying definitive principles for design support

The aim of this paper is to put into perspective several aspects of the research into definitive principles to CAD systems represented in previous papers [4,5,6,7]. This research is focussed upon supporting interaction between the designer and the system in crucial aspects of the design process:

1) representing intrinsic relationships associated with design objects, and (potentially) simulating their behaviour. Relationships referred to here are characteristic of the function of the object;

2) assisting the designer in the manipulation of representations of design objects, support context switching between representations suited for different purposes. Here the relationships being modelled are those that are implicit in the designer's view - how the designer has conceived the structure of the design object;

3) allowing the designer to adapt the feedback from the system to suit the particular characteristics of the design problem. Relationships being modelled here relate the different types of data within the system, perhaps linking data values in entirely different semantic categories.

In each case, a role for definitive systems has been identified, together with a need for a framework such as the ADM supports to allow these definitive systems to be dynamically reconfigured, whether according to context, or under the control of the designer. Even autonomous reconfiguration may be programmable within the system, but should ideally be intelligible to the designer, so that it can be suspended and resumed after intervention by the designer. More research is needed into how the ADM model of computation must be presented for this purpose.

Logic programming methods figure prominently in alternative approaches to 1). Comparison between simulation in the ADM model (as represented in [8]), and the commonsense reasoning approach proposed e.g. in [13,14], raises many interesting issues. Some recent research has questioned the role of inference in cognition [15,16], and it will be of interest to consider whether definitive principles can offer a better cognitive model.The three classical problems for reasoning about action: the frame, ramification and qualification problems also deserve serious examination in this light. There is superficially a strong resemblance between the use of definitions and constraint-based methods [10], but there are significant differences. Whereas constraints are best viewed as assertions about the current state, definitions should be seen as giving information about latent change, and address the concept of relationship in conjunction with potential change.

There are interesting connections between definitive programming and both object-oriented and functional programming to be further explored. The present approach to supplying user-defined functions in the definitive notation DoNaLD promises to import the significant advantages of recursively defined functions, and of object-like abstractions [9]. The absence of data hiding principles in definitive programming distinguishes it from the object-oriented paradigm, and proves to be an advantage where the modelling problems discussed in [20] are concerned. Recent research has involved the implementation of a definitive interface to the **miranda** system that makes it possible to redefine variables without invoking the editor. In such a framework, it is easy to construct a transparent state-based model of a conventional door, but apparently difficult to cope with a stable door, notwithstanding the potential for the use of higher order functions. These experiments strongly indicate the importance of some form of history sensitivity to complement pure functional programming methods [1,2].

The merits of definitive principles in respect of 2) have been well-established in prototypes. It is important to remember that the perspective of the designer changes in the process of design, and that the virtues of referential transparency can only be realised relative to the designer. It is in the process of helping the designer to maintain a computer representation of an object that is consistent with the designer's current view that definitive methods prove most obviously effective. As a particularly simple illustration of this concept, it is trivial to construct a mechanism for consistently pointing at an abstract point within the DoNaLD system. The importance of such reference capabilities has been discussed in detail elsewhere [9]. Definitive principles also promise to support the representation of objects at different levels of abstraction, and where information is partial. The hierarchical nature of agent privileges, whereby the designer works within a framework of definitions and actions reflecting physical constraints, and the designer frames the context within which the behaviour of the design object is simulated, implicitly gives support for views.

The ADM computational model has some features in common with a rule-based approach (c.f. [3,19,17]). The ADM model of computation is intended to help the designer to interpret the autonomous computation performed by entities, and enhance the prospects for intervention and cooperation. In this way, the functions of the intelligent supervisor in the IIICAD system [19] can perhaps be served by a program that relies upon the designer rather than a set of preprogrammed rules for its intelligence.

Communication within a CAD system makes essential use of symbols that reflect the mental structures in the designer's imagination. Definitive principles are very well-suited to establishing the direct links between one form of data representation and another. It would be easy to use the underlying **openshape** data structures in DoNaLD to create visual representations, for instance. Object-oriented methods have already proved quite effective in establishing the relationships between internal and external representations required for good interfaces [11], but there are problems in making these relationships appropriately sensitive to data values [12]. A major problem for a preprogrammed interface is that the most appropriate representations are hard to anticipate without problem specific knowledge. Similar difficulties are encountered in making user-adaptive interfaces. It is in these respects that the blurring of the distinction between program development and execution in the definitive programming paradgm referred to in §1 may prove most advantageous.

Concluding remarks

Current work on applying definitive principles to design support is focussed on two main issues: developing a geometric modelling package (CADNO) following the preliminary design proposed in [5], and developing ADM models of design objects for simulation purposes [8].

The informal description of definitive programming in terms of a state-transition model in §1 invites direct reinterpretation in design terms. States in which the set of values includes undefined values are of limited interest in programming, but can have valid meanings in design. The confusion between the concepts of editing and executing a program is entirely appropriate in the context of design, where the distinction between state changes of the design object associated with design and simulation is essentially a matter of perspective (contrast 'relocating a shelf' with 'opening a window'). The flexibility afforded by the use of definitive principles is directly related to the way that design and simulation are integrated in this fashion. In principle, it allows the designer to assume - or to delegate to appropriate agents - arbitrary privileges to reconfigure a design or the design environment without moving outside the system. Such an approach allows the designer to adopt a "Wrong-Every-Time" rather than a "Right-First-Time" methodology, in the spirit of the retrospective planning of [18]. Experience with prototypes has so far confirmed the importance of these virtues.

Acknowledgements

I am greatly indebted to Steve Russ for helpful criticisms of earlier drafts of this paper.

References

1. Abelson H, Sussman G J *Structure and Interpretation of Computer Programs* MIT Press 1985
2. J Backus *Can programming be liberated from the von Neumann style?* Turing Award Lecture 1977, CACM **21**, 8 (August) 1978, 613-641
3. P Bernus, P J W ten Hagen, P J Veerkamp, V Akman, *IDDL: The Language of a Family of IIICAD systems*, in Intelligent CAD Systems 2: Implementation Issues, Springer Verlag *to appear*
4. W M Beynon, *Definitive Principles for Interactive Graphics*, NATO ASI Series F:40, 1987, 1083-1097
5. W M Beynon, A J Cartwright, *A Definitive Programming Approach to the Implementation of Intelligent CAD systems*, in Intelligent CAD Systems 2: Implementation Issues, Springer Verlag *to appear*
6. W M Beynon, A G Cohn, *Representing design knowledge in a definitive programming framework*, (paper prepared for IFIP WG 5.2 Workshop Cambridge, UK September 1988)
7. W M Beynon, M T Norris, M D Slade, *Definitions for modelling and simulating concurrent systems*, Proc IASTED Conf ASM'88, Acta Press 1988, 94-98
8. W M Beynon, *Definitive programming for parallelism*, Univ of Warwick RR#132, 1988
9. W M Beynon, *Evaluating definitive principles for interactive graphics*, Proc CGI'89 *to appear*
10. A Borning, *The programming language aspects of ThingLab, a constraint-oriented simulation laboratory*, ACM Transactions on Programming Languages **3**(4), 1981, 353-387
11. A Borning, R Duisberg, *Constraint-Based Tools for Building User Interfaces,* ACM Transactions on Graphics, Vol 5 No 4 October 1986, 345-374
12. J Foley, *Models and Tools for the Designers of User-Computer Interfaces*, NATO ASI Series F: Computer and Systems Sciences, Vol 40, 1121-1152
13. M L Ginsberg, D E Smith, *Reasoning about Action I: A Possible Worlds Approach*, Artificial Intelligence 35 (1988) 165-195
14. M L Ginsberg, D E Smith, *Reasoning about Action II: The Qualification Problem*, Artificial Intelligence 35 (1988) 311-342
15. P N Johnson-Laird, *Mental Models* CUP 1983
16. D McDermott *A critique of pure reason* Comput Intell **3**, 151-160, 1987
17. R Popplestone, T Smithers et al, *Engineering Design Support Systems*, IKBS/MS 7, 1986
18. T Takala, *Design Transactions and Retrospective Planning Tools for Conceptual Design*, in Intelligent CAD Systems 2: Implementation Issues, Springer Verlag (to appear 1988)
19. T Tomiyama, P J W ten Hagen, *Organization of design knowledge in an intelligent CAD environment*, CWI Report CS-R8720, 1987
20. T Tomiyama, *Object-oriented programming for intelligent CAD systems*, in Intelligent CAD Systems 2: Implementation Issues, Springer Verlag (to appear 1988)
21. The **miranda** manual, *Research Software Ltd*, 1987

Position papers

Jean-Philippe Bernardoux

OUR ACTIVITIES

Our society

SYSECA is a French expanding System and Software House created in 1966. Its unique Shareholder since 1975 is THOMSON-CSF.

Its turnover is 500 MFF for 1988, and it gathers 1300 people including 65 in the Artificial Intelligence field. SYSECA is among the French leading Software House in Artificial Intelligence.

Our activities

Our professional services spread from Engineering to Consultancy and Software Packages. Our fields of activity are Real Time Systems, Industrial Turnkey Systems, Computer Aided Software Engineering, Communication Systems, Data Bases Management Systems, Business Applications and, of course, Artificial Intelligence.

For more information about our group, we enclose hereby some documentation.

OUR POSITION

Within our Artificial Intelligence activities, we work in several projects dealing with Intelligent Computer Aided Systems, and more especially in Design problems.

One of our most advanced projects is a Layout Design one in which the problem is to place equipments on ships.

Our Layout Design Project

Ships are very complex structures, and there are many constraints between the equipments themselves, and between the equipments and the different parts of the structure of the ship.

For examples, some equipments must be away from some others, others must be close to certain parts of the structure, or rather high over the sea-level,...

Our design approach

We made a bibliographic research on Design problems, and we mainly met two important concepts, that is the Object Oriented Modelisation for the intrinsic description of the system, and the constraints modelisation for the description of the interactions between the different objects of the system. We finally became persuaded that it should be interesting to use Artificial Intelligence tools to solve this kind of problems.

The first step in the resolution of such problems is the creation of a model of the analized system, that is for us the ship. For this goal, we use an Object Oriented Approach. The second step to reach is the definition of the constraints on these objects, and we use for that a Rule Oriented Approach.

Our solution today

We have already made some software developments with these considerations, with the Expert Shell Development Tool Inference ART^{TM}. However, there is no connection with any CAD system yet.

The objects

Our system is composed of three different kinds of objects : parts of the structure of the ship, relevant areas defined on the ship, and equipments. The ship is represented by a hierarchy of objects, in terms of schemata.

In the one hand, the ship is mathematically divided into several areas organised in a top-down tree, and in the other hand, these parts involve the definition of implicit areas organised in a top-down graph. Then both are combined by intersecting overlayed areas, and the result is a complex top-down graph.

The constraints

In our system, three kinds of contraints are requested : constraints on objects, constraints between several objects, and constraints between objects and constraints, that is meta-constraints. They are all represented by rules, caracterized by a priority which is essential for the way the system works.

The search of the solution

According to the constraints, the system tries to place the equipments in the defined areas, and then it evolves in parallel hypothetical worlds, in which all the constraints are satisfied. When it can not evolve any more, it chooses the world containing the lowest priority constraint, and it relaxes it to go on in its progression with this world.

This solving solution which makes possible to relax some constraints in over-constrained situations is one of the most advocated today.

Our future hopes

The problems we discussed above are exactly the same as those of the draughtsmen when they design new mechanical parts or new buildings, and so we believe that those tools should be integrated in the CAD system itself.

The object modelisation would improve the work of the Designer when he defines his system, by enabling him to go faster and further.

This constraints modelisation should consequently improve the efficiency of the Designer, because they would free him from steadily thinking about them, once they will have been defined.

According to us, these advantages should be very interesting.

OUR INTEREST

As to sum up our position towards the ICAD, we can say that one of our main interest is the use of Artificial Intelligence in CAD systems. We really think that the constraints modelisation combined with the object oriented description will be the next step of CAD systems towards Intelligent behaviour, and so, it is in this way that we want to conduct our future projects.

Consequently, we are particularly interested in Experiments with intelligent CAD systems, Software engineering for implementations of intelligent CAD systems and Knowledge representation languages for design.

Wim Eshuis, Dirk Soede, Per Spilling

A user controlled process planning system

A USER CONTROLLED PROCESS PLANNING SYSTEM

Wim Eshuis, Dirk Soede, Per Spilling
CWI, Amsterdam.

ABSTRACT

This position paper gives a short discription of what has to be understood with a user controlled system, and further a discription of the project where we want to apply this method.

Keywords: UIMS, user controlled / user programmed systems, CAPP.

1. USER CONTROLLED SYSTEMS

Project IS5 "User Controlled Systems" is an application oriented working group of the IS department of the CWI (Interactive Systems dep. of the Centre for Mathematics and Computer Science) manned by Wim Eshuis, Paul J.W. ten Hagen, Dirk Soede and Per Spilling.

User Controlled Systems (UCS in the sequel) are information systems for which all tasks are command driven. Meta-level commands allow the definition of new tasks. The task oriented architecture requires a high degree of integration among program libraries and databases. In addition this task oriented approach must bring about comprehensible user interfaces for very complex systems, such as CIM or Integrated CAD systems. UCS embody the information technology needed for design, operation and maintainance of complex systems. According to the concept of UCS, standards are designed for the exchange of data and for external control. These standards are a combination of function schemata and data formats. A UCS enables a user to become an on-line programmer of a system; a user can interactively compose programs from process activations and database transactions. A new dimension is added by allowing the building blocks to interact mutually and with the composer, optionally under interactive control of the latter.

Examples of applications of UCS are: a datascheme for design information together with a collection of pre- and post-processors; the description of an object to be manufactured, and the set of manufacturing processes together with a procedure to select an optimal process. Each example shows aspects of integration, distribution and contemporal multiple users. The view on the system has to change with the nature of the usage or its user. Particularly interactive usage requires a flexible distribution of transactions and decisions between user and system.

2. THE PART-PROJECT

A first realization of a UCS will be PART (Planning of Activities, Resources and Technology), a process planning system developed together with the Mechanical Engineering Department of the University of Twente. PART is a process-planning system for the metal industry, and is mainly concerned with the small and medium sized industry where the production is done in small batches. The project was started in 1987 and is scheduled to be finished in 1991.

PART is open ended on both sides wich means that in a later stage it may evolve to become the second link in the CIM-chain: CAD - CAPP - CAM. The PART system consists of 10 main modules (CAD interface, Volume Editor, Feature Recognition, Machine Tool Selection, Jigs & Fixtures, Machining Methods, Tool Selection, Cutting Conditions, NC Output Compiler and Planning) a database and a supervisor. In addition there is a scenario editor and a database tuning module. An overview of the system is given in fig. 1.

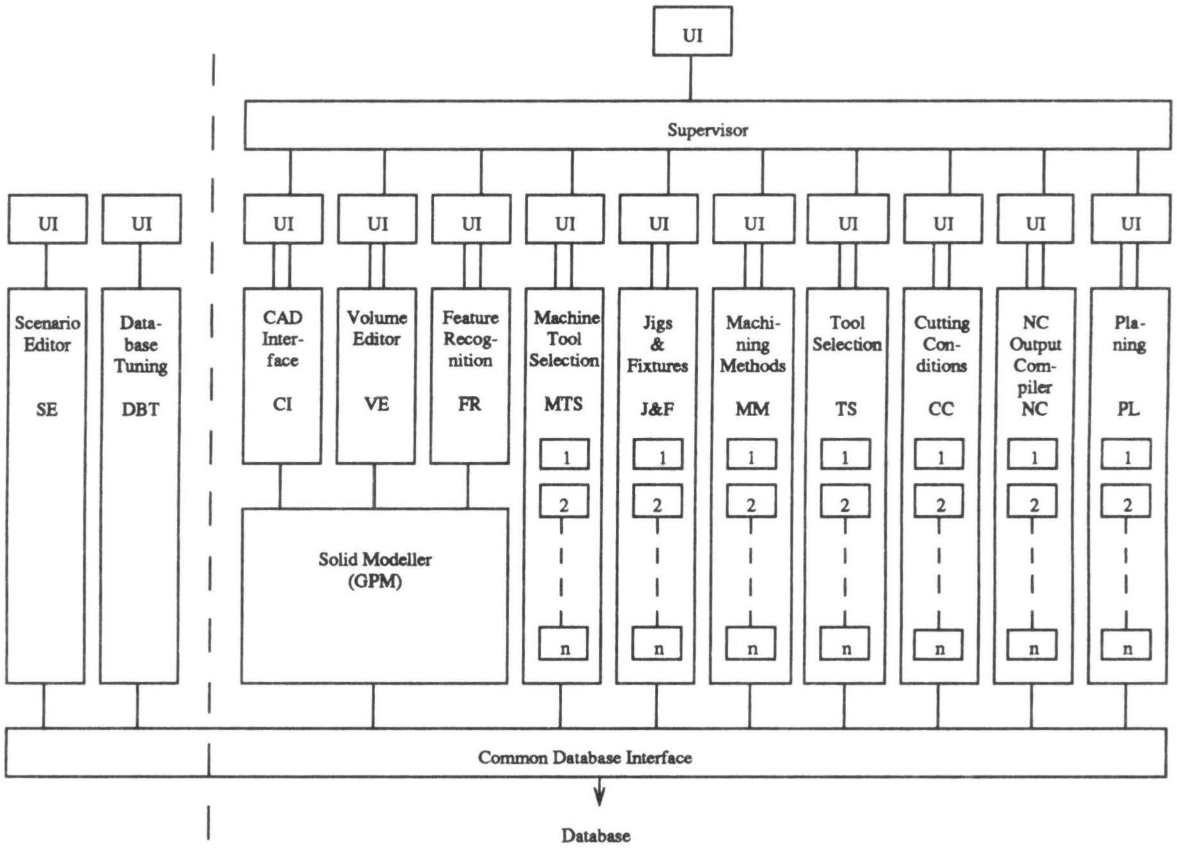


Fig. 1: Schematic representation of the PART system.

PART's architecture is a three level hierarchy consisting of:

- the supervisor

- the 10 modules

- the phases, a further functional subdivision of the modules (see fig. 1).

Input to the system comes in the form of a product model containing geometrical and technological data, and a scenario (or possibly several scenario's). In the current setup the product model is produced by a boundary representation solid modeller (GPM).

The scenario's are made by the scenario editor. The phases are the building blocks of scenario's, which may be understood as a user defined way to walk through the system. In the scenario's the user can specify which factors he/she wants to give priorities to. These factors can be such things as:

- product specifications (tolerances, etc.)

- delivery deadline

- machine load

- minimalization of tool change

- tool availability

The supervisor's task is to dynamically allocate processors to the phases (assuming that the system is being used in a distributed environment), and to start and monitor the execution of the phases.

Our task in the project is to develop and implement the user interfaces, the supervisor, and.if time permits an interactive graphical scenario editor. Supervisor, User Interfaces and the scenario editor will all be specified as DICE (Dialogue Cells) programs, built upon an application (i.e. PART) dependent basic cell library with eg. 3D input techniques for the Volume Editor module. This library aids in achieving uniform user interfaces.

AI techniques are used in several of the modules in the system, and for our part of the project we are considering using AI techniques for the implementation of the scenario editor.

3. IMPACT of the WORKSHOP

From the project description it may be clear that we are involved in:

- User Interfaces for Intelligent Systems,

- Integration of application software,

- User controlled / user programmed systems, and

- AI techniques in process planning.

We think that the workshop can give us valuable input on these matters.

REFERENCES

[1] Kals, H.J.J., van Houten, F.J.A.M., et al., "Een werkvoorbereidingssyteem", research proposal for the PART project.

[2] ten Hagen, P.J.W., van Liere, R., "Introduction to Dialogue Cells", CWI report CS-R8703, 1987.

[3] van Liere, R., Schouten, H.J., "The DICE language and Cookbook" (in preparation).

[4] van 't Erve, A.H., "Generative Computer Aided Process Planning for Part Manufacturing, an expert system approach", PhD. thesis, Department of Mechanical Engineering, University of Twente, 1988.

[5] van Houten, F.J.A.M., van't Erve, A.H., Kals, H.J.J., "PART, a Feature Based CAPP System", report PA89-008, University of Twente, 1989.

Josef Hofer-Alfeis and Klaus-Peter Sondergeld

Intelligent CAD Systems for Mechanical Engineering a general view

Munich, Nov. 7th, 1988

Dr. Josef Hofer-Alfeis and Dr. Klaus-Peter Sondergeld
Siemens AG
Otto-Hahn-Ring 6
8000 München 83
Phone: 089-636-47100 and 089-636-49846

Position Paper

for the Third Eurographics Workshop on Intelligent CAD Systems, Texel, The Netherlands, April 3 - 7, 1989:

Intelligent CAD Systems for Mechanical Engineering - a General View

1. Design as a Strategic Factor in an Integrated Production Process

The increasing degrees of automation in manufacturing, assembly and quality assurance make the different steps of the production process more and more interrelated and necessitate an integrative planning of the whole process in which product design is the most cost-decisive step.

An intelligent CAD system for mechanical engineering should be integrally imbedded in a more comprehensive design system (comprising e.g. a design system for the electrical components) satisfying all the requirements for an adequate integration of design in the CIM process.

The integration of design into a CIM system goes beyond the requirement of an uninterrupted data flow from design via manufacturing planning to manufacturing and testing, without reentering or recalculating data which are available from previous stages.
The integration of design should rather be considered as a mean to make design-for-production possible. This requires an information system with the capability of transfering knowledge, from design to manufacturing as well as backwards. In particular, the information system should support an appropriate participation of the manufacturing engineer in the design process.

2. Intelligent Information Technology as a Requirement for a Cost-effective Design-for-Production

In a cost-effective design, the functional specifications must be realised, and at the same time the requirements of the following production steps must be forseen and met as best as possible. In order to do so, the design engineer must have easy access to that part of the manufacturing knowledge which is relevant to his design.

In traditional design, the relevant knowledge is either not available in explicit form (e.g. rules of thumb) or too difficult to access. The existing catalogues of design rules and construction guide lines are not flexible enough because they miss intelligent interfaces and in particular do not support questionnaires with incomplete or fuzzy data. In addition, it requires a very experienced and well slept-out designer to know at each stage of his design which aspect of production is relevant at the particular stage and which sort of knowledge source should therefore be consulted.

This inadequate support by present CAD systems entails that deficiencies of a design are not detected immediately by the designer, but instead many feeedback cycles from manufacturing planning and even manufacturing back to design become necessary. Our work in knowledge-based design support has the intention of reducing the number of these time-consuming and costly iterations considerably.

The tight interrelationship of design and manufacturing - and design-for-production as a goal resulting from it - is best taken into account by organising the modelling process and the corresponding data management using object oriented data base and programming techniques - in particular if the product is complex with hierarchical parts structure and many functional and technological relations among the parts and subassemblies.

More recently, design systems have appeared on the market which are based on these principles (e.g. *ICAD* by ICAD Inc., Boston and *Concept Modeller* by Wisdom Systems, a McDermott company). At present, it still appears difficult to develop a transition strategy from traditional data organisation to these new concepts. The integration of the huge amount of existing data and programs having developed over decades is feared to be one problem. Another problem, probably more important, will arise from the fact that a new approach to design -even if it corresponds very well to the mental process of an expert designer - still requires a considerable learning process, in particular on part of the average engineer.

3. Knowledge Based Systems as "Smart Design Assistants"

For the above-mentioned reasons, we restrict ourselves at the moment to a more traditional approach on our way towards the goal of design-for-production. In particular, we do not touch the traditional sequential ordering of the production process and the corresponding information process.
Furthermore, the dialogue of the designer with a traditional CAD system continues to play the predominant role in mechanical product design.
In our envisioned engineering environment, the knowledge of the succeeding production stages is provided by various engineering information systems. These could be data bases (e.g. containing the available tools), computational methods (e.g. stability evaluation via FEM) or expert systems (e.g. simulation of a particular manufacturing step).
However, we do not expect of the designer to use these informations systems in a merely conversational way. This would require the superdesigner who always knows what to ask for. Instead, we provide an intermediate level with one or more "design assistants" which accept very general orders from the designer (s. Fig.1). E.g. there could be assistants for cost aspects, manufacturability, assemblibility or functionality. The design assistants must be capable of giving advice in the course of the design based on incomplete geometric and functional models. The latter is in contrast to existing solutions which analyse the completed design (e.g. AEM by HITACHI for cost evaluation or the method of Boothroyd/ Dewhurst for assemblibility).

4. Design-for-Assembly as an Application with High Potential of Rationalisation

The design of parts is not so much of a problem: for certain classes of parts, integrated CAD/CAPP processes with automatic generation of NC programs directly from design data have already been realised. Furthermore, prefabrication is not so important for SIEMENS because of its low "depth of fabrication", i.e. 50% and more of the manufacturing costs result from assembly. Thirdly, the degree of automation in assembly is low, mainly due to the fact that the products are not adequately designed for assembly automation.
For these reasons, our first priority is the development of a Design Assistant supporting design-for-assembly.

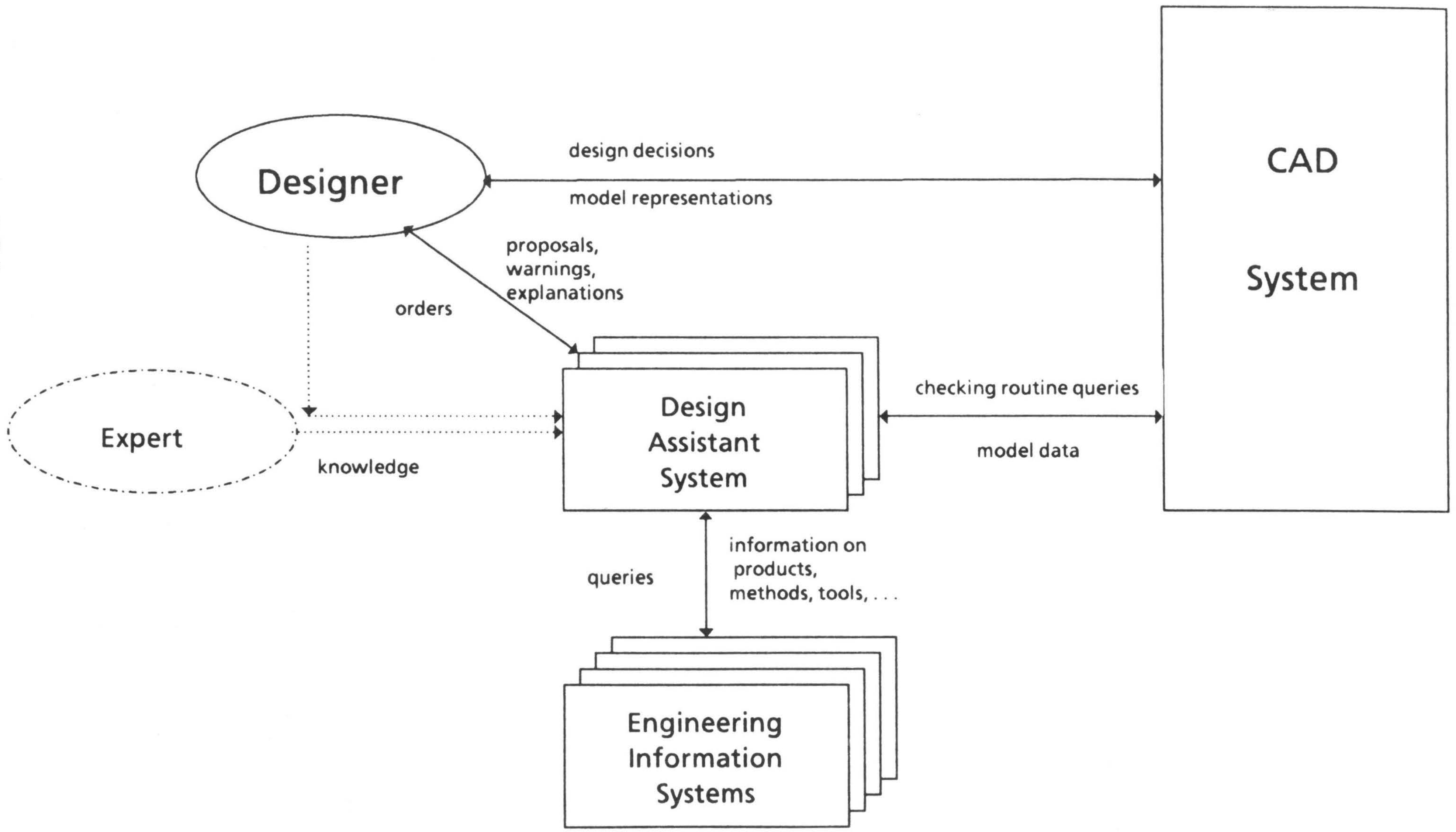


Fig.1: Knowledge-based systems to support designer and CAD: Information flow and Tasks for an "Design Assistant Shell"

Chang-Goo Kang, Kyung-Hyune Rhee

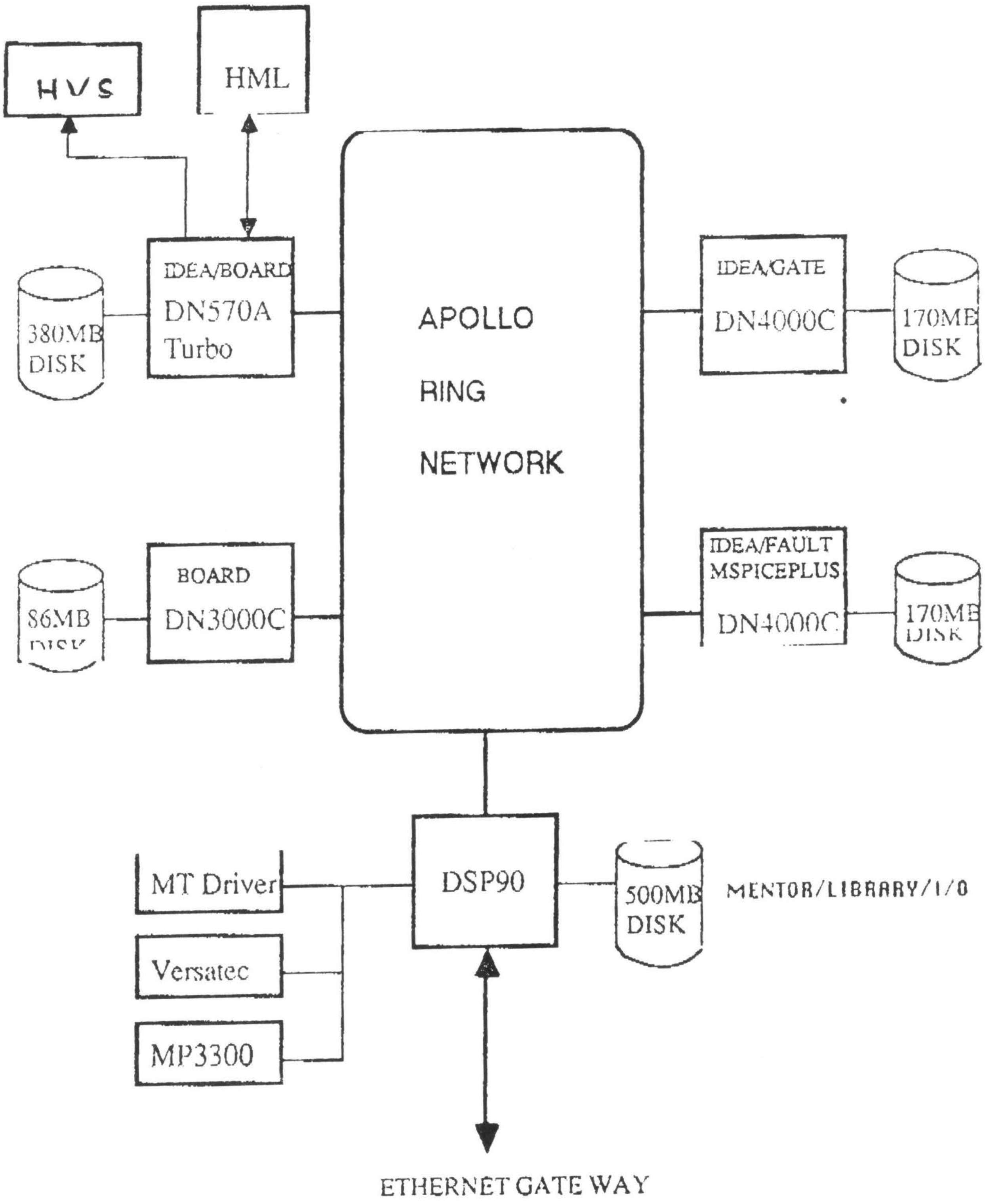
POSITION PAPER

1. REFERENCES

Attendees	CHANG - GOO KANG (Author) KYUNG - HYUNE RHEE
Status	Senior System Designer
System	Communication System
Address	E.T.R.I. (Electronics and Telecommunications Research Institute). SECTION 4120 P.O. BOX 8, DAEDOG SCIENCE TOWN CHUNG-NAM, 302-350 KOREA

2. Design Environment

2.1 CAD/CAE System Configuration



2. 2 System Description

- DN4000C IDEA Station
  - Digital/Analog Circuit Design
  - Digital/Analog Logic Simulation
  - Fault Simulation
  - Gate Array Design
- DN570A IDEA Station
  - Digital Logic Design
  - Logic Simulation
  - PCB Design
- DN3000C BORAD Station
  - Digital Logic Design
  - PCB Design
  - Standard Cell Design
- HML (Hardware Modelling Library)
  - Custom IC Real Chip Simulation
  - VLSI Processor Simulation
- HVS (Hardware Verification System)
  - PCB Prototype Test
  - Custom IC Prototype Test

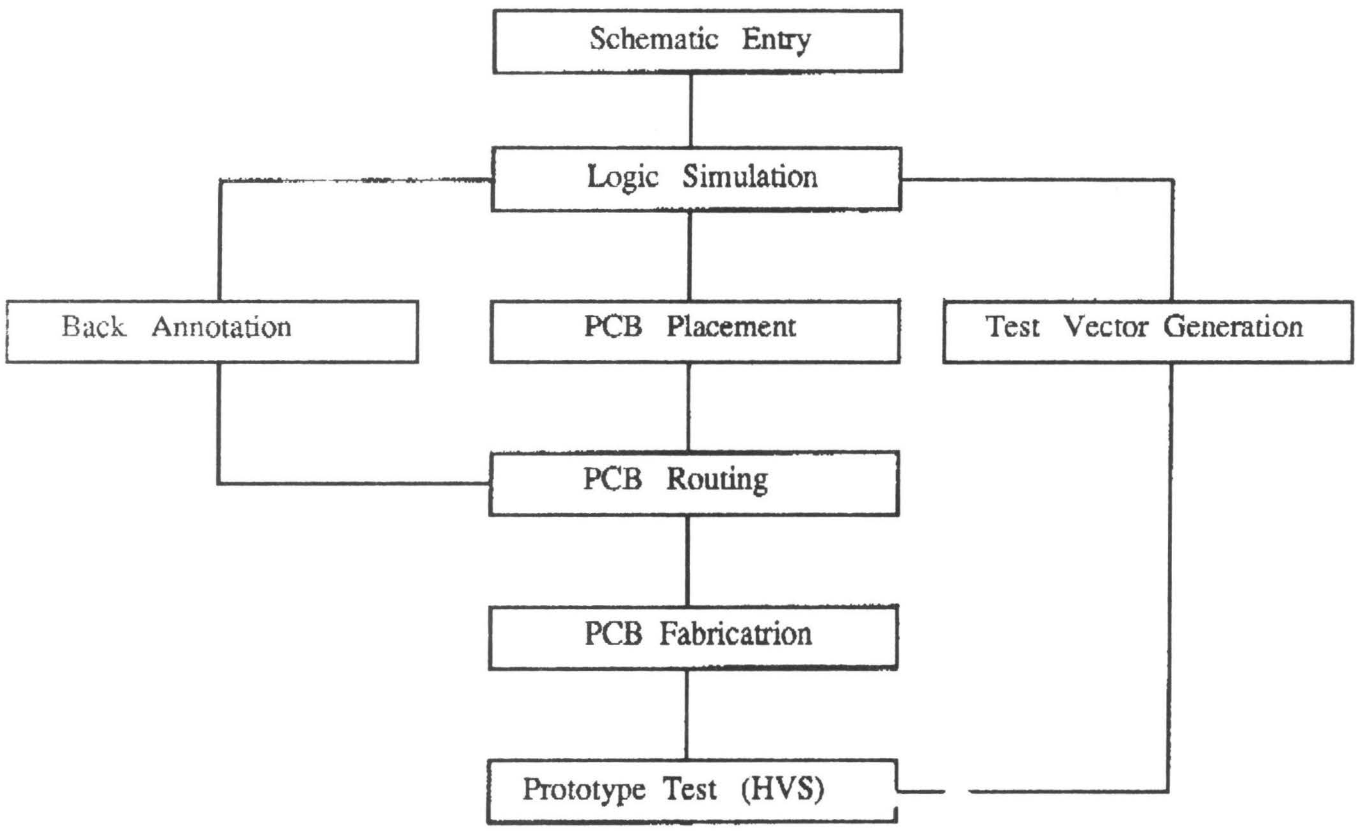
2. 3 Using Tools

- Design Tool
  - Schemetic Entry Tool (NETED)
  - Symbolic Editor (SYMED)
- Modelling Tool
  - Behavioral Language Modelling (BLM)
  - Hardware Modelling Library (HLM)
- Simulation Tool
  - Digital Logic Simulator (QUICKSIM)
  - Timing Simulator (CPA/TVER)
  - Fault Simulator (QUICKFAULT)
  - Analog Simulator (MSPICE - PLUS)
- PCB Layout Tool
  - Layout
  - Librarian
  - Fablink

3. Experiments

3.1 PCB Design

○ PCB Design Flow



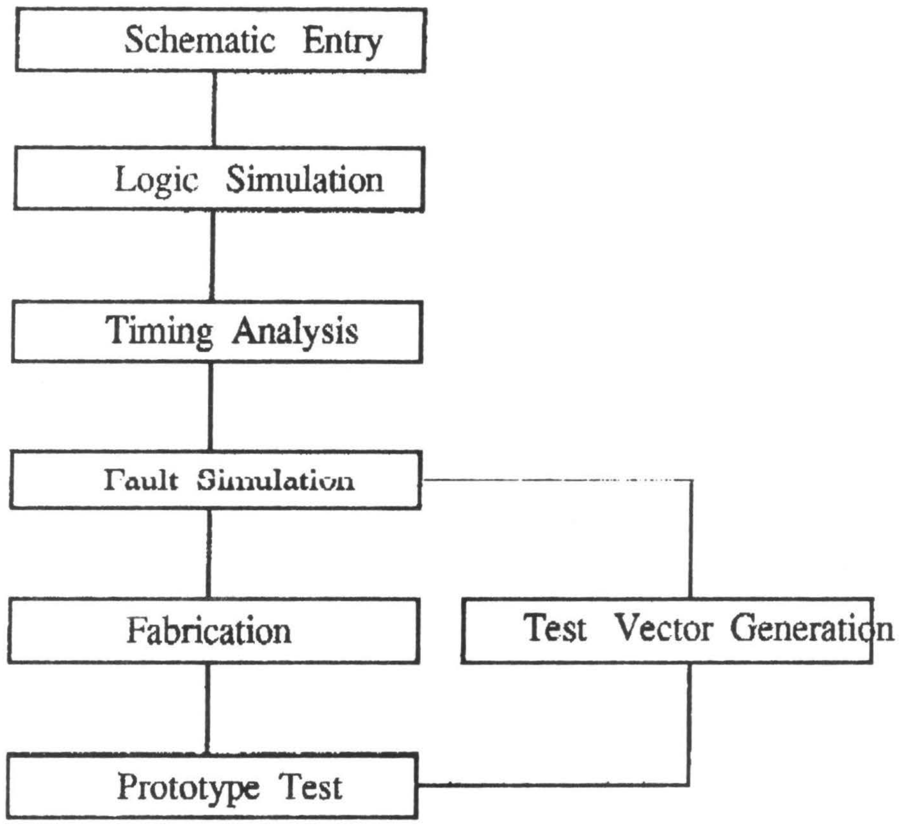
○ Results

- Twenty Kinds of PCB for Communication System

- 4 Layer / 2 Layer PCB

3. 2 ASIC Design

○ ASIC Design Flow



○ Results

Application	Gate Density	Process Technology	Design Approach	Development Year
DSP 1 Chip	5, 000	2 um	Gate Array	1986
DSP 2 Chip	6, 000	2 um	Gate Array	1987
DSP 3 Chip	60, 000	2 um	Full Custom	1988
DSP 4 Chip	50, 000	1. 2 um	Standard Cell	1989

- Gate Array Layout Tool
  - Gate Graph
  - Gate Placer
  - Gate Router
- Standard Cell Layout Tool
  - Cell Graph
  - Cell Placer
  - Gate Router
- Test Tool
  - HVS/Logic Master (IC/Board)
- Analysis Tool
  - Package Station (IC/Board)
- Others
  - Tool Supplier : Apollo (H/W), Mentor Graphics (S/W)
  - Library Supplier : Mentor Graphics,

    National Semiconductor Maker (GSS, SST, etc)

4. REGARDINGS ON THE ISSUES

1. The Role of Intelligent CAD Systems in Design
2. Software Engineering for Implementations of Intelligent CAD Systems
3. User Interfaces for Intelligent CAD Systems
4. How to Use Intelligent CAD Systems
5. Integrations of Application Software to Intelligent CAD Systems.

Jonathan McCullough

Survey Paper on Intelligent CAD Issues

CHALMERS UNIVERSITY OF TECHNOLOGY
School of Architecture
Department of Computer Aided Design

Survey Paper on Intelligent CAD Issues

Submitted for participation in the Third Eurographics Workshop on Intelligent CAD
Texel, The Netherlands, April 3-7, 1989

Jonathan McCullough
Department of CAD
School of Architecture
Chalmers University of Technology
Gothenburg
Sweden

1. Introduction

The paper starts with a study of the major goals of ICAD systems.. This is followed by an examination of the deficiencies of current CAD systems, in order to anticipate some of the expected functionalities of future ICAD systems. Within this the contributions of design theory and methodology to ICAD are assessed, and the envisaged role of ICAD systems as intelligent assistants or autonomous systems is investigated. An analysis of the major ICAD 'issues' is then presented: for example, the expert systems 'debate', user interfaces for ICAD, and the role of meta-level design.

The main interest of the author is architectural design and the development of intelligent computer aided architectural design (ICAAD) systems. Therefore the influence of domain dependent factors (in this case from the architectural domain) on the design of ICAD systems is discussed in the final section.

2. Goals of Intelligent CAD

In general terms the goal of ICAD should be an improvement in the level of design competence of *human* designers. This will be reflected by the exploration of more design alternatives in the time constraints under which the designer is working, and by allowing more complete testing of the solutions proposed by the system.

The key question is whether this improvement in design competence is best achieved by the development of design apprentices/assistants or autonomous design systems. The latter approach certainly does not encourage the development of human designers, but, as Tomiyama and ten Hagen point out [1], the autonomous approach is more likely to be applied to well known and clearly defined domains. However, it should be noted that while certain domains may be well defined (say in comparison with architectural design), they are still subject to change (for example, VLSI design). By their very nature autonomous design systems are somewhat inflexible, and cannot easily adapt (or be adapted) to changes in the nature of their domains. For this reason, the design of fully autonomous systems is questionable. Autonomous systems in the domain of architectural design have been challenged by Pohl and Chapman [2]: '... we cannot forsee an intelligent CAD system that can accomplish, on its own, any building design that is beyond the capabilities of an architect'. It has been suggested by Bijl in a discussion session in [1] that there are no substantial differences between the two approaches as the apprentice has to know too much about the designer. He claims "We do not simplify the problem by thinking of a machine as an assistant". It is the author's belief that, while we do not simplify the *design problem* by viewing the machine as an assistant, we do make more realistic assumptions of the relationship between the designer and his or her tools, the implicit limitations of those tools in terms of their depth of knowledge about the domain, and their abilities to adapt to domain changes.

Before leaving the issue of the goals of design, it is necessary to look at the goals in residual terms. It should not necessarily be the goal of ICAD systems to permit the design of more complex objects. In the context of architecture, more complex buildings are obviously not necessarily better buildings. Some authors have fallen into the trap of comparing the complexity of objects across different domains, as can be seen in the following dangerous comparison [3]:

> '...the complexity of buildings has been virtually unaffected by the use of [computer systems in architectural practices]. In comparison, the use of computer aided design systems by electrical engineers has enabled them to increase the complexity of integrated circuits by several orders of magnitude, while significantly reducing their design time.'

What ICAD systems should assist in, is the handling of design process complexity, by increasing the number of variables the designer *can* cope with efficiently if he or she *has to* or *wants to* - and there should be a choice in certain contexts. Arguments suggesting that it is always positive to maximize the number of constraints a designer is handling, especially in the early stages of the design process, do not always hold. For example, an architect may wish to concentrate on the aesthetic aspects of a building design, reducing the weighting of other constraints to negligible levels, but applying constraint filtering progressively as ideas develop.

Another goal of ICAD systems which could be proposed would be that of reducing the number of iterations in the design process. However, in some design domains (certainly architecture) the degree of iteration in the design process is more a measure of individual style than design competence. Without doubt expensive iteration should be minimized (for example, logic simulations of circuit design), but it should be borne in mind that not all iteration is expensive. Some iteration in the design process sometimes improves the chance of the designer making creative associations that would otherwise have been missed.

3. Current problems with CAD

While partly refering to the potential contribution of artificial intelligence techniques to the future development of CAD, in some respects the word 'intelligence' is superfluous to the title 'intelligent CAD'. The very nature of design implies intelligent activity, although this fact is often inadequately reflected in existing CAD systems.

One of the major problems which is recognised by most authors lamenting the current state of CAD systems is the tendency for CAD systems to dominate the design process. Bijl [4] claims that on one level the CAD system is programmed, and on the other, the unfortunate consequences are that 'we are programming the user to conform to the anticipation of the user world built into the computer program'.

The study of ICAD must have a stronger base in design theory than is evidenced in current systems. At present there is too much of an emphasis on the representation of the objects or artifacts of design, and not enough on the actual process of design. The representation of the final solution seems to have much greater priority over the representation of the development of that solution.

What pointers has design theory provided? Few clues are offered by Alexander [5] in describing design as a process whose objective is to produce forms with no major mismatch with the environment. First of all the environment is often not very well defined, and is generally changing over time. What else has been said? Bijl [6] prefers to see design as 'an activity of event exploration in which partial responses lead to a redefinition of a goal'. Here design is seen as the concurrent evolution of both design specifications and design descriptions, and this perspective appears to be reflected in some models of ICAD [7]. Logan [8] and Simon [9] confirm the need for suitable representation of intermediate specifications, by seeing many design problems as having no inherent structure, but acquiring structure as solutions are proposed. The issue of representation is dealt with later.

3.1 Integration Issues

Integration is a key issue in the future development of CAD systems. There are two directions in which the integration of CAD systems can viewed: in terms of the integration of models and applications of a given CAD system (vertical integration), and in terms of the flow of information between different task domains to facilitate the sharing of information between users who are contributing to the design process (horizontal integration).

Suitable integration in the vertical direction would appear to be a prerequisite for effective horizontal integration. For this reason, it is the author's view that considerable effort should be put into providing a firm conceptual basis for vertical integration. Mechanical data exchange is inadequate as it causes a deterioration in meaning. In theoretical terms this loss of information is a side effect of conversion between intensional and extensional descriptions, for example consider the loss of information resulting from the transformation between a CSG solid modelling system description and a boundary representation description [10]. The implication of this is the need for the development of unified 'metamodel' [11] descriptions as a central model through which different representations can 'communicate', and through which a solution descriptions can evolve.

In the discussions surrounding integration, emphasis seems to be placed on the 'coupling' of systems, and the term seems to be applied in a number of quite diverse contexts. In some cases this coupling refers to a union of expert systems and database concepts [12], whereas in others it means a union of symbolic reasoning and numeric operations. It is important to question which forms of coupling are useful, and what types of coupled systems should be replaced by a single system with a common language formalism. The convergence of database theory and knowledge representation ideas seems to suggest that for some systems a common formalism may be more appropriate

As regards horizontal integration, it will certainly cause changing roles within industries. For example within the construction industry one possible result may be to change the role of the architect to one of greater involvement with a building throughout its entire lifetime (through activities such as facility management), as opposed to just having the responsibility for the creation of the building. Another effect of horizontal integration will hopefully be improved cooperation and communication between the different members of design teams. This applies as much to the construction industry as to VLSI design.

3.2 Partial and Approximate Information

The lack of competence of CAD systems in handling partial information can be seen to indirectly affect the design process, reducing the impact of many CAD systems to the level of drafting (and hence documentation) tools [13].

> '...these systems generally cannot tolerate partial information. Thus before a user can tell a CAD system about a point, line, surface or solid, he must precisely know the attributes (eg., coodinates, slopes, control points, lengths etc.) that *uniquely* define that entity. It can be argued that by the time that a concept is broken down to such level of detail, and each detail is known so specifically, the real design, or at least a good deal of it is done. From this perspective, existing CAD systems, including those based on solid modeling, merely aid the *documentation* of a design process that takes place in the mind of the designer'.

The unjustified demand for detail in the early stages of design can also be said to encourage 'bottom-up' design, whereas the design process, especially in the context of architectural design, should accomodate variation between both 'bottom-up' and 'top-down' approaches.

Related to the issue of partial information is the claim that CAD systems, especially in the early stages of the design process, should be able to accomodate approximation via some form of 'pseudo-sketching' technique. This is a question connected with the semantics of drawings; it is common to associate drawing precision with the level of effort required to produce the drawing, which is usually associated with a particular stage in the design process. However, given the advances in the graphics industry in both screen and paper presentation techniques (high resolution, bit mapped workstation screens and laser printer and plotter output), this direct mapping between presentation and effort involved in producing a drawing no longer holds. It can be shown that considerable effort is required to produce drawings of 'back of an envelope sketching realism' using current systems. As a result of this breakdown in the presentation to effort mapping, it becomes much easier to misread 'early sketches' as final working drawings. This can affect both designers and those to whom the designer is attempting to communicate ideas. Some attempts have been made to deal with this question of representing approximation in drawings, but approaches to date have proven inadequate. In one case the degree of approximation of lines was directly represented by line intensity or colour, however the visual side effects of this proved too distracting. An attempt to convey the approximate nature of spatial layouts was achieved by drawing a group of lines (instead of one boundary line) whose endpoints had small but random variations with the given endpoints of the boundary line [14]. It is obvious that much more work needs to be done with this particular aspect of the drawing semantics.

3.3 Data input considerations

Most conventional CAD systems provide some level of error checking, for example, format checks on user input. However this implies basically a form of 'passive assistance' on the part of the system, in that the designer must input the data. It would be more valuable to encourage 'active assistance' [15] on the part of the system to minimize the amount of information which the designer is expected to provide, by using contextual information which is held in the system. For example, if a designer specifies a material for part of a device, the system should be able to help the user at some stage by automatically understanding that the material has certain properties and performance attributes by reference to encoded knowledge.

3.4 Prototyping

It has been argued that all design can be seen as 'prototype modification', and that the distinction between innovative design and design by modification is only caused by the quality and nature of the prototypes, and the modifications involved [16]. The best prototypes are those with the highest number of potentially modifiable features.

It would appear that in the context of ICAD, more attention needs to be paid to the development of powerful prototypes and primitives. Lansdown confirms this by considering the very limited range of primitives which are provided in many solid modelling packages. There must be greater appreciation of which primitives on are more likely to be used in which domains, as well as the limitations of these primitives.

4. Some ICAD Issues

4.1 User Interfaces for ICAD

It has been suggested that the goal of intelligent CAD systems is not just to make intelligence available, but to actually stimulate intelligent behaviour on the part of the designer. The user interface for an ICAD system should thus act as a vehicle for such intelligent behaviour.

To solve his design problem the designer must not only deal with complex concepts and phenomena he or she is attempting to understand, but also with the complicated tools which help in that understanding. The role of the user interface should be to attempt to maximize ease the use of those tools, but the fact that complex design involves the use of complex tools should not be avoided. The wrong emphasis is in attempting to develop 'simple to use' ICAD systems, as there is the danger that they will lack the depth required to perform the required design tasks. More emphasis should be placed on getting the correct conceptual basis for ICAD, and accomodating this in a user interface which matches the needs and experiences of the user.

One important issue is whether interfaces contain intelligence in their own right, or if they are merely acting as interfaces to intelligent systems. The author's view would be that the intelligent interfaces for intelligent CAD systems are possible. The justification for this is provided by user interface systems which have been built to be sensitive to the context of communication. Ruttkay's [17] user interface 'provides more flexible communication than a static mapping of i/o actions and communication actions, as it depends on the context of communication: different media can be used to define or visualize a communication action; the presentations and their usage can be dynamically altered according to the user's and the design system's need'. Naturally the fact that the user interface system makes use of knowledge engineering tools does not automatically justify the label 'intelligent', but it should be said that the system does attempt to incorporate rule based production systems and a blackboard mechanism as a means of coping with the context problem.

Error handling will be an important aspect of the user interface side of ICAD systems. It is crucial that the innovative aspects of design are not inhibited by the continual interuption of error warnings by the user interface in cases where the designer is, through the expression of new concepts, redefining what constitutes an error or what justifies a warning. Facilities for the controlled suppression of 'suggestions' and 'warnings' must be provided. All this necessitates a more complex (even intelligent) model of the error handling and warning facilities.

The importance of the visual aspects of user interfaces for ICAD systems should not be underestimated. This is emphasized by Nadin and Novak [18] when they talk of 'the primacy of the visual' and point out that as well as being essentially concurrent, 'the visual representation allows designers to concentrate on the form of the design, where form is not used to describe the surface characteristics of the evolving design, but the abstract structure'. Multiple representations of objects in combination with high quality visualization techniques based on emerging technologies (for example, holographic imaging in the long term) will be essential to the development of visually rich interfaces for intelligent CAD. The importance of multiple views of the design object can be indicated by quoting various design traps into which designer can fall if only a single view of the designed object is available, for example, in the domain of architecture it has been shown that for 3D drawings there is more likely to be symmetry about the axis running forwards away from the designer. This is a case of limited views imposing their own grammar on the design and can be used to justify the idea of multiple views.

4.2 ICAD System Architecture and the Expert Systems Debate

Even in the key ICAD literature [1,19,20,21] there are quite a broad range of opinions on the role that expert systems will play in the future development of ICAD systems. On one hand there are the 'grand designs' such as the IIICAD and MOLE systems [4,6,10,11,22] which reflect a concentration on the conceptual foundations and formal approaches to the development of total environments for ICAD, and on the other there is the approach which suggests that design can be handled by a collection of communicating expert systems [7,23]. These are referred to as the 'framework' approach and the 'intelligent tools' approach respectively [11]. Some proponents of the former approach (for example the CWI group) dismiss the latter approach claiming that it provides no overall framework for design, only a collection of tools providing *ad hoc* functionalities through which designer expected to find his or her way. Others choose not to refute expert systems totally but prefer to 'moderate ambitions associated with practical application of these systems' with the following arguments [6]:

> 'When expert systems are imported into the field of design the knowledge which they are intended to handle looks very similar to intuitive knowledge. We should therefore approach expert systems with caution. A weak anticipation of design objects, the lack of prior knowledge of the properties that will describe such objects cannot be translated into the firm goal specifications required by expert systems. As a consequence, while the external products of the intuition of designers (as human

experts) may be conveyed to an expert system it does not follow that the system will be able to employ those products as knowledge whan dealing with a new instance of design. *The role of experts in design is inherently limited to discrete analytic subtasks of design, and cannot contibute to design synthesis.*'

The author would agree with Bijl's view that expert systems are *useful* for 'discrete analytic subtasks' (see [24-29] for examples), but would disagree that they should be *limited* to such subtasks. Not enough research has been done in the area of coordination strategies for communicating expert systems to justify complete rejection of the 'intelligent tools' approach. While the best long term strategy would be the development of ICAD 'frameworks' with a strong formal basis, study of the multi-expert systems approach should continue in the knowledge that the lessons learnt in the short term will contribute to the long term strategy. Contributions are likely in terms of the 'formalisation' of the expert systems approach: this will be achieved in a variety of ways, for example through the development of new knowledge representation formalisms, structured partitioning of rules, greater use of meta-level knowledge and the application of new control mechanisms, as well as a deeper understanding of the distributed expert systems.

One point worth considering is that within the AI community one of the main problems is the lack of precision in terminology. This certainly applies to defintions of expert systems. It is easy to criticse the limitations of the stereotypical 'rule based expert systems' (using such arguments as have been put forward in [30]), forgetting the range of other, often rich, mixed formalisms, which have been adopted for use in expert systems. The relationship between 'knowledge based' approaches and 'expert systems' approaches has become somewhat blurred, the consequence being that some often useful techniques are questioned on the basis of the connotations of their 'label'. While it is obvious that systems such as those consisting of a fusion of object oriented programming and logic programming should be distinguished from basic rule based expert systems, the point at which a formalism rises above the level of the 'mere' expert systems to a knowledge based approach remains unclear, so care should be taken not to reject promising techniques without justification.

One of the realities of expert systems/knowledge-based systems, which should not be ignored, is their current limitations in dealing with 'deep' knowledge. Some expert systems which do not accomodate the modelling of deep knowledge have proven more than adequate for certain applications in commercial and industrial environments, and this should be accepted. However, given the nature of the design process and the use that designers make of deep knowledge through common sense reasoning, it seems reasonable to expect that future ICAD systems will incorporate the modelling of this so called deep knowledge. It is argued [31] that deep knowledge permits logical reasoning and explanation in terms of both the real world and its abstract models - not just in terms of a list of rules used. In particular, justification for the incorporation of commonsense physical knowledge (using the ideas of naive physics and qualitative reasoning) into ICAD systems is also given, for example, design usually results in physical objects, which must interact with the environment and the physical notions, such as force, motion etc., associated with the environment in which the objects are used. Also, to produce reliable design objects which continue to function under adverse conditions, these physical ideas must be taken into account in the design process. (Although the justification is presented in terms of the mechanical design domain, it can also be seen to apply to domains such as the architectural domain). In addition, an appreciation of physical phenomena would appear to be a prerequisite for innovative design. It should be noted that while much effort has been expended in justifying the need for deep knowledge (in particular commonsense physical knowledge) [31,32], very little has been done with the practical integration of these ideas into the context of CAD through the choice of appropriate knowledge representation formalisms.

4.3 Knowledge Representation for ICAD

Given the complex nature of design objects and specifications and the processes operating on them, it is clear that sophistocated and expressive techniques are necessary for the representation of design knowledge in a manner which permits both intuitive and idiosyncratic handling by the designer, and which permit efficiency in processing. While the system must at some level be responsible for maintaining the semantic integrity of the design, there must be accomodation of the 'deferral process' [23] in which the designer defers consideration of some aspects of design to concentrate on others, and works under the assumption that undesigned parts of the artifact are designed. This deferral process, in combination with the need for multiple (often inconsistent) views implies the need to adopt reasoning methods which can operate non-monotonically. The adoption of non-classical logics in the form of three valued logics (true,false,unknown), modal logics (of possibility and necessity) and the use of special operators have been proposed as methods of dealing with the non-monotonicity issue, and beliefs about knowledge [33]. Temporal aspects of non-monotonic logics must also be considered. Lansdown [23] suggests the use of timestamping (as used in the database context) in combination with event calculus.

On a theoretical level it seems easy to talk in terms of these logics as a solution to the representation problems, but the issues of adequate implementation of these logics in combination with other formalisms (such as object oriented programming) still remain.

A rather rough classification of the possible approaches to the question of intelligent CAD and the representation issue has been proposed [34]. It classifies the approaches into: 'fully integrated' intelligent systems completely rewritten using knowledge engineering techniques; mixed systems formed by interfacing existing languages and software packages (for example, Prolog and

CADAM); and finally a mixed language approach (based on, for example, C and Prolog). While the mixed package and language is a useful experiment, the approach seems to hold little promise in the long term, as it carries with it the inherent limits and potential inflexibility in representation of the software package. A fully AI approach seems unlikely - and possibly unwise, as efficient numerical processing, graphics handling and i/o operations are, more often than not, still best handled by procedural languages. From the above discussion, it can be seen that the solution to the representation problem seems more likely to be in the form of mixed language formalisms, and this appears to be where most effort is being directed. The object oriented paradigm seems to have gained popularity, although certain issues such as 'delegation versus inheritance' are still under consideration.

4.4. Meta-level Design

Nadin and Novak [18] emphasize the fact that ICAD systems should encourage designers to enter what psychologists call the 'flow state': a state where the mind is free to move fluidly from concept to concept and during which a person's sense of the passage of time is diminished. It could be argued that at certain points in the design process it would be beneficial for the ICAD system to actually terminate the current flow state in order to force the designer into thinking more consciously (possibly retrospectively) about the design process - to question what has been done and why. This corresponds to Jones' [35] ideas on 'designing designing': 'the conscious direction of part of one's activity and energy, while designing, into the meta-process of designing the process of design'. It may be that this may not necessarily involve exiting the flow state. An example of the application of meta-level design can be seen in the method of retrospective planning as a tool for conceptual design [36]. The technique involves the use of a history mechanism , which provides a trace of the procedures involved in getting the design object to its current state. The history network can be treated as a meta-object, and the design of the meta-object regarded as meta-design, in which the user can be directly involved. From a study of other examples of meta-design [37], not necessarily requiring such direct user involvement, it would appear that meta-design has a crucial role and should be integrated into the development of ICAD systems.

5. Domain dependent factors and ICAD design

It is important to isolate domain dependent factors which will affect the form of the ICAD systems developed to serve the needs of that domain.

It is difficult to put forward domain dependent 'design maxims' for ICAAD systems, such are the idiosynchrasies of approaches to the architectural design process. Architectural design can be viewed as the sculpturing of solids (Le Corbusier), or, more commonly, as spatial modelling and plan manipulation (Mies van de Rohe), or something between these two extremes. Any useful ICAAD system must therefore accomodate both styles.

The way in which space is dealt with is unique to the domain of architecture. In most other domains, the emphasis would tend to be on objects and artifacts, whereas in architecture the semantics of space in terms of relationships between spaces, and between objects and spaces is vital. Akiner [25] would emphasize the spatial modelling side of architecural design: 'Architectural design proposes the assembly of physical objects for the synthesis of space'. It would be fair to say that existing CAD systems do not reflect this emphasis on space. The types of questions architects need to ask in the context of design appraisal often relate to spaces, for example, 'Which rooms does one pass through in order to go from room A to room B?'[25]. However, most geometric modelling systems are more concerned with objects. As Akiner indicates, another drawback with these geometric modellers is their emphasis on the internal dependencies of objects (such as dependencies between vertices, edges and faces, and issues such as the well-formedness of the objects). They tend to exclude the external dependencies, which are more concerned with the relationships of objects/spaces to other objects/spaces , and which are critical to adequate modelling in the architectural domain. Thus, both spatial reasoning and modelling will be vital aspects of the development of ICAAD systems.

References

1. P J W ten Hagen and T Tomiyama (eds),*Intelligent CAD Systems 1 - Theoretical and Methodological Aspects*, Springer-Verlag, Berlin, 1987

2. Pohl J and Chapman A, 'Expert Systems for Architectural Design', Working Paper, April 1987, CAD Research Unit, School of Architecture and Environmental Design, California Polytechnic State University

3. Kalay Y E, 'Redefining the role of computers in architecture: from drafting/modelling tools to knowledge-based design assistants', *Computer Aided Design*, Vol. 17, No. 7, September 1985

4. Bijl A, 'Architects and Computers: A Human Approach', *Proceedings of the SAR conference*, 1983

5. Alexander C, *Notes on the Synthesis of Form*, 1964, McGraw-Hill, New York

6. Bijl A, 'Designing with words and pictures in a logic modelling environment', in *Computer Aided Architectural Design Futures*, Pipes A (ed) Butterworths, 1986

7. David B T, 'Multi-Experts for CAD', in *Intelligent CAD Systems 1 - Theoretical and Methodological Aspects*, P J W ten Hagen and T Tomiyama (eds), Springer-Verlag, Berlin, 1987

8. Logan B S, 'Representing the structure of design problems', in *Computer Aided Architectural Design Futures*, Pipes A (ed) Butterworths, 1986

9. Simon H A, 'The Structure of Ill-structured problems', in *'Developments in Design Methodology'*, Cross N (ed), John Wiley & Sons, 1984

10. Tomiyama T and ten Hagen P J W, 'The concept of intelligent integrated interactive CAD systems', Department of Interactive Systems, Centre for Mathematics and Computer Science, Amsterdam, Report CS-R8717, April 1987

11. Akman V et al., 'Design as a formal knowledge engineered activity', Report CS-R8744,, Department of Interactive Systems, Centre for Mathematics and Computer Science, Amsterdam, September 1987

12. Rehak D R and Howard H C, 'Interfacing expert systems with design databases in integrated CAD systems', *Computer Aided Design*, Vol. 17, No. 9, November 1985

13. Arbab F, 'A Paradigm for Intelligent CAD', in *Intelligent CAD Systems 1 - Theoretical and Methodological Aspects*, P J W ten Hagen and T Tomiyama (eds), Springer-Verlag, Berlin, 1987

14. MacCallum K and Green S, 'THESYS - Implementation of a knowledge-based design system with multiple viewpoints', in *Intelligent CAD Systems 2 - Implementation Issues*, P J W ten Hagen and T Tomiyama (eds), Springer-Verlag, Berlin, 1988 (to appear)

15. Aldridge J et al., 'Expert Assistants for Design', *Applications of AI in Engineering Problems*, Springer-Verlag, 1986

16. Lansdown J, 'The creative aspects of CAD', *Design Studies*, Vol. 8, No. 2, April 1987

17. Ruttkay Z, 'Multi-media presentation in CAD systems', in *Intelligent CAD Systems 2 - Implementation Issues*, P J W ten Hagen and T Tomiyama (eds), Springer-Verlag, Berlin, 1988 (to appear)

18. Nadin N and Novak M, 'MIND: A Design Machine - Coneptual Framework', in *Intelligent CAD Systems 1 - Theoretical and Methodological Aspects*, P J W ten Hagen and T Tomiyama (eds), Springer-Verlag, Berlin, 1987

19. Gero J S (ed), *Expert Systems in Computer Aided Design, Proceedings of IFIP WG5.2 Workshop in 1987, Sydney, Australia*, North-Holland, Amsterdam, 1987

20. Gero J S (ed), *Knowledge Engineering in Computer Aided Design, Proceedings of IFIP WG5.2 Workshop in 1984, Budapest, Hungary*, North-Holland, Amsterdam, 1985

21. P J W ten Hagen and T Tomiyama (eds), *Intelligent CAD Systems 2 - Implementation Issues*, Springer-Verlag, Berlin, 1988 (to appear)

22. Bijl A, 'Strategies for CAD', in *Intelligent CAD Systems 1 - Theoretical and Methodological Aspects*, P J W ten Hagen and T Tomiyama (eds), Springer-Verlag, Berlin, 1987

23. Lansdown J and Roast C, 'The possibilities and problems of knowledge-based systems for design', *Environment and Planning B: Planning and Designing*, Vol. 14, 1987

24. Rosenman M A, Coyne R D, Gero J S, 'Expert Systems for Design Applications', in *Applications of Expert Systems*, Quinlan J R, Addison-Wesley, Sydney, 1987.

25. Akiner V T, 'Topology-1: a knowledge-based system for reasoning about objects and spaces', *Design Studies*, Vol. 7, No. 2, April 1986

26. MacCallum K and Duffy A, 'An expert system for preliminary numerical design modelling', *Design Studies*, Vol. 8, No. 4, October 1987

27. Thomson J V, 'A Water penetration Expert System using Prolog with Graphics', in *Applications of Expert Systems*, Quinlan J R, Addison-Wesley, Sydney, 1987.

28. Maher M L, 'HI-RISE and beyond: directions for expert systems in design', *Computer Aided Design*, Vol. 17, No. 9, November 1985

29. Hutchinson P J et al, 'RETWALL: An Expert System for the selection and preliminary design of earth retaining structures' in *Knowledge-Based Systems*, Vol. 1, No. 1, December 1987

30. Golden M, Siemens R W, Ferguson J C, 'What's wrong with rules?', *Proceedings of the Western Conference on Knowledge Based Engineering and Expert Systems*. 1986

31. Akman V and ten Hagen P J W, 'The Power of Physical Representations', in *Intelligent CAD Systems 2 - Implementation Issues*, P J W ten Hagen and T Tomiyama (eds), Springer-Verlag, Berlin, 1988 (to appear)

32. Green D S and Brown D C, 'Qualitative reasoning during design about shape and fit: a preliminary report', in *Expert Systems in Computer Aided Design, Proceedings of IFIP WG5.2 Workshop in 1987, Sydney, Australia*, Gero J S (ed),North-Holland, Amsterdam, 1987

33. Akman V et al., 'Knowledge engineering in design', Report CS-R8745,, Department of Interactive Systems, Centre for Mathematics and Computer Science, Amsterdam, September 1987

34. Bond A and Soeterman B, 'Integrating Prolog and CADAM to Produce an Intelligent CAD System', in *Proceedings of the Western Conference on Expert Systems, June 1987, Anaheim, California*

35. Jones J C, 'Designing Designing', *Design Studies*, Vol. 1, No. 1, July 1979

36. Takala T, 'Design Transactions and Retrospective Planning - Tools for Conceptual Design', in *Intelligent CAD Systems 2 - Implementation Issues*, P J W ten Hagen and T Tomiyama (eds), Springer-Verlag, Berlin, 1988 (to appear)

37. Coyne R D, 'A Logic Model of Design Synthesis', PhD Thesis, Sydney University, Sydney, Australia

Daan B.M. Otten

On the Role of Delegation and Inheritance in Database Management Systems

385

--

On the Role of Delegation and Inheritance in Database Management Systems

Daan B.M. Otten

Centre for Mathematics and Computer Science (CWI)
Department of Interactive Systems
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
telephone: +31-20-592 4056
usenet: daan@cwi.nl

ABSTRACT

Nowadays, techniques from the area of Artificial Intelligence and Logic (e.g. backward and forward reasoning, reasoning with uncertainty, default reasoning and version control) are used to make current and the next generation of Computer Aided Design systems more capable to represent the designer's complex model of the world and to reason over this model.
Studying the literature on the development of Computer Aided Design systems, we believe that inheritance and delegation are powerful mechanisms to represent knowledge in the field of Computer Aided Design. However, in current systems both mechanisms are only implemented at the user level as tools and are not supported by the basis mechanisms of a Database Management System. We are developing a Database Management System which is based on the use of inheritance and delegation mechanisms and can be integrated in Computer Aided Design systems. When database interactions become a more intelligent activity we expect that a better support can be given to components of the CAD-system which exhaustively use the database, e.g. the user interface and the inference engine.

1. Position Paper

During the last few years, a lot of progress has been made on the development of Computer Aided Design systems (CAD-systems). An important share in this progress originates from the use of Artificial Intelligence techniques and Logic (e.g. backward and forward reasoning, default reasoning, reasoning with uncertainty, version control, multiple worlds). These techniques all have in common that they manipulate knowledge and therefore rely on the Database Management System (DBMS) for the suppliance of it.

From evolution, we have learned that more powerful DBMSs are in response to more expressive conceptual models as known in the field of Computer Aided Design and Computer Aided Manufacturing (CAD/CAM). However, this generation of DBMSs still lacks support for structured objects, data abstraction, version control and multiple worlds.

Studying the literature on the development of new generations of Computer Aided Design systems, we believe that *inheritance* and *delegation* are powerful mechanisms to represent knowledge in the field of Computer Aided Design. Although most of the authors have a slightly different interpretation of both mechanisms, the general thought is quite the same.

Both inheritance and delegation, are based on modeling knowledge according to a certain structure and passing information via the relationships of the structure to provide shared data and behaviour. Inheritance uses an acyclic directed graph structure, follows a set oriented approach to classify objects in which the relationships among objects are less or more fixed, and uses method passing to attain shared behaviour. Delegation is stooled on a network structure, follows a prototypical approach to create objects, allows changing relations at run time, and uses message passing (which is more transparent than method passing is) for shared data and behaviour.

When we are able to give assistence to the inheritance and delegation mechanism at the database level then a better support can be given to components of a CAD-system which exhaustively use the database, e.g. the user interface and the inference engine.