Proceedings of

the Second Eurographics Workshop

on

Intelligent CAD Systems

– *Implementational Issues* –

11–15 April, 1988

Koningshof Conference Centre

Veldhoven, The Netherlands

Centre for Mathematics and Computer Science

Centrum voor Wiskunde en Informatica

Proceedings of
the Second Eurographics Workshop
on
Intelligent CAD Systems

– *Implementational Issues* –

11–15 April, 1988

Koningshof Conference Centre

Veldhoven, The Netherlands

Foreword

Welcome to the *Second Eurographics Workshop on Intelligent CAD Systems* (Workshop theme: *Implementational Issues*). It is a great pleasure to have you here in Koningshof Conference Centre.

The present volume serves as the preliminary record of this workshop and will eventually (hopefully towards the end of this year) be published by Springer-Verlag in the *EurographicSeminars* series. Probably many of you already know that the record of the last year's workshop is now available: *Intelligent CAD Systems I: Theoretical and Methodological Aspects*, Edited by P.J.W. ten Hagen and T. Tomiyama, Springer-Verlag, Heidelberg (1987).

Please note that, in addition to the papers which will be presented during the workshop, there are additional contributions in this volume which will *not* be presented but will appear in the final book.

Even a small gathering such as this demands careful attention to detail and sometimes unwarranted amounts of time of the people who undertook the job of organizing it. We were lucky this year to have the invaluable help of Ms. Marja Hegt, our workshop secretary. Our heartfelt thanks to her for her excellence. In addition to single-handedly managing this year's workshop, Ms. Hegt has succeeded in making a great arrangement for the next workshop: Hotel Opduin on the island Texel! We kindly ask you to take notice of this upcoming workshop and refer you to the preliminary announcement that you'll find in this volume.

Workshop Organizers:

Paul J.W. ten Hagen, Varol Akman, Tetsuo Tomiyama, Paul Veerkamp

Centre for Mathematics and Computer Science (CWI), Amsterdam

TABLE OF CONTENTS

WORKSHOP TIME-TABLE

April 11 (Monday)

PARTICIPANTS ARRIVE .....
16.00 - 18.00 (**registration**)
18.00 - 19.00 (free)
19.00 - 20.30 (dinner)
20.30 - (free)

April 12 (Tuesday)

7.30 - 9.00 (breakfast)
8.45 - 9.45 (**workshop programme committee meeting** and **late registration**)
9.45 - 10.00 Welcome speech by P.C. Baayen, Director of CWI
10.00 - 10.30 Opening speech by co-chairman P.J.W. ten Hagen
10.30 - 10.45 (coffee)
10.45 - 12.15 Paper session *Languages*:

T. Tomiyama (Japan) Object oriented programming paradigm for intelligent CAD systems
J.F. Koegel (U.S.A.) Planning and explaining with interacting expert systems

12.15 - 13.15 (lunch)
13.15 - 14.00 (free)
14.00 - 15.30 Paper session *Languages* (continued):

F. Arbab (U.S.A.) Examples of geometric reasoning in Oar
P. Bernus, P.J.W. ten Hagen, P.J. Veerkamp, V. Akman (The Netherlands) IDDL: The language of a family of intelligent, integrated, and interactive CAD systems (IIICAD)

15.30 - 16.00 (tea)
16.00 - 17.30 Discussion on *Languages* [Moderator: K.R. Apt]
17.30 - 19.00 (free)
19.00 - 20.30 (dinner)
20.30 - (free)

April 13 (Wednesday)

7.30 - 9.00 (breakfast)
9.00 - 10.30 Paper session *Interfaces*:

Zs. Ruttkay (Hungary) Multi-media presentation in CAD systems
J.R. Rossignac, P. Borrel, L.R. Nackman (U.S.A.) Interactive design with sequences of parametrized transformations

10.30 - 10.45 (coffee)
10.45 - 12.15 Paper session *Interfaces* (continued):

M. Beynon, A. Cartwright (England) A definitive programming approach to the implementation of CAD software
C. Tweed, A. Bijl (Scotland) MOLE: A reasonable logic for design?

12.15 - 13.15 (lunch)
13.15 - 14.00 (free)

14.00 - 15.30 Discussion on *Interfaces* [Moderator: G.R. Joubert]
15.30 - 16.00 (tea)
16.00 - 18.15 Paper session *Application Modules and System Architecture*:

V. Akman, P.J.W. ten Hagen (The Netherlands) The power of physical representations
F.-L. Krause, F.H. Vosgerau, N. Yaramanoglu (West Germany) Implementation of technical rules in a feature based modeller
D. Ben-Arieh (Israel) Product and process design in intelligent CAD workstation

18.15 - 19.00 (free)
19.00 - 20.30 (dinner)
20.30 - (free)

April 14 (Thursday)

7.30 - 9.00 (breakfast)
9.00 - 10.30 Paper session *Application Modules and System Architecture* (continued):

K. ElDahshan, J.P. Barthes (France) Implementing constraint propagation in mechanical CAD systems
K. MacCallum, S. Green (Scotland) THESYS — Implementation of a knowledge based design system with multiple viewpoints

10.30 - 10.45 (coffee)
10.45 - 12.15 Discussion on *Application Modules and System Architecture* [Moderator: P.J.W. ten Hagen]
12.15 - 13.15 (lunch)
13.15 - 14.00 (free)
14.00 - **(excursion** and **special dinner)**

April 15 (Friday)

7.30 - 9.00 (breakfast)
8.15 - 9.00 **(workshop programme committee meeting)**
9.00 - 10.30 Paper session *Design Process*:

M. Nadin, M. Novak (U.S.A.) Implementing intelligent processors
T. Takala (Finland) Design transactions and retrospective planning — Tools for conceptual design

10.30 - 10.45 (coffee)
10.45 - 12.15 *General Discussion* [Moderator: M. Mac an Airchinnigh]
12.15 - 13.15 (lunch)
PARTICIPANTS LEAVE .....

— Preliminary Announcement —

THIRD EUROGRAPHICS WORKSHOP ON INTELLIGENT CAD SYSTEMS
"Practical Experiences and Evaluation"

April 3 - 7, 1989
Hotel Opduin, Texel, The Netherlands

LOCATION We are pleased to announce that the *Third Eurographics Workshop on Intelligent CAD Systems* will take place in a unique location: Hotel Opduin on the island Texel (north of Holland). Hotel Opduin has superb facilities and promises to be an excellent conference site for this occasion.

SCOPE Applying knowledge engineering to CAD has become during the last decade a major area of research, known as *intelligent CAD*. The scope of this workshop includes (but is not limited to):

- Experiments with intelligent CAD systems.
- The role of intelligent CAD systems in industrial design.
- Acquisition and maintenance of design expertise.
- Software engineering for implementations of intelligent CAD systems.
- User interfaces for intelligent CAD systems.
- Knowledge representation languages for design.
- Integration of application software (finite elements, qualitative physics, etc.) to intelligent CAD systems.

The proceedings of the workshop will be published by Springer-Verlag in the *EurographicSeminar Books* series. The record of the first workshop has already been published by Springer-Verlag and the volume covering the second workshop should be out before December, 1988.

DEADLINES The following dates are provisional but will most probably stay unchanged:

• Nov. 1, 1988	Deadline for extended abstracts.
• Jan. 15, 1989	Notification of acceptance.
• April 3 - 7, 1989	Workshop (full papers due just before the workshop).
• May 15, 1989	Deadline for final manuscripts for Springer-Verlag.
• Dec. 1, 1989	Springer-Verlag third volume in bookstores.

ABSTRACTS We are planning to accept 20 papers and we shall limit the total number of participants to 50. Please submit 3 copies of a single-spaced extended abstract of **minimum** 1000 words (not counting figures and references) on A4 sheets before November 1, 1988 to the workshop secretary:

Ms. Marja Hegt, ICAD WS #3
Centre for Mathematics and Computer Science (CWI)
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
Tel. +31-20-592-4058, Fax +31-20-592-4199, Telex 12571 mactr nl, E-mail marja@cwi.nl

Submission by electronic mail is encouraged as long as the author sends the additional material (e.g. figures) in time. The abstract should include: title of the contribution, author's name, address (phone, fax, telex, e-mail information are very useful), main text and figures, and references. If you are interested only in participating (but *not* in presenting a paper) then you are still required to submit an abstract which should take the form of a *position paper* explaining how you regard the issues of intelligent CAD. *We are not planning to admit applicants who have not submitted an extended abstract (or as explained, a position paper).*

WORKSHOP FEE Approximately 1000 Dutch guilders. This price includes accommodation, food, and a special excursion. The workshop starts on April 3rd (Monday) with dinner and ends on April 7th (Friday) after lunch.

ORGANIZER This conference is sponsored by Eurographics and is being organized by Center for Mathematics and Computer Science (CWI). The cochairmen are: P.J.W. ten Hagen (CWI, The Netherlands), T. Tomiyama (Univ. of Tokyo, Japan), and P.J. Veerkamp (CWI, The Netherlands).

PROGRAMME COMMITTEE A. Agogino (Univ. of California - Berkeley, USA), V. Akman (Bilkent Univ., Turkey), F. Arbab (Univ. of Southern California, USA), P. Bernus (Hungarian Academy of Sciences, Hungary), A. Bijl (Univ. of Edinburgh, Scotland), J. Encarnacao (TH Darmstadt, West Germany), S. Fenves (Carnegie Mellon Univ., USA), D. Gossard (Massachusetts Institute of Technology, USA), F. Kimura (Univ. of Tokyo, Japan), T. Kjellberg (Royal Institute of Technology, Sweden), G. Kramer (Schlumberger Palo Alto Research Center, USA), M. Mac an Airchinnigh (Univ. of Dublin, Ireland), K. MacCallum (Univ. of Strathclyde, UK), S. Murthy (IBM Thomas J. Watson Research Center, USA), F. Schramel (Philips Eindhoven Research Center, The Netherlands), D. Sriram (Massachusetts Institute of Technology, USA), W. Strasser (Univ. of Tuebingen, West Germany), T. Takala (Technical Univ. of Helsinki, Finland), and F. Tolman (TNO, The Netherlands).

Paper Session:

Languages

Object Oriented Programming Paradigm for Intelligent CAD Systems

T. Tomiyama

Object Oriented Programming Paradigm for Intelligent CAD Systems

Tetsuo Tomiyama

Department of Precision Machinery Engineering
The University of Tokyo

Hongo 7-3-1, Bunkyo-ku, Tokyo 113, JAPAN
Telephone: +81-3-812-2111 ext. 6454
Telex: 272 2111 FEUT J, Fax: +81-3-812-8849
Internet: b39711%tansei.cc.u-tokyo.junet@relay.cs.net

Abstract: *The object oriented programming paradigm (OOPP) is expected to play an important role to implement intelligent CAD systems, because objects in the OOPP can be regarded as arbitrary objects in the world that the system is intended to model. It is so, as long as the message passing mechanism of object oriented languages is a good approximation of what happens in that world. We challenge this view by examining typical descriptions about physical objects. We see that some aspects of the OOPP are not necessarily appropriate to describe complicated design knowledge and to generate proper classes for the design objects. Finally, this paper is closed by suggesting some principles to avoid these difficulties.*

Key Words: *Intelligent CAD systems, Object oriented programming, Knowledge engineering, Smalltalk.*

1. Introduction

The concept of intelligent CAD systems is now widely discussed [Gero 1985, 1987; ten Hagen and Tomiyama 1987], although its definition is totally unclear. One of the most acceptable definitions is that an intelligent CAD system is what we originally intended when we started to work on CAD [Sutherland 1963]. What a CAD system should aim at can be categorized as follows.

- A place to describe what designers have in mind.
- A tool to verify the designer's ideas in terms of feasibility, cost and performance, etc.
- A system to store and retrieve design information and to transform it among various subsystems.

Conventional CAD systems were built for automating drafting processes and they satisfy these three points from a point of view of drafting. The only mistake in this story is a misunderstanding that design processes do consist only of drafting. This means that conventional CAD systems are equipped with knowledge for drafting but not for designing. However, as design objects become more and more complicated and sophisticated, designers require more help from computers and this resulted in the ideas of so-called intelligent CAD systems.

Advances of knowledge engineering ignited development of expert systems for CAD applications [Gero 1987]. However, most of them do not aim at satisfying the three points mentioned above and are intended to help unskilled designers by giving more problem solving abilities. At the same time, any knowledge based system should not be regarded as a magic tool which solves problems of current CAD systems. Even a knowledge-based system must be maintained much the same as conventional programs. In this sense, knowledge engineering is more than software engineering but not so much [Bobrow, Mittal, and Stefik 1986].

One seemingly promising approach to intelligent CAD systems is first to develop a general framework or environment to support future intelligent CAD systems [Tomiyama and Yoshikawa 1985; Tomiyama and ten Hagen 1987; Veth 1987] and then to build intelligent design subsystems on it. The framework serves as a sound, clean, yet robust basis for reorganization of design knowledge, including integration and bringing in flexible operations against them. Naturally, this approach does not consider the intellectualization of the system as the primary goal. Rather, designers are provided with various sorts of design knowledge in a more flexible, natural, and rapid way (probably than Fortran, C, or any other conventional languages). By *flexible* we mean, for example, a multi-purpose data modeling scheme. By *natural* we mean using the terminology of the target domain. For instance, we implement programs for linear algebra in Fortran using *arrays* and *do-loops*; the language concepts of APL, on the other hand, include *matrix* and various matrix operations. A language for intelligent CAD systems, in this way, should have language constructs that are naturally able to describe design processes and objects. Thus, an intelligent CAD system must be captured conceptually as a system on which more complex and non-numerical *design knowledge* should be implemented better than conventional CAD systems.

In this paper we examine the object oriented programming paradigm (OOPP) from a viewpoint of languages to implement intelligent CAD systems. Object oriented languages are becoming more and more popular and the ideas of the OOPP are considered indispensable for implementing applications like CAD, because the OOPP allows for usage of domain specific concepts in terms of objects. Chapter 2 analyzes the OOPP in general. In Chapter 3 we discuss CAD applications and the OOPP. First we consider a concrete example and try to describe it in an object oriented language. Then we see to which extent the OOPP is powerful and flexible in such applications by analyzing problems in that particular example, followed by some proposals to improve or avoid them.

2. Object Oriented Programming Paradigm

The OOPP [Stefik and Bobrow 1986; Cox 1986] has a number of features that make itself superior to and differentiate itself from other programming paradigms. But the most distinguished one from a point of view of users is that objects in an object oriented language are in fact role players. An object can possess both behaviors and properties of the entity that it is supposed to simulate. It behaves like that entity by sending and receiving messages based on our program. By doing this, objects become our private actors who perform scenarios written in the program. Thus, objects can capture the semantics of our world written in the program as long as message passing is a good approximation of what is happening in the real world. The OOPP liberates programmers

from thinking on the level of programming language concepts; we do not need to cast our ideas into a mold made of programming language jargons, and we just concentrate to describe our ideas in their own terminology. Therefore, it is reasonable to consider the OOPP as a promising vehicle toward intelligent CAD systems [Cholvy and Foisseau 1985; Veth 1987].

The following two concepts might be found widely in so-called object oriented languages, according to Stefik and Bobrow [Stefik and Bobrow 1986].

- *Message passing*
- *Specialization*

Message passing allows for *encapsulation* in terms of objects. An object understands a message and responds to it based on a *method*. A method must be defined in such a way that invoking the method should not require any additional knowledge on the implementation or inner structure of objects. This feature of *data abstraction* or *information hiding* results in another important concept of object oriented languages, *classes* and *instances*. The behavior of an object might be defined generally by a set of methods which as a whole defines a class. Objects in a class respond to a particular message in exactly the same way. In this case, these objects would be called *instances* of this particular class.

Specialization, on the other hand, makes it possible for an object to behave *like something*. This is usually done through *object hierarchy* which can implement the so-called *is-a hierarchy* in artificial intelligence (AI) most naturally, but there can be other techniques to do so (for instance, delegation; see the discussions in the next chapter). This kind of mechanisms allows *polymorphism* which is in fact another major issue. The object hierarchy allows for reuse of codes which in turn contributes to software engineering points of view of object oriented languages.

As discussed at the beginning of this chapter, the most essential concept of the OOPP is its programming style, i.e., abstraction in terms of objects based on message passing. (The reason to say this is that there are several ways to achieve data abstraction. For instance, Ada is one.) Now, a question resides within the appropriateness of object oriented languages as the base language for future CAD systems, because our experiences with the OOPP revealed some fundamental problems. For example, an interesting discussion can be found in [Arbab 1987] where Arbab questions the inheritance concept. He suggests that instead of inheritance the delegation concept [Lieberman 1986] should be used especially in CAD applications due to inflexibility of the inheritance mechanisms.

The present paper is an attempt to answer this question. In the next chapter, we try to describe problems of the OOPP in an illustrative way taking Smalltalk-80† [Goldberg and Robson 1983] as an example of an object oriented language.

† Smalltalk-80 is a trademark of Xerox Corp.

3. Problems of Object Oriented Languages

3.1. Is-A and Part-Of Hierarchies

As mentioned in the previous chapter, the *is-a hierarchy* is a method to implement specialization. Smalltalk-80 realizes the is-a hierarchy by using the inheritance mechanism through the class/instance hierarchy.

Most of object oriented languages have another important concept besides message passing and specialization. An object can have *inner structure*. For instance, Smalltalk-80 has concepts of *class variables* and *instance variables*. These are equivalent, to some extent, to *slots* in Misnky's frame theory [Minsky 1975]. In many cases, these variables are inherited through the class hierarchy. CommonLoops [Bobrow, Kahn, Kiczales, Masinter, Stefik, and Zdybel 1986], has a concept called *composite object*. These are ways to realize the *part-of structure* (or *hierarchy*) which is also an important issue in AI.

These two types of hierarchies, is-a and part-of hierarchies, are fundamental and essential to represent our knowledge in AI oriented information processing. In this chapter, we would like to depict problems of the mechanisms of Smalltalk-80 relevant to these two hierarchies. Smalltalk-80 was chosen because of the author's familiarity with it. However, the mechanism of inner structure of objects can be found in most of object oriented languages, including Smalltalk-80, Flavors, C++, etc. Therefore, the nature of the problems are common to other object oriented languages more or less.

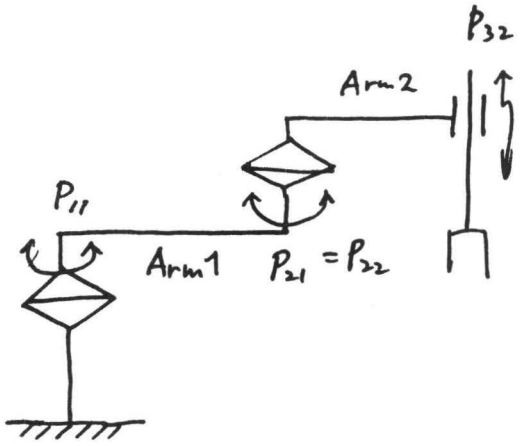


Fig. 1. A Robot

3.2. An Example

Let us consider the design of a robot (Fig. 1). From a view point of mechanical engineers, a robot can be regarded as connected rigid bodies for the purpose of kinematic evaluation. Thus, we can compute the position of the hand using mathematical concepts such as position vector, coordinate transformation matrix, etc. (In fact this is a typical exercise for engineering students.)

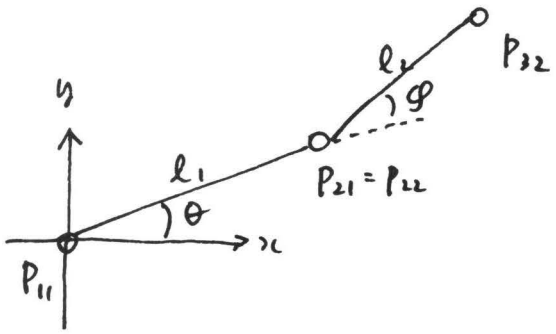


Fig. 2. Coordinates

Now suppose we program this problem in Smalltalk-80. It is reasonably natural to say;

- aRobot is an instance of the class Robot.
- a robot has 6 arms, arm1 to arm6, as instance variables.
- arm1 is an instance of the class Arm.
- an arm has instance variables, endPoint1, endPoint2, length, transformationMatrix, etc.
- endPoint1 is aPoint which has x-, y-, z-coordinate as instance variables.

We can, of course, define instance methods to compute the position of arm6 by sending a message "computePosition" to arm6. This computation may create a message "computePosition" sent to arm5, and this in turn may create further message-sendings.

class Arm Instance Variables	arm1	arm2
endPoint1	p_{11}	p_{22}
endPoint2	p_{21}	p_{32}
length	l_1	l_2
relativeAngle	θ	φ

Fig. 3. Descriptions in Smalltalk-80

For the sake of simplicity, let us consider a simple industrial robot with two rotating arms and one sliding arm shown in Fig. 1. Figure 2 shows its mathematical modeling. The most natural implementation of this problem in Smalltalk-80 would be to define a class "Arm" with four instance variables: endPoint1, endPoint2, length, and relativeAngle. Thus, we can give descriptions for the two arms of this robot as shown in Fig. 3.

Suppose we need to control the arms, given the position of the hand, P_{32}. (This means that we need to compute θ and φ.) Here arise many problems in this computation. Since the configuration of this robot allows to build an equation as in Fig. 4, it is solved quite easily by introducing a couple of auxiliary variables. To implement this, we have

to somehow implement dependency relationships among variables as in Fig. 5.

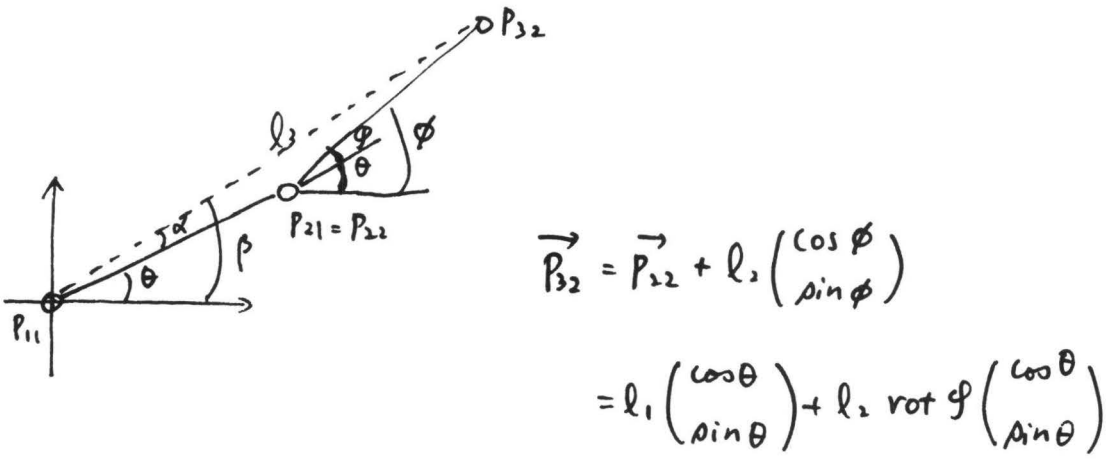


Fig. 4. Mathematical Equation

3.3. Problems

By observing the dependency network in Fig. 5, we realize problems in this implementation.

- How do we define the fact that endPoint1 of arm2 is the same as endPoint2 of arm1?

Smalltalk-80 does not allow sharing instance variables among two objects because of the principle of information hiding (see Fig. 6 (a)). Another way to express it is to say that, for example, in the description of endPoint1 it is identical to endPoint2 of arm1. Unfortunately we are not able to say it, either, because the object arm2 is not allowed to know the structure of other objects due to information hiding (see Fig. 6 (b)). Thus, we need to create an independent object, point2, and the instance variable, e.g. endPoint1 of arm2, must be a pointer to it (see Fig. 6 (c)). Here the problem is that this information is static rather than dynamic and it is not exactly well abstracted in terms of message passing.

- How do we represent the fact that the joint between arm1 and arm2 is a rotating joint?

To do so, we have to define an imaginary object "joint1-2", for instance, and we say arm1 is connected with arm2 via joint1-2 of which type (or class) is a rotating joint. This joint1-2 description can replace the point2 object discussed above. A problem arises here: Is this "joint1-2" really a mechanical "object"? From a point of view of simulating this robot, perhaps it does not matter whether or not joint1-2 is a real mechanical entity. But from a point of view of programming, it is obvious these that imaginary objects make programs unclear, because they are not based on real entities. The following question asks the same issue.

- What are, then, those auxiliary variables such as α and l_3?

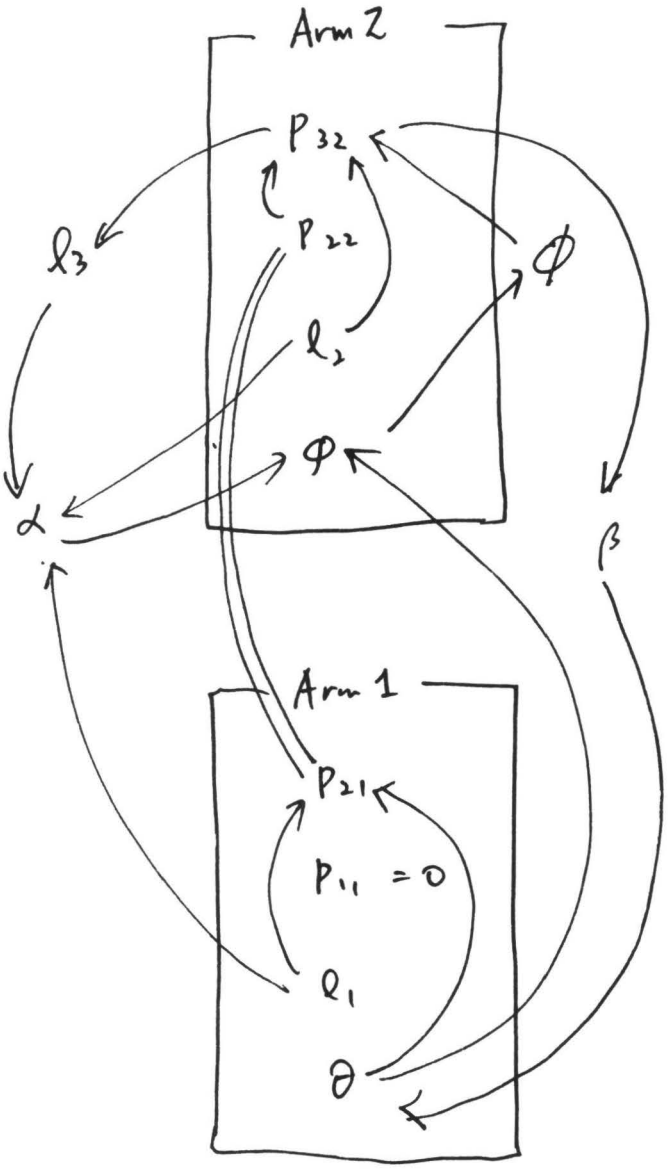


Fig. 5. Dependency among Variables

In order to know l_3, we need to know P_{32}. Are we allowed to refer to P_{32} directly from the outside of arm1? This is not allowed in Smalltalk-80 and we must implement a method such as "endPoint" in the class Arm. If so, in order to refer to endpoints even from the inside of arms, does this method have to be called? This isn't the case in Smalltalk-80 and we can freely access to instance or class variables within the class. However, if the instance variable is defined in one of its superclasses, doesn't this mean violation of the principle of information hiding? Practically speaking, instance/class

variables and methods defined in superclasses are more or less foreign to subclass objects. In particular, if the distance between the object's class and its superclass is big, that superclass is not familiar to the programmer and he might regard it as a foreign class. This problem happens only in inheritance mechanisms but not in delegation mechanisms.

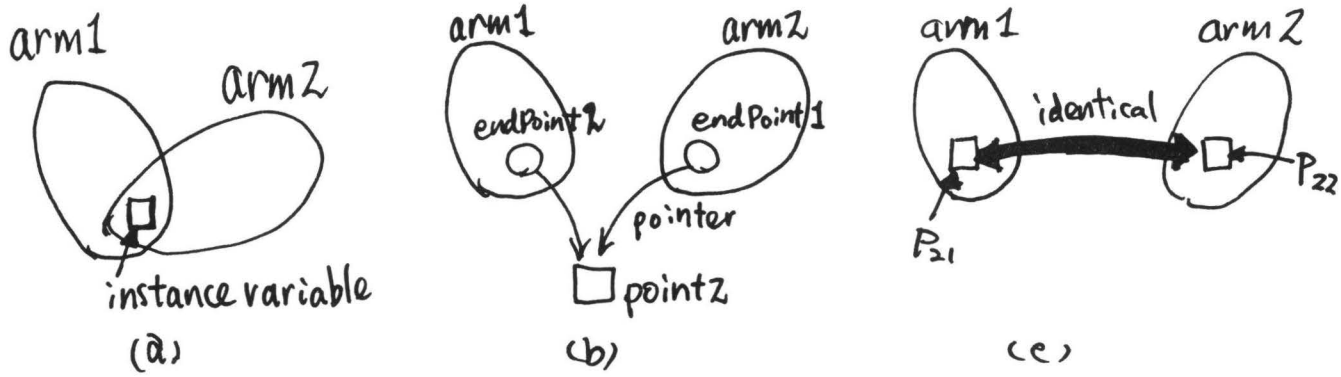


Fig. 6. Representation of Static Relationships

3.4. Analysis of the Problems

The discussions in the previous section can be summerized as follows.

1. Static relationships cannot be expressed well using message passing.

As discussed in Chapter 1, the OOPP can provide a good simulation environment as long as the message passing paradigm is a good approximation of what happens. Since message passing itself is somehow a dynamic event, static relationships are difficult to describe.

2. In order to represent static relationships, we have to create *imaginary* objects.

The drawback here is that these imaginary objects make programs unclear.

3. The concept of class/instance variables is a problem when there are inter- and intra-structural relationships, because this contradicts with the principle of information hiding.

The class/instance variables (i.e. inner structure) of objects are used mainly to represent the part-of hierarchy. However, if it is possible at all to represent this hierarchy by other means, we better avoid to use.

4. The inheritance of class/instance methods and variables does not fit into the principle of information hiding, because sometimes superclasses are completely foreign for subclasses.

Besides the problems having inner structure, the inheritance mechanism also disturbs the principle of information hiding.

3.5. Proposals

As pointed out in Chapter 1, one of the main goals of having a framework in an intelligent CAD environment is to obtain flexibility to *implement design knowledge*. In this context, object oriented languages like Smalltalk-80 have problems in that they have an inheritance mechanism and inner structures in objects, because they violate e.g. the principle of information hiding.

Instead of the inheritance mechanism, we can use the delegation mechanism; a sort of non-automatic inheritance mechanism. Unless clearly declared, an object does delegate only itself. If it must be treated as another type of object, we must clearly declare.

Inner structures of objects should be avoided as well. One alternative is to have "plain objects" which have no internal structures and some additional mechanisms to represent the part-of and is-a hierarchies. For example, we can employ logical expressions only to represent static relationships among objects. If we have to use instance variables for one reason or another (e.g. to represent attributes of objects), instead we may use functions defined over objects [Veth 1987]. The is-a hierarchy which must be realized by the delegation mechanism can be represented by a first-order predicate clause. Suppose x is an object of class c. If this object belongs also to other class cc, it can be expressed by a Prolog-like clause:

$$c(x) := cc(x).$$

After the moment this clause is asserted, x delegates not only c but also cc and := should be treated as a special symbol which conveys the essential meaning of the is-a hierarchy.

The part-of hierarchy can be treated by predicate logic in the same way. For example, we can treat the *partOf* predicate as a special one which expresses the same meaning as instance/class variables.

4. Conclusion

Intelligent CAD systems need powerful (as in the logic programming paradigm), natural (as in the object oriented programming paradigm), and yet flexible basis. As a conclusion of this position paper, we would like to point out that the OOPP is yet to be revised. Among other things, the concept of having inner structure and inheritance must be re-examined. These two concepts should not be treated within the framework of the OOPP but outside it. For instance, we might be able to introduce the logic programming paradigm so that we could represent the is-a hierarchy. The IDDL language developed by a group at Centre for Mathematics and Computer Science, Amsterdam, solves these problems by coupling the logic and object oriented programming paradigms [Veth 1987].

This research was mainly done while the author was at Group Bart Veth of the Centre for Mathematics and Computer Science, Amsterdam. The author would like to thank the members of Group Bart Veth, especially Paul J.W. ten Hagen and Varol Akman, for their valuable advises and help.

References

[Arbab 1987]
Arbab, F.: "A paradigm for intelligent CAD," in [ten Hagen and Tomiyama 1987], pp. 20-39.

[Cholvy and Foisseau 1985]
Cholvy, L. and Foisseau, J.: "Encoding requirements to increase modelisation assistance," in [Gero 1985], pp. 205-221.

[Bobrow, Kahn, Kiczales, Masinter, Stefik, and Zdybel 1986]
Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F.: "CommonLoops: Merging Lisp and object-oriented programming," in Meyrowitz, N. (ed.), *Proceedings of OOPSLA '86,* special issue of *SIGPLAN NOTICES,* **21**(11), pp. 17-29.

[Bobrow, Mittal, and Stefik 1986]
Bobrow, D.G., Mittal, S., and Stefik, M.: "Expert systems: Perils and promise," *Communications of the ACM,* **29**(9), pp. 880-894.

[Cox 1986]
Cox, B.J.: *Object Oriented Programming: An Evolutionary Approach,* Addison-Wesley, Reading, MA, USA.

[Gero 1985]
Gero, J.S. (ed.): *Knowledge Engineering in Computer Aided Design,* North-Holland, Amsterdam.

[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 1: Theoretical and Methodological Aspects,* Springer-Verlag, Berlin, Heidelberg, New York, Tokyo.

[Lieberman 1986]
Lieberman, H.: "Using prototypical objects to implement shared behavior in object oriented systems," *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications,* pp. 214-223.

[Minsky 1975]
Minsky, M.: "A framework for representing knowledge," in Winston, P.H. (ed.), *The Psychology of Computer Vision,* McGraw-Hill, New York, pp. 211-277.

[Stefik and Bobrow 1986]
Stefik, M. and Bobrow, D.G.: "Object-oriented programming: Themes and variations," *AI Magazine,* **6**(4), pp. 40-62

[Sutherland 1963]
Sutherland, I.E.: "SKETCHPAD - A man-machine graphical communication

system,'' *Proceedings of Spring Joint Computer Conference,* pp. 329.

[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 Yoshikawa 1985]
Tomiyama, T. and Yoshikawa, H.: ''Requirements and principles for intelligent CAD systems,'' in [Gero 1985], pp. 1-28.

[Veth 1987]
Veth, B.: ''An integrated data description language for coding design knowledge,'' in [ten Hagen and Tomiyama 1987], pp. 295-313.

Planning and Explaining with Interacting Expert Systems

J.F. Koegel

Planning and Explaining with Interacting Expert Systems

J. F. Koegel
Dept. of Mathematics and Computer Science
University of Denver
Denver, Co 80208
USA

Institutes for Information Processing Graz (IIG)
Technical University of Graz and
Austrian Computer Society
A-8010 Graz
Austria

Abstract

We distinguish planning as a specific and generalizable component of the design activity. Subtasks of design planning are identified and mapped to an architecture of a planning program. This architecture extends current planning technology by combining distributed problem solving, domain-expert problem solvers, interactive planning, and plan re-use. A mechanism for explanation of design plans and design decisions is presented in the context of the proposed architecture. Theoretical limits of conventional planning strategies are discussed.

1. Introduction

Planning is an important part of the design activity. Complexity of function requires that designs be decomposed. Reasoning about alternate decompositions, the order in which components should be analyzed, and interactions between component designs are concerns of planning.

Although expert systems have been developed to perform design in specific domains, such systems are difficult to adapt to other domains. The planning process itself is implicit in the rules and specialized to the domain. By separating the planning process from other aspects of the design process, important steps in the design activity can be generalized. Although some generality might be attained in the expert system approach by adding planning-related metarules, greater generality follows from using existing domain-independent planning techniques.

The benefits of this approach for intelligent CAD (ICAD) systems are several. First, it leads to the development of a general-purpose planning component which can be used in many different domains. Second, it makes the planning process explicit so that it can be reasoned about and explained. Third, it facilitates simultaneous exploration of design alternatives. As the designer may wish to switch between alternatives, to propose his own, and to link separately developed plans together, the ability to maintain multiple foci is important. Fourth, by being domain-independent, it provides facilities for reasoning about and supporting the design process when the problem falls outside of the scope of the domain-specific problem solvers.

This paper presents some important results for implementation of the model for ICAD presented in [Koegel 1987a]. These results include:

1. A characterization of engineering design that permits the planning tasks to be identified and generalized

2. A view of these tasks from the perspective of the GDP model presented in [Koegel 1987a]

3. A mapping of the GDP model to a planner architecture which combines distributed problem solving, interactive planning, domain expertise, and plan re-use

4. New proposals for domain representation and constraint management for planner programs for design planning

5. A mechanism for explanation of plans and decisions in a distributed environment consisting of deductive agents which interact via message passing

The next section describes different theories of design that have been proposed. These provide the backdrop for Section 3, The GDP Model. Section 4, Architecture of a Design Planner, discusses the design of a planning program which implements the GDP model. The following section addresses interactive planning and explanation of plans in the context of the proposed architecture.

Three more sections briefly discuss theoretical limitations, implementation results, and related work.

2. Models of the Design Process

2.1 Definitions

Design: The development of a detailed specification so that the desired object or process conforms to constraints relating to both tangible and intangible properties such as function, materials, cost, physical dimensions, aesthetics, etc. A design is complete if the specification can be used to manufacture, construct, or realize the design. A design is correct if the object or process produced from the specification conforms to the original constraints for the object or process.

Planning: The act of determining a set of actions and their (partial) ordering to achieve a given task.

Plan: The (partial) order of actions produced by planning.

Design Plan: Although in some domains a "plan" can be synonymous with the design itself, we mean the sequence of decisions to be made that when performed will produce an acceptable design.

2.2 A Survey of Domain Specific Models

As previously stated, planning is an important part of the design activity. Identifying its features requires an understanding of the design process. However, it is generally agreed that the design process is incompletely understood. The following descriptions of the design process taken from different domains elucidate common features of design. This provides a stepping stone for discussing the planning component of design.

Sussman and Steele [Sussman 1980] have discussed a model of design in the context of electronic circuit design called problem solving by debugging almost-right plans (PSBDARP). In this model the designer begins by forming a high-level description of the design. This is divided

into subparts which can be dealt with somewhat independently. As the design proceeds, each subpart may be further partitioned until primitive elements are reached. The identification of the primitive elements and the specification of their parameters is dealt with iteratively in order to produce a consistent set of parameter values which satisfy the design constraints.

The premise of this model is that expert designers know the form of the answer when they begin a design. The design then involves manipulation of these forms until the problem constraints are specified. The constraints model the functional behavior of the set of elements currently under consideration. This locality is used to reduce the complexity of the design. Since a given set of constraints may not account for certain global effects of interest, different views can be attached to the design. The views allow alternate sets of equivalent constraints to be used as needed.

A model for mechanical design has been developed by Brown and Chandrasekaran [Brown 1983, Brown 1985, Brown 1986]. Design can be broken into three classes, which correspond roughly to significant invention, innovation, and routine design. Their model deals with the third category, routine design. In this case, the design process and the components of the design are well understood.

The design problem solver is structured as a hierarchical collection of specialists. Each specialist is dedicated to a particular sub-problem of the design and has knowledge only for that sub-problem. The organization of the specialists is isomorphic to the problem decomposition. Thus specialists at the top of the hierarchy deal with more general aspects of the design, while those near the bottom deal with more detailed aspects.

Each specialist has one or more pre-defined plans that it can choose from to solve its particular problem. A plan is a predetermined sequence of steps that are known to be applicable to the problem. Because this is routine design, all possible plans are known ahead of time. The specialist selects a plan and attempts to carry it out. This may invoke other specialists lower in the hierarchy. When a specialist successfully completes its part of the design, the results are returned to the next level. This continues until the design is complete or until a failure which can't be resolved occurs.

A multiple expert system design paradigm has been proposed for design automation for VLSI in [Brewer 1986]. A design is developed in stages of decreasing abstraction. A separate expert system provides the design expertise for each level of abstraction that the design passes through. Each expert system performs five tasks related to the design: constraint propagation, planning, refinement, optimization, and evaluation.

Constraints are passed top-down from level to level based on the initial design goals; constraints are also derived from physical limitations at the lowest level of the design. In this domain, planning refers to selection of a particular design strategy; selection is based on choosing a design style most likely to achieve the constraints. The strategy follows from the selected style. If a design fails at one level, the system backtracks to the previous level. Each level optimizes its design before passing it to the next level; this provides local but not global optimization.

In comparison with other domains, levels of abstraction are more delineated in VLSI. This simplifies the partitioning of the expertise and the interaction between levels.

2.3 Summary

Common features of each of the preceding models include:

1) Hierarchical problem decomposition with greater abstraction at the top of the hierarchy

2) Use of the "nearly decomposible into independent subtasks" assumption to simplify the problem

3) Use of design constraints to guide the design process

4) Synthesis of the design accomplished by rote

This characterization does not constitute a comprehensive model of design. Aspects of design that are omitted include qualitative design (e.g., aesthetics), synthesis, and innovation.

The planning tasks that can be identified for this restricted view of design include:

1) Development of the problem decomposition in a hierarchical fashion

2) Selection of a problem decomposition from a number of alternatives

3) Responding to failure in a given component design due to a constraint violation

4) Coordinating the resolution of interacting component designs

5) Ordering the application of constraints

3. The GDP Model

3.1 GDP and Design

In the ICAD model presented in [Koegel 1987a] we described a model for design planning called Goal Driven Problem Solving By Debugging Almost Right Plans (GDP). This model combines two complementary techniques of design planning: a goal directed approach for making the design process explicit and PSBDARP for performing design when substantial expertise exists.

The GDP model supports the planning tasks identified above for the restricted view of design. The plan-debugging approach fits the design characteristics listed above such as the use of hierarchical decomposition, the assumption of near independence of subtasks, and so forth, which were extracted from three different domains. The goal-driven approach ensures that the planning tasks of this more mundane type of design can be isolated for explanation and analysis.

The goal-driven approach provides that any design goal that can be represented can be included in the design process. Thus aspects of design which are currently less understood (synthesis, qualitative design, innovation) can be incorporated to the degree that this current understanding permits.

3.2 GDP and Conventional Planning Technology

Planning techniques most relevant to implementing GDP use a hierarchical and non-linear

representation for plans. Hierarchical refers to the ability to plan at different levels of abstraction; non-linear refers to the fact that plan steps are not required to be strictly ordered. The first planner to provide these capabilities was Sacerdoti's NOAH [Sacerdoti 1977]. Wilkins [Wilkins 1985] subsequently improved NOAH by providing mechanisms for constraint-posting, reasoning about resources, and interactive planning in a program called SIPE.

Neither SIPE nor NOAH support plan re-use. The first planner to demonstrate this capability was a linear, non-hierarchical program called HACKER [Sussman 1975]. HACKER produced plans by trying to debug plans for similar problems which were previously stored in its plan library. It used pattern matching to search for candidate plans. Proposed plans would be evaluated by a critic which would look for known bugs. These would be corrected and then the plan would be attempted. If a failure occurred, there would be further debugging until a successful plan was produced. HACKER could selectively update the plan library when new plans were encountered.

Since the planning problem has frequently been cast in the domain of robot manipulation, these and most other planners use a situation-action representation for the domain for which the plans are constructed. Specifically, the planning problem is stated as follows: There is an initial state and a goal state. Performing an action in the current state produces a new state. The planner selects and orders a sequence of actions which transforms the initial state to the goal state.

The situation-action representation is appropriate for domains where the attributes of objects being acted on can be different from one state to the next. The design process already described accumulates constraints on the attributes of the objects which are associated with the form of the design. These constraints must be consistent in order for the design to be realizable. While it is possible to distinguish each newly accumulated constraint as producing a new design "state", the objects in these "states" do not change state. Rather, their "state" becomes more constrained as the final design is approached.

Conflicting interactions between design components could be viewed as two components whose mutual constraints put them in different "states". Then repair of the conflict could be performed by changing one of the "states" or inserting a new intermediate "state" as a bridge. However, this approach seems artificial and might be better handled by domain-dependent debugging techniques.

Since we view the design process as producing a sequence of constraints which specify the design object in detail, the situation-action model is inappropriate. Instead of actions which add some effects and delete other effects in the current "world", we have plan steps which add a new constraint to the design. Consequently we avoid the frame problem, that is, the problem of deciding which things change and which do not in a state transition.

Since design can also involve time-dependent processes or kinematics, it might be argued that designs involving dynamics require a situation-action representation for planning. However, tools such as systems analysis and simulation which are currently used to deal with dynamics are not likely to be displaced by a situation-action planning representation for practical reasons.

Most domain-independent planners have been demonstrated for relatively simple domains. The planners mentioned so far deal with planning in the blocks world, although SIPE also has generated plans for cooking, travel, and air plane scheduling. MOLGEN [Stefik 1981] solves problems in molecular genetics but is a domain-specific planner. Planners to efficiently support the complex problem solving needed for CAD have yet to be demonstrated.

In order to extend such planners to more complex domains, modification must be made to those

parts of the planners in which such domain knowledge is imbedded. In hierarchical planners such as NOAH and SIPE, this knowledge is found in the plan operators and critics. In HACKER this knowledge is distributed across a number of components, including the plan library, bug classifier, and the plan critic. It is desirable to provide such extensions modularly so that the expertise can be used by other parts of the system or by the designer directly. Our approach is to provide a uniform interface to domain specific components such as operators and critics. We expect that by implementing these components as objects in a language such as POOL [Koegel 1987b] we will obtain both a modular design and a performance improvement.

3.3 Planning Procedure and Planning Strategy

The planning procedure can be isolated from the planning strategy. The planning procedure is simply three steps: 1) decompose, 2) solve each sub-plan independently, and 3) resolve interactions. Since there usually is more than one way of performing each of these steps, some strategy might be used to select the preferred choice. This separation is frequently referred to as planning and meta-planning.

This distinction can be clarified by comparing Figure 1 and Figure 2. Figure 1 shows an abstract procedure for planning which performs the three steps previously identified. Figure 2 shows an abstract meta-planning procedure which uses strategies to determine rankings for alternate ways of decomposing and composing plans.

```
Plan (Goal)
    Let D be the set of all possible decompositions of Goal

    While D not empty
        Remove next Di from D
        Let Pi be the set of all possible combinations of
          plans of the components of Di

        While Pi not empty
            Remove next Pi,j from Pi,
              a set of designs for the components of the
              decomposition Di
            Let P'i,j be the set of all possible plans
              derived from Pi,j, but with interactions resolved

            While P'i,j not empty
                Remove next Answer from P'i,j
                Return Answer
            End While

        End While

    End While

    Return Failure
```

Figure 1 An Abstract Design Planning Procedure

The advantage of making this separation is that it makes the planning process explicit for explanation. It can also contribute to the modularity of the implementation. Also, although the planning procedure doesn't require situation-action representation, a planner used to implement the meta-planner may.

```
Meta-Plan(Type,S)
    Select Type

        case Decomposition
          Use strategy S1 to order set of
          decompositions S in decreasing preference
          to produce S'

        case Combinations-of-Plans
          Use strategy S2 to order set of
          combinations of plans in decreasing preference
          to produce S'

        case Combinations-of-Plans-with-Interactions-Resolved
          Use strategy S3 to order set of
          combinations of plans with interactions resolved
          in decreasing preference to produce S'

        Return S'
```

Figure 2 An Abstract Design Meta-Planning Procedure

4. Architecture of a Design Planner

4.1 Key Features

The design planner architecture presented here, which we will refer to as *DP-1*, combines features found individually in a number of previous planners with a distributed problem-solver implementation. These features are:

1. A hierarchical goal network, similar to the procedural network of NOAH and SIPE, but each goal is an active agent rather than a data node.

2. A control algorithm which allows the designer to direct and override the planning procedure.

3. Plan operators and critics which contain the domain expertise. Each can be a few rules or a complex expert system. Each operator and critic is implemented modularly so that the "expert" can be used both by the planner and the designer. Each operator and critic is an independent agent which communicates with the goal network and other operators and critics using message passing.

4. Explanation of the planning activity using the goal network, plan operators and critics. Special techniques for explanation in a distributed problem solver.

5. A plan library for saving interesting new plans and retrieving relevant old plans. Plan re-use techniques modeled after those of HACKER.

6. Avoidance of a situation-action domain representation.

7. A hierarchical constraint network that follows from the goal network. Constraint management that differs from the constraint-posting approach found in SIPE.

4.2 Goal Network

NOAH and SIPE represent the plan hierarchy using a procedural network. The procedural network is a hierarchy of plans where each successive level is a more detailed version of the plan of the previous level. Each plan is a directed graph connecting nodes which represent an action to be performed. Each node contains a list of effects, connections to other nodes, and a list of procedures which when invoked generate a more detailed plan for that node.

The goal network used in *DP-1* is similar in structure and content to the procedural network. However the nodes in the procedural network are replaced by agents which communicate with plan operators and critics. This supports parallel expansion of plan steps and easy access to the goal network by all agents in the system.

4.3 Operators and Experts

The domain knowledge needed to generate plan steps is found in the plan operators of conventional planners. Each operator obtains the goal for a given node and produces a detailed sub-plan if the operator is applicable. This sub-plan may contain parallel branches or iterative loops. Both NOAH and SIPE provide special language constructs which operators use to build the new nodes for the sub-plan. *DP-1* uses constructs derived from its implementation language in order to retain uniformity in the implementation.

Operators for complex design are likely to require significant domain expertise. This requires adequate representation power of the implementation language. The language used to implement *DP-1* combines distributed problem solving with a deductive facility which has been used successfully for building expert systems. Thus each operator in *DP-1* provides the necessary expertise either directly or through interaction with other agents. If the message protocols for each operator are provided to the user interface, then the designer can potentially directly access this expertise.

The *DP-1* architecture does not specify how expertise is to be distributed across the various agents. There could be one set of agents which provided the domain expertise and another set which mediate this expertise to the planner. There could be one agent for each type of design problem, or one agent for each level of abstraction of the design. The design of a collection of agents for a given domain is important research question. The architecture simply provides for a collection of agents to create plans and interact.

Pre-defined plans for routine design [Brown 1983] are an important subset of design planning. Operators can retrieve such plans from the plan library.

4.4 Critics

In non-linear planners which use the situation-action domain representation, critics are used to fix conflicts between different plan steps or to eliminate redundant operations. In HACKER, a critic was used to fix bugs in plans which had been proposed as possible solutions but which were not exact matches. Since *DP-1* doesn't use the situation-action representation, NOAH-style critics are not needed. HACKER-style criticism can be used in operators which re-use plans.

DP-1 requires domain-specific critics to resolve conflicting constraints between components of a design. These critics would be invoked when a constraint inconsistency between two component designs was detected at given plan level. Critics can be categorized by the type of recovery technique employed:

1. Insertion of an intermediate component or stage to act as a bridge
2. Modification of either or both of the conflicting component designs
3. Use of a different combination of designs
4. Use of a different decomposition of the higher-level goal from which these components are descended

4.5 Constraint Management

Hierarchical decomposition allows the designer to localize his problem solving in order to simplify the design problem. Constraints are added incrementally and evaluated independently for each design component. As the constraints for each component are satisfied, the next level higher checks for consistency between the components. This process continues until the top-level constraints are satisfied.

Each level of the plan can contribute constraints to the problem. Each goal at a given level can contribute one or more constraints. These should be both locally and globally consistent. Local consistency can be evaluated first at the goal level (if a goal has more than one constraint) and then at the given plan level. Global consistency is checked at higher levels in the plan.

In the implementation language used for *DP-1*, constraints are executable programs. Both algebraic (linear, non-linear, inequalities) and predicate logic constraints can be represented, all in declarative form. Each goal in the goal network has an attached procedure which defines the constraints defined for this goal. The local consistency of these constraints can be checked by executing the procedure. Global consistency is checked by starting at the top level and executing the procedures at each successive level.

This approach differs from the constraint posting mechanism first implemented in MOLGEN and generalized in SIPE. First, the constraint language used in *DP-1* is a general-purpose programming language which is available for all agents in the system. Second, different constraints can be considered for the same objects simultaneously since disjunctive branches can occur in the plan (to represent alternatives) and constraints are stored locally rather than posted to a global object hierarchy.

4.6 Meta-Planning

As discussed in Section 3.3, meta-planning is needed whenever a design decision is made. Unlike the knowledge needed for fixing plan interactions, we expect that meta-planning knowledge has a substantial domain-independent component. These could be rules such as: "Always choose the simplest decomposition" or "Choose the decomposition with which the planner has the most experience". This general meta-planning knowledge would be developed through experience with planning in a number of domains, and would be used when domain-specific knowledge was exhausted.

Although once again there are different ways of distributing meta-planning knowledge across a collection of agents, each node in the goal network would know the identity of the agent (or agents) which provide the expertise needed for its expansion. Then when a decision is needed, the appropriate agent(s) can be invoked by message passing.

4.7 Plan Re-Use

The simplest kind of re-use occurs if a given goal has been previously solved by the planner. This kind of facility is straight-forward to supply. The more general and difficult situation is that a plan for a similar goal exists. In this case the planner must decide between debugging the plan for the similar goal or generating an entirely new plan. The degree of similarity between the two goals is a partial metric in determining how difficult the plan debugging might be.

Criteria for saving plans in the plan library are needed. HACKER would save a plan if the same goal pattern was used more than once. Sacerdoti [Sacerdoti 1977] suggested that only highly criticized plans needed to be saved; others could always be re-generated.

5. User Interface

5.1 Interactive Planning

The reasons for supporting interactive planning have been previously discussed [Koegel 1987a]. Possible interactions include:

1. The designer proposes a plan for all or part of a design

2. The designer wishes to interact with several different alternative plans simultaneously, perhaps by viewing them in separate windows

The planner is able to maintain multiple alternatives by representing disjunctive branches at a given node in a plan. Each branch can then be expanded in parallel. Plans which the designer proposes are handled the same way as plans proposed by operators. This is a consequence of having a common interface and modular implementation.

5.2 Explanation of Design Decisions

In this section we discuss how different types of explanations can be supported by the architecture presented in section 4, and present a novel approach to constructing explanations in a distributed expert system environment where the expert systems interact by passing messages.

Various types of explanation are needed to make the planning component useful to a designer. These include:

1. How was a plan produced?

2. Why was a value needed?

3. Why wasn't alternative X used?

4. How was an interaction between two sub-plans resolved?

5. Why didn't a plan (or sub-plan) work?

Robust explanation is supported in two ways: first, each of the planning decisions, including failures, is explicit in the goal network. Questions about plans and sub-plans can be determined directly from this hierarchy. How was a plan produced? The system traces out the decompositions and constraints. How was an interaction resolved? The system locates the record of the interaction and its resolution in the goal network and uses the record and the behavior of the critic to construct an explanation.

Second, all agents are implemented as deductive procedures allowing standard techniques of inferentially generated explanation to be used. Thus "how" questions are answered by tracing down the proof tree of a given result, and "why" questions are answered by tracing up the proof tree.

When a result is produced by the interaction of a collection of distributed problem solvers, derivation of the proof tree requires special techniques. We assume that each agent is implemented deductively and communicates with other agents via asynchronous message passing. The basic idea is that the proof tree is accumulated and passed with each message.

The system starts with a top-level goal which results in a message to some agent to solve the goal. The agent performs deductions and may transmit messages to other agents (or recursively). The current sequence of deductions made so far constitutes an inference chain. Whenever an agent sends a message to another agent, this inference chain is also transmitted. The agent which receives the message makes further deductions which are added to the inference chain received with the message. It too may send other messages. Each message sent includes the inference chain accumulated at the point the message was transmitted.

Finally, an answer message is sent to the user interface. This message includes the entire proof tree starting from the message which contained the initial top-level goal. The user interface can manipulate this tree to produce an acceptable explanation. Both "why" and "how" questions can be answered using this tree. "Why" questions can also be answered at any point along an inference chain.

Agents can also have local state which is used in computing answers. The reason why a given variable has a particular value is important for explaining a decision which used that value. Variables are annotated with the reasons for their values. These reasons are integrated in the proof tree for decisions which use the variables. The initial value of a variable is a given. When the value for a variable is changed, it is because the agent received a message which caused it to perform some deductions and change the value of the variable. In this case the inference chain that led to the change can be used as the annotation. If a variable is a list, it may be desirable to have annotations for each element in the list.

Proof trees can be large and may contain much unnecessary detail. Sometimes the deductive sequence does not represent the clearest explanation of the reasoning involved. In these cases, it is desirable to prune and modify the tree that would normally be passed to the next agent. The function that generates the proof tree at each agent can be directed to produce the pruned or modified tree as desired. The tree that reaches the user interface can be further pruned or modified using more global knowledge.

6. Planner Correctness and Completeness

Recently Chapman [Chapman 1987] has investigated limitations of planners which make the STRIPS-assumption--i.e., planners which use the situation-action representation and assume that an action changes only those objects which are listed in its effects and no others. He presents theorems which suggest that general-purpose planners which use a truth criteria which allows the plan generation to be proven correct and complete can not be made efficient. While these results are not directly applicable because of the avoidance of the situation-action representation, they show formally the trade-off between representational power and correctness and completeness of the planning algorithm. As discussed by Chapman, the possible remedies for this limitation are to avoid domain-independent planners, live with the limitations, or augment domain-independent planners with various problem-solving techniques that are neither fully generaly nor domain specific. Since *DP-1* is not an autonomous planner, there is also the possibility of relying on the designer.

Any plan generated by *DP-1* is correct if the design constraints are consistent, which is easily checked by executing the constraints. (While the implemented constraint solver solves only linear equalities, any non-linear equations could be replaced with a piece-wise linear approximation to make this check.) Plan generation could also be made complete, but this could lead to exponential search. However, completeness is less crucial than correctness since the designer is in the loop and could potentially provide designs that the planner might miss. Thus in the absence of suitable expertise or options, the planner would do well to interact with the designer to resolve problems. Any solutions generated by the designer in these situations might be acquired by the planner for future use.

7. Implementation

POOL [Koegel 1987b], a language to implement deductive agents which communicate by asynchronous message passing, is implemented on a network of workstations. Agents are mapped transparently and automatically to different machines.

A constraint logic programming extension to Prolog following the CLP(R) system [Jaffar 1987] has been implemented and integrated with POOL. This provides the constraint-based reasoning facility for *DP-1*. CLP(R) solves linear equations and inequalities, and has a delay mechanism for non-linear equations.

A prototype of *DP-1* is near completion; currently the prototype can generate a 3-level plan hierarchy. Plan operators for the blocks world have been devised for testing purposes. Operators for a CAD application have yet to be devised. Facilities for plan re-use are currently in the design stage.

We have implemented a meta-interpreter in Prolog and used it to explain plans generated by WARPLAN [Warren 1974] (a non-hierarchical, linear planner), as well as for explanation for an expert system written in Prolog. The meta-interpreter is similar to that presented in [Clark 1982, Sterling 1986]. The integration of this facility with the POOL system is straight-forward.

8. Related Work

8.1 Distributed Problem Solving

Chandrasekaran [Chandrasekaran 1987] has proposed a functional architecture for artificial intelligence that involves cooperative problem solvers, each of which is designed for handling a generic task. A number of such tasks have been identified and implemented for domains involving diagnosis and design. Representative generic tasks include hierarchical classification, database inference, and object synthesis. The advantage of this approach is in supporting a higher level of abstraction by providing task-oriented problem solvers. Further research is needed in identifying other generic tasks and in coordinating their activities.

Work in actor languages [Agha 1986] has provided a theoretical foundation for computation in distributed systems. Concepts such as open architecture and acquaintances are also provided in POOL [Koegel 1987b], which differs from actor languages in that agents are deductive rather than applicative.

8.2 Interactive Planning

Most planners which support interactive planning have dealt with it in the context of plan execution monitoring [Wilkins 1984, Broverman 1987]. The planner interacts with the user to the extent that it monitors the actions performed by the user, and attempts to recover from any deviations or surprises. We have not considered this type of interactive planning since few CAD environments currently support on-line construction of the design.

A medical expert system ONCOCIN has been developed which allows user plans to be critiqued [Langlotz 1984]. The approach is based on having the system compare the plan produced by the user with that the system produced for the same problem. This technique would be relevant for the user interface for the planner.

9. Summary

This paper presents planning as a distinct and generalizable component of the design process in CAD. An informal model of such planning is derived from descriptions of engineering design in three different domains. The GDP model presented in [Koegel 1987a] supports this informal model and allows extensions in areas of design that currently are not fully representable. The GDP model is mapped to a planning program architecture which combines features not found collectively in earlier planners, including distributed problem solving, interactive planning, domain expertise, and plan re-use. The domain representation and constraint management approach used in this architecture are also significantly different from other planner programs. Also significant is the proposed mechanism for explanation in a distributed system of deductive agents. Substantial implementations exist for many of the components of the proposed architecture.

References

[Agha 1986] Agha, G. *Actors--A Model of Concurrent Computation in Distributed Systems*, MIT Press, (1986).

[Brewer 1986] Brewer, F.D., and Gajski, D. D. An Expert System Paradigm for Design. *23rd IEEE Design Automation Conference*, pp. 62-68, (1986).

[Broverman 1987] Broverman, C. A., and Croft, W. B. Reasoning About Exceptions During Plan

Execution Monitoring. *AAAI-87*, Seattle, Washington, pp. 190-195.

[Brown 1983] Brown, D.C., and Chandrasekaran, B. An Approach to Expert Systems for Mechanical Design. *Trends and Applications '83*, IEEE Computer Society, (May 1983), pp. 173-180.

[Brown 1985] Brown, D.C., and Chandrasekaran, B. Expert Systems for a Class of Mechanical Design Activity. *Knowledge Engineering in Computer-Aided Design* (ed. J.S. Gero), North-Holland, (1985), pp. 259-282.

[Brown 1986] Brown, D.C., and Chandrasekaran, B. Knowledge and Control for a Mechanical Design Expert System. *IEEE Computer*, (July 1986), pp. 92-100.

[Chandrasekaran 1987] Chandrasekaran, B. Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks. *IJCAI 87*, pp. 1183-1191, (August 1987).

[Chapman 1987] Chapman, D. Planning for Conjunctive Goals. *Artificial Intelligence* 32, pp. 333-377, (1987).

[Clark 1982] Clark, K. L., and McCabe, F. G. Prolog: A Language for Implementing Expert Systems. *Machine Intelligence* 10, pp. 455-475.

[Jaffar 1987] Jaffar, J., and Michaylov, S. Methodology and Implementation of a CLP System. in *Proc. of the Fourth International Conference on Logic Programming*, (1987), pp. 196-218.

[Koegel 1987a] Koegel, J.F. A Theoretical Model for Intelligent CAD. *First Eurographics Workshop on Intelligent CAD*, Noordwijkhout, The Netherlands, (April 1987).

[Koegel 1987b] Koegel, J. F. POOL: Parallel Object-Oriented Logic. *Proc. of the 2nd Annual Rocky Mountain Conference on Artificial Intelligence*, Boulder, CO, (June 1987).

[Langlotz 1984] Langlotz, C. P., and Shortliffe, E. H. Adapting a Consultation System to Critique User Plans. *Developments in Expert Systems* (ed. M. J. Coombs), Academic Press, (1984), pp. 77-94.

[Sacerdoti 1977] Sacerdoti, E. *A Structure for Plans and Behavior*, Elsevier-North Holland, Inc., NY, NY, (1977).

[Sacerdoti 1981] Sacerdoti, E. Problem Solving Tactics. *AI Magazine* 2 (1), pp. 7-15, (Winter 1980-81).

[Stefik 1981] Stefik, M. Planning with Constraints (MOLGEN: Part 1). *Artificial Intelligence* 16, pp. 111-140, (1981).

[Sterling 1986] Sterling, L. and Shapiro, E. *The Art of Prolog*, MIT Press, (1986).

[Sussman 1975] Sussman, G. J. *A Computer Model of Skill Acquisition*, Elsevier Publ. Co., NY, NY, (1975).

[Sussman 1980] Sussman, G.J. CONSTRAINTS--A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence* 14, (1980), pp. 1-39.

[Warren 1974] Warren, D. H. D. WARPLAN: A System for Generating Plans. U. of Edinburg Tech. Rep. (June 1974).

[Wilkins 1984] Wilkins, D. E. Domain-Independent Planning: Representation and Plan Generation. *Artificial Intelligence* 22, pp. 269-301, (1984).

Examples of Geometric Reasoning in Oar

F. Arbab

This paper will be available during the Workshop

IDDL: The Language of a Family of Intelligent, Integrated, and Interactive CAD Systems (IIICAD)

P. Bernus
P.J.W. ten Hagen
P.J. Veerkamp
V. Akman

IDDL: the language of a family of intelligent, integrated and interactive CAD systems (IIICAD)

Péter Bernus, Paul J.W. ten Hagen
Paul Veerkamp, Varol Akman

Interactive Systems Department
Centre for Mathematics and Computer Science†

Abstract: We present the design of IDDL (Integrated Data Description Language) which consists of two main parts. For the static representation of concepts IDDL provides an object oriented database integrating logic programming and object oriented programming. Representational capabilities of the language are selected according to special needs of CAD data representation and apply the results of former studies about IIICAD systems. These include several non-classical logic features of the language. The dynamic part of IDDL manipulates the database in a forward reasoning manner by executing rule base scenarios and maintains multiple worlds of the design database. This reflects the evolutionary aspect of design development. We present the application of these concepts to design process representation. Also an envisioned architectural plan for IDDL based IIICAD base system is given.

Keywords: Intelligent CAD, object oriented programming, logic programming, conceptual modeling, forward reasoning, non-classical logics, multiple worlds, CAD systems architectures.

1. Introduction

Considerable effort has been spent over the years aiming at the integration of extant design tools in Computer Aided Design. As it was shown in [21] and [33] design process integration is not only an interfacing problem, but calls for a fundamental understanding of the design process itself. Integration is routinely tackled by human designers when they are using computer tools in their everyday work. It reasonable to expect that future CAD systems should carry out or at least better support this tedious task. Several major obstacles seem, however, to prevent any simple solution.

The CAD system — in its role of a designers assistant — is only capable of such integration if it captures the meaning and relevance of operations carried out by the human designer. Thus a designers assistant must maintain a representation of the design process. The language of interaction in design as many authors argue (see again for example [21] or [8]) is only partly verbal and in many cases pictorial (although much of it symbolic and not "impressionist" i.e. even graphical interaction is symbolic, but it uses its own kind of primitives and has a multidimensional syntax). A rich medium for interaction is needed to convey dialogues using such languages. In addition, multiple levels of abstraction, nested contexts (with goals, motivations and plans) and implied knowledge is characterising design interactions envisioned by us [7] . The interaction medium between the designer and the designers assistant serves two equally

† Interactive Systems Department Centre for Mathematics and Computer Science Kruislaan 413, 1098 SJ Amsterdam, NL phone: +31-20-592.9333, fax: +31-20-592.4199 E-mail: CSNET - {pb,paulh,pauljan,varol}@cwi.nl UUCP - {seismo, munnari}!mcvax!{pb,paulh,pauljan,varol}

important purposes. Conversation using a particular language is the more understood usage. The second purpose is to provide a common modelling facility which can equally be manipulated and perceived by both partners. We expect that conventional interaction tools will not suffice for building appropriate interfaces along these lines.

The requirements that a suitable representational environment (comprising one or more languages) must fulfill, therefore, are influenced not only by the need of design process representation and design object modelling itself. Sophisticated *and* still efficient verbal and graphical interactions, dialogues and discourses also need to be represented.

If the CAD system is made aware of the design process from the early conceptual stages, the forthcoming steps can meaningfully be interpreted and evaluated by the computer in order to act and react in a helpful way. Intelligence to make early interaction possible and integration of the multitude of design related models seem therefore to be prerequisites.

A generic, theoretically well-founded framework for representing design processes in general had to be established (as it was done in [32]) and requirements for an appropriate representation language had to be analysed (see [36]). This essentially top-down analysis method yielded a host of requirements. Throughout this paper we will use the results of this requirements analysis and refer to investigations that we have made for assessing the feasibility of these.

Most of the requirements address the advanced information technology concerning logic and object oriented programming. Such language constructs and the mechanism that blends them together, needs to be embedded in a language that has constructs which are directly relevant for the design activities.

We present in this paper IDDL, the result of an ongoing language design effort. IDDL has two main components. The static component captures conceptual aspects (Section 2.). The dynamic component captures the way in which concepts are manipulated (Section 3.). Section 4. puts IDDL into the perspective of a CAD system architecture which can be based on the language.

2. Representation of concepts in IDDL — the static aspect

In the first place IDDL is a language to describe design objects in a CAD environment. The specific needs of design can very well be met by applying a logic language as a primary tool as many authors have argued in the past [16] . Previous analysis [36] has shown, however , that the logic behind the language needs to have several non-classical characteristics. This involves the use of nonmonotonicity [2, 24, 29] , non-classical rules of inference [35] and the enrichment of the representational tool by several modalities (like knowledge and belief [19, 26] or possibility and necessity [18]).

The basically flat database of logic programming languages are not adequate for large CAD databases from either efficiency or software-/ knowledge engineering points of view. The techniques used for overcoming these difficulties can roughly be divided into two categories: the introduction of higher order capabilities into the representation language (see for example [25, 27]) or the introduction of frames, scripts, objects or other forms of data abstraction techniques [13, 15] . The large amount of data to be handled in CAD knowledge bases poses another problem area: integrating knowledge base and database management techniques for efficiency purposes [3] . Object oriented programming can serve as a powerful paradigm for knowledge modularisation and information hiding. Another important notion in this paradigm, inheritance, is also appealing but needs to be implemented in a flexible enough way, with the needs of CAD data representation in mind.

IDDL is based on the integration of logic programming and object oriented programming paradigms. Section 2.1 gives an outline of the corresponding subpart of IDDL. Section 2.2 discusses some of the issues an implementor of IDDL is likely to face and theoretical decisions made in the design of the language in order to avoid complexity pitfalls or usual discrepancies as incompleteness or unfairness. Section 2.3 presents how the static part of the language can be put into use.

We have taken a conservative course of development: first we defined a basic language layer with few new features and then we analysed the way and the effects of adding the non-classical logic extensions.

2.1 Objects, functions and logic

2.1.1 Objects and logic

In the (conceptual) modelling task there is a need to build knowledge structures with both complete and incomplete knowledge. Conceptual models are built using objects. These objects have particular properties which makes it possible to treat them as either manipulable entities or to extract information from them in the form of logical constructs which can be added to the facts base. Moreover each object has:

- an abstract, outside view of objects which provides access to one well defined subset of the information it contains (i.e. using functions which map objects to their abstracted properties).
- a concrete, inside view which provides access to every aspect of the object and which informationb can be expressed as a collection of facts. (Such facts stand for internal representation of properties or integrity constraints.)

The total knowledge base is the collection of all objects tha have been created together with their current status (e.g. values of their attributes etc.).

Due to the control structures of the language (to be described later when scenarios are introduced) one can identify a particular configuration of objects. Such configuration may be built of

— global objects
— local objects
— relationships between the objects[1]

Such a configuration of objects and their relationships provide in a straightforward manner a partitioning of the knowledge base.

At any point in time a further functional separation of a knowledge base (~partition) is to call the set of available objects an objects base and the relationships between them a facts base.

There is a mechanism to construct a facts base from a given object configuration. The basic action is to add or remove a *clause*[2] to or from the facts base. In case of the objects viewed from the outside these clauses are also constructed with the help of functions defined as object-attributes. Such functions parallel with transaction methods of Smalltalk-80. These are in contrast with inquiry methods which parallel pure functions of IDDL to present abstracted properties of objects and which have no side effects.

Object functions are evaluable *terms* which in turn appear in *clauses*. In the present version

---

[1] Normally only the outside view of these objects is provided unless (using appropriate control structures) they are explicitely entered into.

[2] Throughout the text we italicise words referring to language elements which appear in Appendix I.

of the language we basically restrict the facts base to contain HORN clauses. There is an interpreter called the query processor (QP) to answer queries about the facts base. Since queries also contain evaluable function ecxpressions the answer is also influenced by the so-called exported functions of objects (see. Section 2.1.2)

The result of the queries is used by *scenarios*. In case of an object seen from the inside some further (explicit or implicit) constructs exist to build the facts base. Special functions can be defined for instance for initialisation of objects to build their inside and outside view at instantiation time. Technically this amouns to adding clauses to the facts base or to the objects inside facts base (see below).

The entire mechanis is controlled in scenarios. Scenarios can also manipulate objects in ither ways, e.g. by changing function definition they contain (i.e. assigning values to object attributes). By this functionality the scanarios may change the clauses which will subsequently be extracted from objects.

A world is a particular configuration of objects as it appears in scenarios. It is possible to view each world as an object, the definition of which is implied by the scenario definition. For the time being as few automatisms as possible are assumed in connection with scenarios. Hence they describe their world fairly explicitly.

The workspace of IDDL is a database called the "global world". In this global world we find *objects*.

Objects which have more to them as a name own an associated database. For such objects it is possible to speak of their inside. The structure (database) of such objects is made of:

- *object declarations*,
- *function definitions*,
- a *facts base* and
- *constraint definitions*.

The basic language construct concerning objects are *object definitions* and *object declarations*. Object definitions are explicitely written in IDDL for formal definition of objects (e.g. for those which will be used to represent classes or prototypes of objects). Object definitions are also implicitely created by scenarios which can create an object (possibly using a prototype object definition). This process is called instantiation. When instantiated objects are assigned rigid designators (internal names) to them; these references are assigned to the names (constants of the language) which will refer to these objects.

See an example object definition on Fig.1.

Object declarations which appear in object definitions are references to other objects. We call these *static object declarations*. Their function is to differentiate constant names used in the given scope whic stand for other objects and those which have no objects behind them. This declaration is the counterpart of dynamic object declarations which can be used in scenarios, to define certain variables to stand for a list of possible objects later to be instantiated.

The *facts base* of an object contains *clauses*. Together with the *function definitions* the object's facts base behaves like a database which can be queried by the same query processor which is used to query a facts base created by a scenario.

The basic data structures for this subset of IDDL are *terms*. A *term* has the form: $f(t_1,t_2,...)$ where f is a constant *function symbol* and t_1, t_2 are terms.

As usual, if the number of subterms is zero, the term is called *atomic* or just a *constant*. Examples are: *a* 1, *pancake* , 164, 1.6*E*–2, *tyuhaj*. Variables are also allowed as terms. For terms like .(*a* ,.(*b* ,.(*c* ,*d*))) we introduce the *list* notation {*a* ,*b* ,*c* ,*d* }. { } stands for the empty list .().

The QP can be accessed from the language or from a front-end to a user. In both cases there are built-in predicates (and built-in functions as well) to issue queries. The single query *call* (*p* (*t*)) finds one set of ground terms for the free variables in *t* for which *p* (*t*) holds; the query *setof* (*t* ,*p* ,*S*) binds *S* to the list of all possible ground instances of *t*.

We also introduce *call* [*p* (*X* ,*Y*)] as an evaluable function where the result of the evaluation will be a possible binding or { } if there is none. Similarly *setof* [*t* ,*P*] evaluates to the same list which otherwise would bind to *S*. By definition a list of the form {*t* ,*S* } is called an instantiation pair list, where elements of *S* are ground instances of *t*.

When a query is a *setof* -type, backtracking automatically collects all bindings. This will require some careful design to avoid pitfalls of inefficiency[28] and unfairness or falling into infinite loops. For instance the search in the SLD tree will therefore have to divert from a purely depth first mode (see [23] page 59.).

Note that there are no built-in predicates in this part of IDDL, which could modify the facts base (e.g. assert, retract) while proving a goal. Real assertions shall be handled by a separate interpreter called "supervisor" which executes *scenarios* (see Section 3.1).

2.1.2 Functions

In Section 2.1.1 we only had objects which were completely sealed off from each other. We introduce functions into IDDL in order to connect objects. There are two questions:

a.) How much of an object's database can be seen from the outside?
b.) How can an object refer to other objects' outside view?

Exported functions of objects are similar to inquiry methods of Smalltalk-80.™ They create the outside view of objects.

An object can refer to other objects by *(remote) function expressions. Function expressions* can be thought of as evaluable *terms*, E.g.: *f* [*x* ,*Y*], where *f* is a defined function name and *x* and *Y* are terms. *Function expressions* can appear in the body of clauses and in the body of function definitions.

The evaluation of *function expressions* is carried out using the *function definitions*. Evaluable functions bear slight resemblance with arithmetic expressions (terms) of PROLOG [10] evaluated by PROLOG's built-in evaluable predicates. In IDDL evaluation is not limited to terms which appear in a certain subset of predicates. Also *function expressions* may appear in the place of *predicates* . The occurance of a function expression determines if it is to be treated as a term or as a predicate.

2.1.2.1 Local and remote function expressions

A *function expression* may appear in the facts base of an object if there is a corresponding *function definition* in the object's data base. Such function expressions are called *local*. When we want to use functions which are defined in other objects we use *remote function expressions*. For example:

™ Smalltalk-80 is a trade mark of Xerox Corporation

$o_2::f_2[x,Y]$ and $o_3::'f_3[x,Y]$

are remote function expressions, meaning function f_3 of object o_2 and function f_3 of object o_3 respectively. Here o_2 and o_3 must be valid object names in the scope of the object's database at the time these remote function expressions are evaluated. We can even write $X::f_4[x,Y]$ if we can make sure that by the time the evaluation of the expression is attempted, X is bound to a valid *object name*.

There is an object with the (reserved) object name *global*. Thus there can be remote references to globally defined functions (Example: $global::f_g[x,y,z]$ or $global::'f_g[x,y,z]$). The local object can also be referred by the reserved object name *self*.

Remote function expressions are normally evaluated with respect to the object in which they have been defined. On the other hand quoting (e.g. $o_2::'f_2[x,Y]$) will import the function definition (found in o_2) and evaluate with respect to the object where the expression appeared. If there was no ' quote, the evaluation would have taken place with respect to o_2. For shorthand the prefix $global::'$ can be omitted.

Remote function expressions should only be used within the body of function definitions. This helps create clear-cut object interfaces.

```
DEFINE_OBJECT(automobile,
  OBJECTS(OBJECT(e 1),OBJECT(b 1),OBJECT(b 2),OBJECT(b 3),
    OBJECT(w 1),OBJECT(w 2),OBJECT(w 3),OBJECT(w 4),OBJECT(w 5)),
  FACTS({color(red)., engine(e 1)., body(b 1)., wheels({w 1,w 2,w 3,w 4,w 5}).,
    has_part(e 1)., has_part(b 1)., has_part(b 2)., has_part(b 3).,
    has_part(Wheel)←wheels(Wheel_list) & in(Wheel,Wheel_list).}),
  FUNCTIONS({
    EXPORTED_FUNCTION(make_of_the_car[],alfa_romeo),
    EXPORTED_FUNCTION(consumption_is[At_speed],

      case[{   ≤(At_speed,90),                    5.6),
               >(At_speed,90) & ≤(At_speed,120),  7.8),
               >(At_speed,120),                   8.2)}]),

    EXPORTED_FUNCTION(color[],
      car[setof[X,color(X)]]), /*the first X found for color(X) */
    EXPORTED_FUNCTION(part_list[],setof[Part,has_part(Part)]),
    FUNCTION(date_of_purchase[],1985),
    EXPORTED_FUNCTION(age[Current_year],
      -[Current_year, date_of_purchase[]])}))
```

— Figure 1. —

2.1.2.2 Function definitions, exported functions

The forms:

FUNCTION(*head*,*body*) and *EXPORTED_FUNCTION*(*head*,*body*)

are called function definitions. The *head* of a definition has the form of a function expression (e.g. $f[x,Y]$), where f is the function symbol to be defined, followed by zero or more *arguments*

within brackets. Formal *arguments* should not contain function expressions.

The *body* is a *term*. The intended meaning of the function definition is to treat it as a term rewriting rule *head* = *body*, so that *head* is replaced by *body*, where variables in the *head* are treated as formal parameters to the *body* (we may quote ' actual parameters to delay their evaluation at replacement time). The body may contain any kind of term (local and remote function expressions as well). Specifically calling the QP by *call*[] or *setof*[] is allowed and so the facts base of the object can be accessed. Exported function definitions can be called from other objects who know the given object, ordinary function definitions can only be used locally. There are also *built-in function symbols* we can use in the *body* (e.g. car[], cdr[], eval[], do_while[], if_else[], case[] etc.).[3]

The example on Fig.1. shows how these constructs can be utilised for abstracting properties of an object and provide them for the outside in form of exported function expressions.

The facts which are true in an object's database may not be independent of the facts true in other objects. We can propagate facts from using remote function expressions. Function definitions can be propagated by the quoting mechanism.

2.2 Implementation trade-offs

2.2.1 Object declarations, higher order effects, intuitionistic logic

2.2.1.1 Static object declarations

In the static part of the language constants can be thought of as object names. It is a question, however, which object is denoted by a given name.
If we consider an an object name as an attribute, than the object which the name denotes is considered the value of that attribute.

An object name is known in the scope of the object's database, where its object declaration appeared. The form of a *static object declaration* is:

OBJECT(*obj*,*term*)

The meaning of the above declaration implies an implicit declaration *FUNCTION*(*obj*[],*term*), so that *obj* is understood as a constant name and *obj*[] as an evaluable term. Here *term* must evaluate to an object[4].

This approach is very useful if object declarations are used to represent attributes. We can both represent facts about the attribute (e.g. *p*(*obj*)) and facts about the "value" of the attribute (e.g. *q*(*obj*[])) in the same facts base.

In the simplest case *term* evaluates to a rigid designator of an object (e.g. *#obj_*000072). Rigid designators have a system-wide scope. In remote function expressions as *t*::*f*[] the term *t* must evaluate to such a rigid designator. Normally the user of the language does not have to know about rigid designators, because they are created as new objects are defined. So they do not appear in the language definition.

2.2.1.2 Definition of predicates by objects

An object can also be used to define predicates. We may have an object called *triangle* to represent all the knowledge that is necessary to judge whether an object qualifies for being an

---

[3] Built-in function names can be thought of as if they were defined functions of the *global* object.

[4] Evaluating to an object means to evaluate to a unique (rigid) object designator or evaluating to some immediate primitive object (e.g. {} or a character etc.).

instance of a triangle and also the knowledge of how one such instance can be created. Here *triangle* is an object which is roughly corresponding to a class in Smalltalk-80. There are two ways to introduce objects as predicate definitions:

1.) Objects (modules) can be introduced as parametric predicates [25] . In this approach an object name is a predicate name also and if the object (module) is made accessible to an other object (module), then the imported predicate can be used for proofs. This is more in line with modularising PROLOG.

2.) We can have objects which stand for classes (to define class membership predicates) and objects which stand for instances of classes (qualified by the class membership predicate) — this is more suited to the object oriented way of thinking.

The first approach has the disadvantage that we cannot modify the intension of the predicate by locally manipulating contents of the module's. Therefore the first approach lacks the extensional form of representation which, for CAD applications, has been shown in [34] to be superior to direct intensional representations.

In the second approach an object which defines a predicate can have an exported function (method, attribute) for membership test. An internal change to the implementation of the module may create a new intension for the defined predicate. Therefore the second approach is more flexible and we follow that one.

Note that the the decision will not inhibit us using the computational semantics developed for the first. An important point is that predicates defined by objects (modules) can be interpreted in an operational way so that restricting inference rules in an appropriate way the logic of the language will be intuitionistic (or minimal) rather than classical. More exactly saying: if the database of an object is checked against the *constraints* defined in the object and is found to be consistent, the derivable consequences of the object's database are intuitionistically true while in-between only minimal logic applies. (See [25] for the treatment of the questions touched upon in this paragraph).

For queries about a predicate defined in an object consider the following example:

DEFINE_OBJECT (*o* ,
...*FUNCTION* (*accepts* [*OBJ* , *TERM*] , *OBJ* ::*do_you_accept_this* [*TERM*])...
FACTS (... *p* (*X* ,*Y* ,*Z*) ← *accepts* [*p* , {*X* ,*Y* ,*Z*}]. ...))

where we supposed that *p* as an objet has a function *do_you_accept_this* , and that this function waits for an actual parameter of the form {*X* ,*Y* ,*Z*} to be supplied. The evaluation of the function can be true, false or unknown. Also note that *accepts* is a function defined locally to *o* .

For asserting *p* (*X* ,*Y* ,*Z*) we must create an instance of the module (object) with the parameters instantiated. The object which defines the predicate *p* must have functions which carry out the assertion. An assertion, therefore, may have slightly different meanings for different predicates. A class can have (or inherit from some other object) a function to create a new instance of itself. We may later establish pools (named inventories of objects) in the *global* object for dealing with predefined multipurpose object packages.

2.2.2 Functional programming

The existence of evaluable function expressions (terms) — which we introduced as a bridge between the objects — influences the behaviour of the query processor (QP).
Also the definition of functions introduces a functional programming feature into the language. Functional programming in its generality can be considered as inferencing under equational

theories. This effects the cost of the theorem proving task of the QP. We shortly consider the trade offs involved.

The main issue here is: how to mold the unification function of the logic programming part with the binding process which takes place when function expressions are evaluated. In the literature several approaches are known for this purpose. One of the early approaches was taken in the design of LOGLISP, where logic and functional variables had been separated [30] . Semantic unification [31] , as a special case of unifying under an equational theory, is another one of the relevant techniques we might use. A third way which results in a more deterministic evaluation of function expressions is described in [6] . (Several modified forms of unification allow defined functions into logic programming — see [4] for an extensive study of the question).

We would like to leave the query processor as it is in the pure HORN-clause case (using basically SLD resolution techniques) and add a co-processor to it which evaluates function expressions (the evaluator). Since SLD resolution amounts to subsequent application of substitution and unification processes, the problem is really how to unify terms which contain evaluable function expressions.

The idea is to try ordinary unification first. If an evaluable term is encountered then evaluation is tried and a subsequent unification attempt is made. Unification of terms involves unification of subterms, so the question is: in what order to evaluate subterms.

— Should the subterms be evaluated first and then passed to the *body* of the function expression, or

— Should the subterms be substituted without evaluation into the *body* of the function definition and evaluation only take place when the need arises?

Intuitively the second, lazy evaluation is more effective and makes more liberal programming possible (for instance unbound variables in an else branch of an if_then_else function expression never have to be bound if we are sure the condition will be true). In spite of this the programmer should be given the liberty of controlling subterm by subterm in the function definition whether to prefer immediate evaluation. In some other cases unbound variables are necessary as for example in calling the QP from within the body of a function definition.

It is disputable whether a resulting term (which by now may not contain function expressions) has to be ground or it still may contain free variables. In case we allow variables, the net result is that the evaluator must allow symbolic computation or the evaluation must be suspended in the given goal and using and-parallelism the next subgoal should be attempted. If a function expression as a subterm could not be evaluated to a ground term, and there is a chance to prove the corresponding subgoal in which it appeared, we can temporarily assume that the subgoal is true and proceed. Later subgoals may bind such variables and thus the strict left to write evaluation of subgoals is not enforced.

We expect substantial reduction of the problem by not allowing evaluable terms into the heads of facts base clauses and heads of function definitions. This makes the evaluation process one-way.

2.2.3 Truth maintenance

When an exported function is evaluated or the QP tries to answer a query, the facts used during the theorem proving process can be recorded. The usual application of truth maintenance (to enhance the performance of theorem provers) would here be extended by supporting the evaluation process of function expressions as well.

This would mean a fairly big storage overhead considering the possible long reference chains through which function expressions finally get evaluated, a trade-off should be made: how much of the utilised facts to record as supports. From the design point of view we think that maintaining a status history for objects and flagging already evaluated function expressions by the names of objects which got involved in the evaluation could save most of the reevaluation of attributes. This is at least the case for objects which are used in the design process as invariant building blocks.

Whenever some change occurs in an object previously involved in the evaluation of a function expression reevaluation is due to happen. We shall have to reconsider this preliminary statement when tried on a number of real example cases. For further application of truth maintenance principles see Section 3.2.1.

2.2.4 The behaviour of the IDDL database

The database of an objects or a world in general (see. Section 4.) should ideally show the functionality of a deductive database. We should strive to it, since the dynamic layer of IDDL builds on top of the static layer in such a way, that database queries become elementary operations. Consequently these operations should as much as possible take constant time.

2.2.4.1 Terms:

As it is apparent from the syntax definition (Appendix I.) we allow compound terms as basic data structures in IDDL. Deductive databases mainly avoid this. The complete abstinence from terms makes completion procedures possible so that the database appears to store both explicitly and implicitly defined facts. In case of unlimited forms of terms completion may lead to potentially infinite amount of facts which is clearly an obstacle. Since dynamic recomputation of facts at every query may again be expensive just not in terms of storage but in terms of time, we have the choce left already explained in Section 2.2.3.

An alternative, as far as terms are concerned, is to introduce a type theory into the language so that types of terms form a hierarchy (see.[23] page 151.).

2.2.4.2 Database:

The full application of deductive database techniques in the implementation of IDDL has two difficulties at the moment (ref.[22] for details):

a.) It is unclear how the presence of evaluated functions could be made consistent with the usually required domain closure axiom for deductive databases. The process of completion for the database may therefore be unfeasible.

b.) We have to investigate the effect of having intuitionistic inference rules on the integrity constraint checks, since these algorithms have been developed for databases which rely on the SLDNF resolution theorem proving (SLD resolution with negation by failure).

2.3 Objects representations in use, inheritance

Objects can be used to represent concepts. Functions defined over objects can represent attributes of concepts. More exactly saying: the function-name is representing the attribute and

the result of a function-evaluation is representing the value of the attribute It may well be the case that a function is defined but cannot be evaluated yet (i.e. the concept has an attribute, but the attribute has no value yet).

Classification of concepts makes the representation task easier and different taxonomies can be built for usefully arranging the concepts which we deal with in design. These arrangements are called taxonomies and give account on how exactly different concepts' attributes and/ or attribute values relate to each other.

Since Simula 67 [1] the notion of class-subclass and class-instance inheritance has been a central and successful technique in knowledge representation.

The needs of knowledge representation systems have however soon added more complexity to the simple inheritance hierarchies, introducing multiple inheritance, handling of exceptions, defined (instead of predefined) inheritance [14] and delegation [5] . IDDL makes inheritance on two levels possible. When scenarios create new objects out of prototype objects, a strict inheritance seems to take place. We should notice, however, that the prototype itself (as an object) can define, in addition to default values for attribute values a number of other ways of inheritance.

As we can see, the terms which result as a value after the evaluation of a function expression (see example on Fig.1.) can have different rationale:

constant terms like	*make_of_the_car* [] or *date_of_purchase* []
computed terms like	*age* []
conditional values like	*consumption_is* [*At_speed*]
derived terms like	*part_list* [] and *color* []

Remote function expressions can even borrow from other objects values of their attributes or methods to compute them (if the quoting mechanism is used). Since the value of attributes is computed by the evaluation of the *body* of the corresponding function expressions, it is up to the creating scenario and the writer of the prototype concept what kind of taxonomic relationships to establish. In particular, there can be two ways of propagating attribute values from object to object:

- The scenario which assigns a value to an attribute consults the attribute value of an object and assigns it to the other objects attribute. (We may call that inheritance by value.)
- The attribute value (whether assigned by a scenario or being there on account of a prototype object's definition) is not a immediate value, but is itself an evaluable term. This creates the attribute value on demand. The demand can be granted locally or via remote function expressions from other objects. Such attributes may change if the attribut values of other objects which were involved in the computation change. (We may call that inheritance by reference.)

3. The dynamic aspect: the facts-base and scenarios

The dynamic part of IDDL consists of a facts-base and and a set of scenarios. The facts-base contains definite program clauses; they denote the facts that are currently known about the artifact. The scenarios contain if-then rules, they perform the (forward) reasoning about those facts. The rules denote the somewhat invariant knowledge about the designing; i.e. how designing takes place. The clauses express the more variant knowledge about the design objects themselves. Consequently, the rules will not change during the design process, whilst the number of clauses will grow significantly.

Resulting from this separation, we may conclude that IDDL acts as deductive database language concerning the facts-base and it acts as a modular production system in case of the scenarios. As a consequence, these parts have a somewhat different syntax and its own interpreter. its the next sections we will describe the facts-base, scenarios, worlds, object declaration, rule selection control and the modal logic operators in this order.

3.1. The facts-base

The facts-base is used for reasoning about objects and relationships among objects. In this section we shall show the essential features of the facts-base. It is a deductive database whose language is built up from definite program clauses and unit clauses [10, 23]. Therefore, if a query is posed to the facts-base, the answer is either directly available through a unit clause, or it is derived from a (series of) program clause(s). To define definite program clauses we will first introduce an alphabet, terms, atoms and literals. The alphabet consists of six classes of symbols:

a. *variables,* symbols beginning with an uppercase letter (e.g X, Y, AContainer).

b. *constants,* symbols beginning with a lowercase letter (e.g. a, b, aContainer). *Number constants such as integer, real etc., as well as string constants are defined as usual (e.g. 123.0, -456, -3.45E3, 'aString', 'Tom''s house').*

c. *functions,* symbols beginning with a lowercase letter (e.g. f, h, volume). *We use the skolem function, "_", to denote the unknown. Note that _ is equivalent to Prolog's "don't care" symbol. Some function symbols are predefined.*

d. *predicates,* symbols beginning with a lowercase letter (e.g. p, q, container). *Some predicate symbols are predefined. These include the unary predicate symbols: Cut is a non-logical annotation to convey a certain control facility, which can be applied in systems using a left-most search algorithm. Although it is used in the position of a predicate symbol, it is not a predicate and it has no logical significance at all. However, it is convenient to regard it as a predicate which succeeds immediately. (cut), fail, true, false, etc.* and the n-ary predicates: use, equals, gets, setof, etc.

e. *connectives.* They are limited to ←, & *and* ~. They are understood in the standard logic programming We assume that the reader is familiar with logic programming and Prolog. In this context we want to stress that IDDL as a deductive database language differs from Prolog in the sense that it does not permit side effects, i.e. assertions and retractions are not allowed. Also, we omit the disjunction operator ";".

f. *punctuation symbols,* "(", ")", "[", "]", "," *and* ".".

Over this alphabet we can define the terms as follows:

1. A variable is a term.
2. A constant is a term.
3. If h is an n-ary function (n>0) and $t_1, \ldots, t_n$ are terms, then $h[t_1, \ldots, t_n]$ *is a term.*

A *ground term* is a term not containing variables. If p *is an n-ary predicate (arity may be zero) and* $t_1, \ldots, t_n$ *are terms, then* $p(t_1, \ldots, t_n)$ *is an atom. A literal is an atom or the negation of an atom. A unary predicate is defined to be an object type and its ground term is the object name (e.g. in case of the predicate car(bmw), the interpretation is that the object bmw is defined to be a car).*

The facts-base is built up from definite program clauses. A definite program clause is a clause of the form

$$A \leftarrow B_1 \,\&\, \cdots \,\&\, B_n$$

A is called the head of the clause and $B_1 \,\&\, \cdots \,\&\, B_n$ is the body. $A, B_1, ..., B_n$ are literals. A clause with an empty body is called the unit clause, e.g.

$$A \leftarrow$$

We will now give an example of a facts-base with some definite program clauses:

```
material(concrete) ←
material(steel) ←
colour(X) ← colours(Cs) & element(X, Cs).
colours(setof[white, black, green, red, brown, yellow, blue]) ←
element(E, cons[E, _]) ←
element(E, cons[Hd, Tl]) ← ~equal(E, Hd) & element(E, Tl).
combination(concrete, white) ←
combination(concrete, black) ←
combination(steel, black) ←
```

In this example we have defined three types of objects: material, colour and colours. Furthermore we have defined the relationship element and combination. (Note that notequal is a built-in predicate; setof and cons are the built-in list constructor functions.) With this example we showed that one can either enumerate all individual objects explicitly (like material) using unit clauses, or one can define them implicitly (like colour) using definite program clauses. The scope of variables are restricted within clauses. That is, all variables in a clause are universally quantified over that clause.

The facts-base differs from a standard Prolog database. The clauses are not allowed to have side effects. Therefore, assertions and retractions are always performed from outside (i.e. via the scenarios). Consequently, the only purpose of the facts-base is to store objects and relations, and to answer questions about these objects. A query may be posed to the facts-base from the scenarios by asking for an atom. The facts-base can either give the answer immediately (since the atom is there as a unit clause) or it can derive it via definite program clauses. When the queried atom contains uninstantiated variables, a unification algorithm is used to find the instantiation-pair list (i.e. the variables are mapped to all possible constants).

So, if we ask material(concrete) to the facts-base described above, the answer will be "true". If we ask material(M), the answer will be "true" and M will be instantiated to one of {concrete, steel, wood}. The one which is actually chosen, will be determined by the supervisor. If we query material(M) combination(M, black), the instantiation-pair list will be {(M, steel), (M, concrete)}. In case of the query colour(green) the answer is derived rather than found directly. Note that the way an answer has been obtained is not visible to the scenario posing the query.

We say that the facts-base is goal-oriented. I.e. when we pose a query to the facts-base, it tries to match this goal with the head of a definite program clause. The atoms of the body of this clause are considered to be the new (sub)goals. This procedure is continued until all (sub)goals are matched with unit clauses. This kind of top-down search strategy is commonly called backward chaining.

3.2. Scenarios

A scenario is a procedure-like structure with an object declaration part, a function definition part and a body with rules. The set of objects, which are declared in a scenario, embody a *world;* a world is a partition of the facts-base and it consists of objects and the clauses concerning these objects. The function definition part is either the declaration of an externally defined function with its return object (e.g. dist[,] = INTEGER;*), or the definition of function which can be used locally during the execution of the scenario (e.g. dist[P1, P2] = { sqrt[(x[P1]-x[P2])**2+(y[P1]-y[P2])**2] };*). The first argument of the externally defined function is the object to which the function is sent. Either the object declaration or the function definition part may be omitted.

```
scenario name(Obj1, ..., Objn)
objects
    object1(Obj1); ...; objectn(Objn);
functions
    f1[] = OBJECTTYPE;
    .
    .
    .
    fs[] = { function body };
begin
    if condition then action;
                .
                .
                .
    if condition then action;
end scenario;
```

The rules consist of two components, a left and right hand side. The left hand side (LHS) is evaluated with reference to the facts-base and if this succeeds, the action specified at the right hand side (RHS) is performed.

The LHS and RHS are both called formulae. The LHS is defined as follows:

1. An atom is a formula.

2l. If F and G are formulae, then so are F∧G and F∨G.

The definition of the RHS is analogous. We have 1. and:

2r. If F and G are formulae, then so are F & G, F | G, F∧G and F∨G.

The connectives and are understood in the standard logical sense. The connectives & and | have respectively the same meaning except that the arguments of the former are evaluated in parallel whilst those of the latter are evaluated sequentially. This parallelism will be described in section 2.3.

The scope of variables in a rule are local to that rule. Local variables inside a rule do not need to be declared. However, a variable which is declared in the object declaration part of the scenario is global throughout the scenario and is known within the rules.

A rule is fired if the LHS is successfully unified with the facts-base. The rule tries to perform the action at the RHS, which is either an assertion to or a retraction from the scenario's world or it is the invocation of an other scenario. A scenario is invoked by the built-in predicate *use*. This creates a new subworld on top of the former, the parent

world, and the invoked scenario becomes active. We call a scenario active if control is passed to it. An example is given below:

```
if p(a)∧q(b) then use(sc1, p(a), q(b), r(c));
```

This rule reads: if both p(a) and q(b) are derived successfully from the facts-base, then use the scenario called "sc1" with p(a), q(b) and r(c) belonging to its world.

A rule succeeds if an assertion (retraction) or an invoked scenario succeeds. There are two ways to make a scenario succeed. The first possibility is when a *noMoreRule* situation arises. If there are no rules which can be applied, control is given back to the parent scenario and the child scenario succeeds. The second possibility is via the built-in predicate, *success*. When this is encountered control is given back to the parent scenario. A built-in predicate, *fail,* makes a scenario terminate unsuccessfully.

In case of successful termination of a scenario, an evaluation (meta) scenario is called to check the scenario's world on consistency with the parent world. If and only if both worlds are consistent, the rule which actually called that scenario will succeed. Otherwise some actions depending on the evaluation scenario will be performed (e.g. backtracking over the previous scenario).

3.3. Worlds

A world is a partition of the facts-base and is automatically created by the invocation of a scenario. The newly created world is called a *subworld* of the parent world. A world consists of the relationships between objects which are local to the scenario to which the world belongs. Changes that occur in the subworlds are reflected to the parent world only if the scenario succeeds and hence the rule which invoked the scenario succeeds. (The latter fact is not always true, but we will come to this in this section.) If the scenario fails, the parent world will not be affected We distinguish between local and global assertions. The former will not affect the parent world, while the latter will.

Using the same mechanism we can invoke two or more scenarios at the same time and thus we can create multiple worlds. In creating multiple worlds we make a distinction between an *and-* and an *or-mechanism.* In the following section we will give some examples. These examples serve as an introduction to the multiple world mechanism. At the end of this section we will give a more precise specification.

```
scenario example(X, Y, Z)
objects
    p(X); p(Y); q(Z);
begin
    if p(a)∧p(b) then use(sc1, p(a), q(Z))∧use(sc2, p(b), q(Z));
    if p(c)∧p(d) then use(sc3, p(c), q(Z))∨use(sc4, p(d), q(Z));
end scenario;
```

Let us consider the first rule. The two scenarios sc1 and sc2 are activated if p(a) and p(b) are both successfully queried to the facts-base. These scenario names are not necessarily unique. The rule might have evoked the same scenario twice with different objects. Two worlds are created:

```
world w1          |    world w2
    p(a) ←        |        p(b) ←
```

q(Z) ← | q(Z) ←

Both scenarios act upon their own world. After termination of the scenarios, control is given back to the parent scenario evaluating the rule. The rule succeeds when both scenarios terminate successfully and when both worlds are *consistent.* Consistency means that the facts-base does not contain any contradictory facts. It also means that object q(Z) *is affected by the changes and additions caused by sc1* as well as sc2. *When two or more scenarios update the same attribute, the value of this attribute should be the same in either world. Otherwise the rule fails and no changes whatsoever happen. We call this the and-mechanism.*

In the second rule above we have shown the or-mechanism. This rule also activates two scenarios. In this case the rule fails only when both scenarios fail. If one scenario succeeds and the other fails, nothing happens. However, if both scenarios succeed two copies of the facts-base are generated, one containing the results of sc3 *and the other those of sc4.* This separation continues as long as the scenarios working on them succeed.

The same mechanism applies to simply asserting facts in the scenario itself. Suppose we have the following rules:

if p(a) then q(a) ∧ r(a);
if q(a) q(b) then s(a) ∨ s(b);

Considering the first rule, both the assertion of q(a) and r(a) should succeed in order for the entire rule to succeed. On the other hand, in the second case with an or-mechanism we can have the situation that as well the assertion of s(a) as that of s(b) succeeds. In that case two copies of the parent scenario's world are generated. The process is continued concurrently with both copies. However, if only one of the assertions succeeds, the scenario continues with that fact asserted.

A mixture of these features is also possible. Consider the following rule:

if p(a) q(b) then use(sc1, p(a), q(b)) r(a,b);

In this case the scenario sc1 is activated together with an attempt to assert r(a,b) in the world of the parent scenario. If this assertion succeeds, once again two copies of the parent scenario are made. One is processed further with r(a,b) included, and the other is preserved, waiting for the results of sc1.

Its is clear that this multiple world mechanism may be applied to more than two cases. It then behaves in a similar way. Note however that ∧ has higher precedence than ∨. Suppose we have the rule:

if p(a) then q(a) ∧ r(a) ∨ s(a);

and p(a) is matched with the facts-base. The rule succeeds if q(a) is successfully asserted together with the successful assertion of r(a) or s(a) or both. In the latter case, once again two copies are made, one containing q(a) and r(a), and the other q(a) and s(a).

The control of the and- and or-mechanism is taken care of by the *supervisor.* This is a set of scenarios which controls all the facts-base traffic. The supervisor determines when and how to generate new copies of the facts-base. It may discard, in interaction

with the user, certain worlds when they do not look promising or they may be suspended for some time. This suspension means that the worlds are maintained and that they may become active after a while when the scenarios which were examined first finally failed.

3.4. Rule Selection Control

The order in which the applicable rules are fired influences the behaviour of the system. A certain rule may cause the failure of a scenario, while another rule might have prevented it, if it were called prior to that. Various remedies can be suggested to solve this problem of controlling the selection of rules.

A certain kind of control is given by means of subscenarios. The rules can be grouped together in subscenarios and they are controlled by using the built-in predicates fail and success. An instantiation pair list of a rule is a list of its variables together with their bindings, e.g. {(C, c1), (E, e312)} The purpose of such a list is to restrain a rule from being applied more than once with equal variable bindings. So, once a rule has been fired, it will not be fired again under the same conditions.

The simplest mechanism to control the selection of rules is to impose an *order* on the rules. The first applicable rule from top is fired first. To determine the next there are two possibilities. Either the search is started over from the top, or the search is continued from the previous rule in a circular fashion. This mechanism may be unfair in the sense that a certain rule at the bottom may never be reached, since other rules are chosen prior to that one.

Another mechanism is that all applicable rules are collected and from those the most appropriate rule is selected, satisfying some criteria. Several criteria have been suggested [11]:

i) *Data order*. Objects in the facts-base are ordered, and the rule that matches object(s) in the facts-base with highest priority is selected.

ii) *Generality order*. The most specific rule is chosen. A rule is more specific if it has more conditions to be satisfied in its LHS.

iii) *Complexity order*. The rule with the most complex condition is chosen. That is the rule which took the most effort to be unified with the facts-base.

iv) *Recency order*. This means choosing either the most recently executed rule (of course with a different instantiation-pair list), or the rule containing the most recently asserted (modified) objects in the facts-base.

v) *Meta rules*. These are rules about rules, and they contain information about rules and embody strategies for selecting rules.

The rule-control mechanism used in IDDL is a combination of all these. It is possible to select a new mechanism dynamicly at run-time. To achieve this we introduce a construct in IDDL called: *directive,* which is yet another built-in predicate. When directive(aRuleSelectionMethod) *is encountered, the rule selection method aRuleSelectionMethod* is picked by the interpreter. This method will then be used until another directive has come across. Out of the methods that might be chosen are the ones we have mentioned above. This set of methods may be extended at any time by the user, we thereby give all possible freedom to influence the behaviour of the system. When such a new rule selection method is introduced, the only thing to be done is to add enough "logic" to the

interpreter so that it can execute it. This is a very powerful technique.

Another feature of IDDL is the possibility to group rules together in so-called blocks. A block starts with a curly bracket { and ends with another }. Blocks are evaluated sequentially in a circular fashion. (This is a decision we made so far. We can imagine a block selection method being active, choosing which block to evaluate next.) The rules inside a block are evaluated using the current rule selection method. A built-in predicate *break* is used to leave a block and continue with the next block. However, when a noMoreRule situation occurs inside a block, that block is also exited and the next block is started to be evaluated. Below we give an example of blocks along with the built-in predicates directive and break.

```
scenario name(...)
objects
    object1(...);
begin
{ directive(dataOrder);
    if <LHS1> then <RHS1>;
                .
                .
                .
    if <LHSn> then directive(generalityOrder) & break;
};
{
    if <LHSk> then <RHSk>;
    if <LHSl> then directive(complexityOrder);
                .
                .
                .
    if <LHSm> then <RHSm>;
};
end scenario;
```

4. Design object models

4.1 Static relationships between models

Two properties of models must be pointed out. Firstly, any model built in a CAD system about the target- and process of design has to be apparently conforming with the model that the designer has in mind. Design object models are externalisations or representations of the designer's concepts. The designer should have a feedback to see that the CAD system understands the progress or status of design in the same way he/she does. This prevents us having to answer the metaphysical question whether the designer actually forms his concepts about the object being designed in form of the models we externalise. All we require that the designer be able to confirm if his concepts and the externalised models conform. These models, when manipulated, have to change or behave according to the theory which describes such models in general. For example, a relevant theory may be encoded in a knowledge base as, say, a qualitative physics system. Then this knowledge must be applicable to the model, and it is the responsibility of the design program to appropriately interface to this part of the knowledge base.

A common characteristics of design related models is their incompleteness. At a

given intermediate stage of design some of the maintained models cannot be completed in a predetermined way. Simultaneous valid developments are possible. Because this is an inherent property of design, the representation of design processes (e.g. the way they act on design object models), has to reflect such parallel courses of model-development.

We can look at design object models from two (inside and outside) points of view — just as we can look at any other object. The inside view of a particular model of a design object is represented by a collection of facts. We can manipulate it to evolve into a description of the design object which is consistent and complete from the viewpoint of the modelling activity.

One can use the design object model for building more complex models. For example suppose that we have an AC model of an operational amplifier described as a collection of facts about the resistors, capacitors and transistors integrated on a chip, and facts about their interconnections. If someone wants to analyse the given circuit, in order to determine its behaviour under certain circumstances, a theory dealing with such low signal models can be applied to these facts so that the desired results be deduced. The outside view of this circuit abstracts from the way the given amplifier is built out of its elements, but, through accessing its properties, can be used as an element to build AC models of higher level circuits such as radio receivers or measuring apparatus.

One can also look at models from the outside viewpoint in order to treat them in their capacity of being models of a given kind. It is possible to supervise the properties of these models and monitor their contribution to the overall goal of the design process. E.g. internal details of an AC model for a radio receiver are probably interesting only for the designer who wishes to create or tune the inner structure of the circuit. Even for him, AC models of the modules (like an operational amplifier) are at this level probably looked at from their outside. The same designer, when he wants to supervise his own work, can look at the same model from the outside. He is then considering properties of the model in its entirety – the sensitivity of the receiver, the completeness of the model to judge the progress of the design activity etc.

For understanding the use of the world mechanism proposed in the IDDL system, it is useful to differentiate a few types of relationships which can hold among design object models during the process of design.

α.) *Abstraction* relates two models whenever one represents a subset of facts expressed by the other. Because of the omission of details the more abstract model generally allows a broader set of implementations than the less abstract one. Such is the relationship between the inside view of the AC model for a radio receiver circuit and the outside view of it.

β.) *Aggregation*: An aggregate model is built using elements (building blocks) which in their turn are (in the general case) models themselves. An aggregate model, taken as the inside view of a concept (e.g."AC model of radio receiver circuit"), composes or aggregates elementary models which on this level are considered from their outside only (e.g. "AC model of operational amplifier"). As it is apparent from this example, lower level building blocks are abstractions of lower level aggregates. Properties of aggregate models may depend on properties of building blocks, i.e. we can speak about a property inheritance between them, although the way this is done is completely governed by definition of the individual cases.

γ.) *Class/instance* relationship holds between two models, where one of them describes a generic type (for example *lathe*) and the instance describes a specific entity (*a_given_lathe*). Classes have two aspects. The model class can serve as a prototype for any model in that class. There is some kind of inheritance of facts by the instances from the class. Instead of the rigid property inheritance we prefer defined inheritance rules which only in the pure extreme cases equal to strict one-to-one copying. The model class as a set itself has properties — first of all to characterise the set of its instances (list of models which are its members, cardinality of members, etc.). An entire model, with the hierarchy introduced by aggregation, may be built on top of a set of component models, therefore instantiation of hierarchical model classes may cause instantiation of its components. Again the instantiation procedure is only in its extreme a prototype copying action, so the way a given model class should be instantiated is a defined not a predefined process.

δ.) *Class/subclass* relationship can hold between two models if they both represent types (classes) of models. In the same way as property inheritance of the class instance relationship is defined rather than predefined it is only the extreme case when all the facts which hold in the class will automatically hold in the subclass. Using defined inheritance more complex propagation of facts can also be created among model classes.

ε.) The *development* of a model is the same model in a later stage of design, with missing detail filled in. Facts added to a world are labelled by the time of assertion. Monotonic and nonmonotonic developments of a model are both possible. IDDL's nonmonotonic operator #D can be used to assert default facts. Consequences of such facts give rise to nonmonotonic developments of models. Nonmonotonic developments label consequent facts by the default fact from which they were deduced.

η.) The *concurrent* of a model is a possible alternative, i.e. two or models are concurrent if they are the developments of a common ancestor. Concurrents may not (do not need to) know of each other, i.e. a world and a scenario working on it may not notice that there is a concurrent version in progress. Modal propositions believed to be true in a model are influenced by the concurrents of that model, since concurrents are *accessible worlds* to their common ancestor in sense of the possible world semantics of propositional attitudes given first by [20] and [17] (See [19] for newer references).

The types of relationships we have listed under α-η are of general use. There exist well known languages which have constructs to support the expression of some of the above relationships. For example Smalltalk-80 has browsers to represent classes and instances of those classes (case γ), the Smalltalk object and its methods are an external, abstract view (α) of the implementation, which in turn may be considered as an aggregation (β) since the methods usually use methods of other objects to compute their response. Class-subclass (η) relationship is also a feature of the language. Besides the similarities of course there are a number of differences in the way of inheritance, the explicit distinction of later and former states of the object, the possibility of parallel versions, the way in which the behaviour of an object (to represent a model) is defined — to name only a few. The relationships explained are well expressed in a declarative way, and intend to be a representative taxonomy of relationships which can be defined among design object models. On the other hand we are aware of the fact, that some relationships (especially those which connect different types of models about the same design object) may be better given a procedural semantics. It may be much easier to represent the way how two models of a manufacturing plant relate to each other, e.g.

how

- the simulation model of a plant's transportation system relates to
- the model of it for stock level control including its financial consequences.

One use of design scenarios is to maintain such relationships.

4.2 Dynamic evolutional relationships between models

Suppose that worlds and objects encapsulated by those worlds are used to create design object models. Since a purely definitional representation cannot capture the behaviour of the models during their evolution into completed models, rule based scenarios, working on worlds are used to record that aspect.

The evolution of such worlds is due to the assertion of new facts in those worlds. A design scenario may assert new facts related to a model because of several reasons:

• *Developing one model:* We simply add new facts to those currently known to be true in the world which represents the model. This action conceptually creates a development of the given world and no concurrent world is coming into existence. In case of non-monotonic development housekeeping of antecendent nonmonotonic assumptions makes future backtracking and usual truth maintenance services possible.

• *Shifting from one model to another one:* A model of a design object (e.g. the simulation model of a plant's transportation system) is queried by a design scenario and the result is used to assert facts in another model (e.g. the plant's model for stock level control with its financial consequences). The same kind of function is required when merging or comparing two different models. The world of the design scenario is a model of the design situation in which there is a need to transport consequences of one model into another one. The world on this level looks at the involved two models from their outside, i.e. as if they were objects.

Another typical case may be when we have a functional model of a machine and a timing diagram (model) of its functioning. Suppose the task is to create a kinematic model. The design scenario in this case has to consult the functional and the timing models as two separate worlds and then build the kinematic model in its own world. In addition to the two mentioned model worlds, which this scenario consults as objects only, other "building blocks" for kinematic models of machines are used as well.

• *Explicit or implicit branching of concurrent models:* By asserting a parallel or-branch in a rule (implicitly) or, by asserting an appropriate built-in predicate (explicitly) the system can be forced to create a set of alternate concurrent worlds and continue the execution of the scenario on them. The effect will be, that the worlds develop in parallel and the scenario in question begins to run concurrently with itself without necessarily being aware of doing so.

From that time anything asserted in the system is labelled to be true in one of the respective concurrent versions which generated them. If the execution of scenarios in a given concurrent version generates a failure which backtracks to the point where the world has been branched, the version of the world is discarded, and execution stops on that branch. If the execution stops with a success, the version generated on this branch is preserved.

Scenarios which want to make use of the fact that parallel versions of the current world exist, might consult facts in the concurrent versions through addressing queries to

the clone manager rather than directly consulting the part of the database seen in their world.[5] The clone manager can provide world_names of the concurrent versions and answer queries addressed to them, but never assert anything in the concurrent worlds. The clone manager can be accessed through built_in predicates and -functions. This is a possibility for scenarios to look at how well their concurrents are doing.

• *Parallel work of monitoring and monitored scenarios:* A different type of design evolution is when the calling scenario does not pass control to the called one, just gives it a start and continues execution. Scenarios are started through asserting built-in predicates. The assertion practically always succeeds (assume the scenario to be started exists). In the simplest case the scenario which has given the start to the other one will be able to monitor the progress of it, given the right to stop, resume or discard the monitored one. The monitoring scenario can access the monitored scenario's world as an object, also they are entitled to consult objects which they both know.

For example the monitoring scenario's world is *pipe_material_design_situation* and looks at the *pipe* under design only as at an object, while the monitored scenario will enter the world of *pipe* to actually assert facts in that world. When the monitored scenario succeeds nothing happens, just the monitoring one stays alive. In the case when the monitored scenario fails all the facts asserted on its behalf are removed, except for the "last cry" and nothing more happens; the monitor stays alive as well. The "last cry" of the monitored scenario (i.e. what has to be asserted at success and what at failure in the parent scenario's world) can be a parameter of the built-in predicate which started it. The monitor can silently stop by declaring success and let its monitored child stay alive. Failure declaration of the monitor means killing the monitored scenario as well.[5] Since the monitor may have been started by an even higher level monitor, or just been called by a higher level scenario, the rule is, that success can be reported to its caller only if its monitored scenarios have terminated. Note that terminated may also mean "failed": it is the monitor's responsibility to judge if that means its own failure as well or not.

• *Combination of concurrency and monitoring:* We are currently investigating the effects of combining concurrency and monitoring, to give a useful definition of accessibility between multiple worlds, which will account for the validity of formulae with simple modalities #P (possible) and #N (necessary).

4.3 Overall architecture: layering IIICAD knowledge bases

The knowledge base of an actual IIICAD system will have to incorporate different bodies of knowledge. The writer of specific design scenarios (e.g. an engineering office) should be able to rely on predefined, general purpose knowledge packages encapsulated in objects and/or scenarios. The engineering activity which will actually create an intelligent CAD system for a class of applications has to be supported in a number of ways. A similar situation prevails among Smalltalk users, who can exploit the power of the system only after having spent a substantial period of studyiing the available tools. We think that in addition to creating IDDL as an environment we shall also have to provide a knowledge representation paradigm. Future users of the environment who create CAD systems for end-users need to uniformly exploit the possibilities of the language and understand the full use of the encapsulated knowledge which is available in a given

general environment.

Some of the knowledge may even be coded in entirely different ways from IDDL, in which case such external tools must be packaged into virtual objects and their handling into virtual scenarios. Such virtual entities create the interface between IDDL based and external tools and can serve the IDDL-based integration of a number of readily available CAD systems. Such systems can be different analysis tools, expert systems with specialised knowledge etc.

The packages of knowledge themselves can be hierarchically ordered into a layered structure as we earlier suggested[7] . We envisage a collection of sample objects and scenarios which could be used as examples for creating concrete CAD systems. Such sample objects and scenarios could describe how typical design scenarios are constructed, the way goal-directed planning of design activities can be encoded and representational clichés for dealing with multiple models of the same design object. Also, in lower level layers of the predefined knowledge tools one could provide definitions for more elementary and ready to use concepts of geometry, mechanics etc.

We intend to look at the user interface from within the language environment as at ordinary objects and corresponding scenarios. In the same manner as external tools can be integrated through virtual object- and scenario interfaces, we think that the user interface of IDDL-based IIICAD systems should connect to the design scenarios using the same principle.

5. Conclusions

We have presented the current state of the IDDL language design effort which concentrates on the integration of object oriented and logic programming paradigms into a knowledge representation language especially suitable for implementing intelligent CAD systems.

We have found that there are a number of trade-offs in the implementation of the original requirements. The flexible inheritance methods and the expressive power of the language and the essentially higher order effect of encapsulating knowledge into objects complicates the theorem proving task in the static part of the language. This made us restrict the type of function definitions and occurence of function expressions as much as possible.

Present advances in deductive databases suggested us that the logic part of IDDL may rely on those techniques. It remains to be seen by subsequent research how the present restrictions imposed onto the language and its logic can be either removed or circumvented by the application of new database update techniques.

The dynamic part of IDDL is mainly concerned with two problems: how can be provided an appropriate control for the forward resoning process used in scenarios. As many have argued one of the impediments in expressing design knowledge in any computer language is the monocultural nature of the languages available [9] . A pluralistic, permissive but still not borderless approach may be built on the analogy of defined inheritance vs. rigid wired-in taxonimy of inheritance relationships. This amounts to defining a set of control primitives which can be imposed onto the forward reasoning process as a rule selection control.

The already well known requirement of maintaining multiple worlds influenced the IDDL design into defining specific constructs for starting parallel worlds. As in

assumption based truth maintenance [12] , default logic is only one of the reasons why assumptions are created. it is the problem solvers duty to judge whether and when to create an assumption — on account of the utilisation of a default fact in the reasoning process (i.e. at the left hand side of a rule) or because of an explicit parallel world branch.

6. Acknowledgements

Sections 1.,2. and 4. has mainly been the work of P. Bernus and P.J.W. ten Hagen. Section 3. has mainly been the work of P. Veerkamp and V. Akman. The Bart Veth research group is lead by P.J.W. ten Hagen. We are all thankful to T. Tomiyama who started the research and to our visiting researchers F. Arbab and Zs. Ruttkay who contributed with their criticisms to the results presented here.

References

1. *Simula 67.*
2. *Non-monotonic Reasoning Workshop,* Preprints. AAAI Workshop October 17-19, Mohonk Mountain House, New Paltz, NY (1984).
3. *On Knowledge Base Management Systems — Integrating Artificial Intelligence and Database Technologies,* Springer-Verlag, New York (1986).
4. *Logic Programming — Functions, Relations, and Equations,* eds. D. De Groot and G. Lindstrom, Prentice Hall, Englewood Cliffs, N.J. (1986).
5. ARBAB, F., "A Paradigm for Intelligent CAD," pp. 20-39 in *Intelligent CAD Systems 1 — Theoretical and Methodological Aspects,* ed. P.J.W ten Hagen and T. Tomiyama (Eds), Springer-Verlag, Berlin (1987).
6. BARBUTI, R., BELLIA, M., AND LEVI, G., "LEAF: A Language Which Integrates Logic, Equations and Functions," pp. 201-238 in *Logic Programming — Functions, Relations, and Equations,* ed. eds. D. De Groot and G. Lindstrom, Prentice Hall, Englewood Cliffs, N.J. (1986).
7. BERNUS, P. AND LETRAY, Z., "Intelligent Systems Interconnection — What Should Come after Open Systems Interconnection?," pp. 44-56 in *Intelligent CAD Systems 1 — Theoretical and Methodological Aspects,* ed. P.J.W ten Hagen and T. Tomiyama (Eds), Springer-Verlag, Berlin (1987).
8. BIJL, A., "Strategies for CAD," pp. 2-19 in *Intelligent CAD Systems 1 — Theoretical and Methodological Aspects,* ed. P.J.W ten Hagen and T. Tomiyama (Eds), Springer-Verlag, Berlin (1987).
9. BYLANDER, T. AND CHANDRASEKARAN, B., "Generic Tasks for Knowledge-Based Reasoning: the Right Level of Abstraction for Knowledge Acquisition," *Int. J. of Man-Machine Studies*(26), pp. 231-244 (1987).
10. CLOCKSIN, W.F. AND MELLISH, C. S., *Programming in Prolog,* Springer-Verlag, Berlin (1981).
11. DAVIS, R. AND KING, J., "An Overview of Production Systems," pp. 300-332 in *Machine Intelligence 8,* ed. Donald Michie, Ellis Horwood Ltd., Chichester (1977).
12. DE KLEER, J., "An Assumption Based TMS," *Artificial Intelligence* **28**, pp. 127-162 (1986).
13. FIKES, R. AND KEHLER, T., "The Role of Frame-based Representation in Reasoning," *CACM* **28**(9), pp. 904-920 (September 1985).
14. FOX, M.S., "On Inheritance in Knowledge Representation," pp. 282-284. in *Proc. IJCAI,* Tokyo (1979).
15. GOLDBERG, A. AND ROBSON, D., *Smalltalk-80: The Language and its implementation,* Addison-Wesley, Reading, MA (1983).
16. HAYES, P.J., *In Defense of Logic,* Proc. IJCAI-77, Cambridge, MA (1977).
17. HINTIKKA, J., "Semantics for propositional attitudes," pp. 145-167 in *Reference and Modality,* ed. L. Linsky, Oxford Univ. Press., London (1971).

18. HUGHES, G.E. AND CRESSWELL, H.J., *An Introduction to Modal Logic*, Methuen& Co. Ltd. (1968). (reprinted with corrections 1972)

19. KONOLIGE, K., *A Deduction Model of Belief*, Morgan Kaufman,Inc., Los Altos,CA (1986).

20. KRIPKE, S.A., "Semantical Considerations on Modal Logic," *Acta Philosophica Fennica* **16**, pp. 83-94 (1963).

21. LANSDOWN, JOHN, "Graphics, Design and Artificial Intelligence," in *Theoretical Foundations of Computer Graphics and CAD, NATO ASI Series*, ed. R.A. Earnshaw, Springer-Verlag, Berlin (1988). To appear

22. LLOYD, J.W. AND TOPOR, R.W., "A Basis for Deductive Database Systems," *The Journal of Logic Programming* **2**(2), pp. 93-109 (1985).

23. LLOYD, J.W., *Foundations of Logic Programming*, Springer-Verlag, Berlin (1987). Second, Extended Edition

24. MCCARTHY, J., "Circumscription - a Form of Non-monotonic Reasoning," *Artificial Intelligence* **13**, pp. 27-39 (1980).

25. MILLER, D., "A Theory of Modules for Logic Programming," pp. 106-114 in *Proceedings of the 1986 Symposium on Logic Programming*, IEEE Computer Society Press, New York, N.Y. (1986).

26. MOORE, R. C., *A Formal Theory of Knowledge and Action*, Tech. Note. #320, SRI International (February 1984).

27. NADATUR, GOPALAN, "Higher-order Logic Programming," pp. 448-462 in *Proceedings of the Third Int. Logic Programming Conf.*, London (1986).

28. PATERSON, M.S. AND WEGMAN, M.N., "Linear Unification," *Journal of Computer and System Sciences* **16**(2), pp. 158-167 (1978).

29. REITER, R., "A Logic for Default Reasoning," *Artificial Intelligence* **13**, pp. 81-132 (1980).

30. ROBINSON, J.A. AND SILBERT, E.E., "LOGLISP: Motivation, Design and Implementation," pp. 299-314 in *Logic Programming* , ed. eds. K.L. Clark and S.-A. Tärnlund, Academic Press, London (1982).

31. SUBRAMANYAM, P.A. AND YOU, JIA-HUAI, "FUNLOG: a Computational Model Integrating Logic Programming and Functional Programming," pp. 157-198 in *Logic Programming — Functions, Relations, and Equations*, ed. eds. D. De Groot and G. Lindstrom, Prentice Hall, Englewood Cliffs, N.J. (1986).

32. TOMIYAMA, T. AND YOSHIKAWA, H., "Extended General Design Theory," pp. 95-130 in *Design Theory for CAD*, eds. H. Yoshikawa and E.A.Warman, North Holland, Amsterdam (1986).

33. TOMIYAMA, T. AND TEN HAGEN, P.J.W., "Organisation of Design Knowledge in an Intelligent CAD Environment," in *Expert Systems in Computer Aided Design*, ed. J. Gero, to be published by North Holland, Amsterdam (1987).

34. TOMIYAMA, T. AND TEN HAGEN, P.J.W., "Representing Knowledge in Two Distinct Descriptions: Extensional vs. Intensional," CWI Report, Centre for Mathematics and Computer Science, Amsterdam (1987).

35. TROELSTRA, A. S., *Mathematical Investigations of Intuitionistic Logic*, Lecture Notes in Mathematics 344, Springer-Verlag, Berlin (1973).

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

Appendix: IDDL syntax[7]

List:

<list> ::= <list_of_forms> | <list_of_terms>
<list_of_forms> ::= <nil> | .(<form>,<form>)
<list_of_terms> ::= <nil> | .(<term>,<term>)
<form> ::= <term> | <clause> | <list>

Constant symbol:

<constant_symbol> ::= <constant_predicate_symbol> | <function_symbol> | <constant>

Constant:

<constant> ::= <built_in_constant> | <user_defined_constant> | <skolem_constant>
<built_in_constant> ::= <integer> | <real>| <string> | integer | real | ....
<integer> ::= <unsigned_integer> | {<sign>}<unsigned_integer>
<unsigned_integer> ::= <digit> | <digit><unsigned_integer>
<digit> ::= 1 |2 |3 |4 |5 |6 |7 |8 |9 |0
<sign ::= + | -
<real> ::= {<sign>} {<unsigned_integer>}.<unsigned_integer>{E<integer>}
<user_defined_constant> ::= <lower_case_letter>{<characters>}
<skolem_constant> ::= _
<string> ::= "<characters>" [8]
<characters> ::= <character> | <character><characters>
<character> ::= <lower_case_letter> | <upper_case_letter> | <digit> | <_>
<lower_case_letter> ::= a | b | ...
<upper_case_letter> ::= A | B | ...

Term:

<term> ::= <atomic_term> | <nil> | <compound_term> | <function_expression> | <variable>
<atomic_term> ::= <constant>
<nil> ::= .() | "{" "}"
<compound_term> ::= <function_symbol>(<n_tuple_of_arguments>) | <list_of_terms>
<function_symbol> ::= <built_in_function_symbol> | <user_defined_function_symbol>

<built_in_function_symbol> ::= *+ | × | / | sin | cos | atan | call | setof | car | cdr | eval | if_else | while | case | pi | hour | minute | second | year | month | day | ...* [9]

<user_defined_function_symbol>::= <lower_case_letter>{<characters>}
<n_tuple_of_arguments> ::= <argument>{[,<argument>]}
<argument> ::= <term> | <atomic_predicate>
<variable> ::= <upper_case_letter>{<characters>}

---

7 Metalanguage conventions: [] means once or more times, { } means optional, {[]} means (consequently) zero or more times, while "[", "]", "{", "}", "<", ">" mean [,],{,},<,> respectively. In addition < > / and ::= are the usual metasymbols.
For notational conveniences introduced see the back of the appendix.

8 equivalent to the list of ASCII codes i.e.*"abc"* is the same as *{49,50,51}*

9 instead of including "." into the list of built in functions, *<list_of_terms>* has been explicitly said to be a *<compound_term>*. So the syntactic forms *.(<>,<>)* or *{<>,<>}* do not automatically imply that we have to do with a *<term>*, it may well be a list of clauses not to be treated as a list of terms.

Function expression:

<function_expression> ::= *<function_symbol> "["{<n_tuple_of_arguments>}"]" /*
<remote_function_expression>

<remote_function_expression> ::= term> :: <function_symbol> "["{<n_tuple_of_arguments>}"]"

Predicate:

<predicate> ::= *<atomic_predicate> / NOT <atomic_predicate> /*
#D <atomic_predicate> / "default" operator
% <atomic_predicate> "unknown" operator

<atomic_predicate> ::= <predicate_symbol> {(<n_tuple_of_arguments>)} / <function_expression>[10]
<predicate_symbol> ::= <constant_predicate_symbol>
<constant_predicate_symbol> ::= <built_in_predicate_symbol> / <user_defined_predicate_symbol>

<built_in_predicate_symbol> ::= *== / atom / = / < / > / setof / call / PROGRAM /*
DEFINE_OBJECTS / DEFINE_OBJECT / OBJECTS / OBJECT /
FUNCTIONS / FUNCTION / FACTS / CONSTRAINTS /
DEFINE_SCENARIOS / DEFINE_SCENARIO / RULES / RULE /
succeed / fail / ....

<user_defined_predicate_symbol> ::= <lower_case_letter>{<characters>}

Formula:

<formula> ::= *<predicate> / (<formula>) / <formula> <serial_connective> <formula> /*
TRUE / FALSE / NOT <formula> /
#P <predicate> / "possibility" operator
#N <predicate> / "necessity" operator
<quantifier> ({<variable>{[,<variable>]}}[12]*) <formula> /*

<connective> ::=& / <parallel_connective> / <temporal_connective>
<parallel_connective> ::= ∧ / ∨ [13] *<temporal_connective> ::= BEFORE / AFTER "temporal" connectives*
<quantifier> ::= ∀ / ∃[14]

Clause:

<clause> ::= <HORN_clause> / <positive_program_clause> / <program_clause> /

<HORN_clause> ::= *<atomic_predicate>. /*
<atomic_predicate> ← <predicate> {[& <predicate>]}.

<positive_program_clause> ::= <HORN_Clause> / <atomic_predicate> ← <formula>.
<program_clause> ::= <positive_program_clause> / <predicate> ← <formula>.
<parallel_clause> ::= <clause> / <clause> <connective> <clause>

---

10 Higher order effects are covered by object declarations and functions that evaluate to atomic predicates.

11 We capitalise higher order predicates

12 an empty variable list means quantifying over all the variables which appear in the scope of the quantifier.

13 If ∧ and ∨ are not available in the basic character set, the words ***and*** (resp. ***or***) can be used instead.

14 If ∀ and ∃ are not available in the basic character set, ***#A*** (resp. ***#E***) can be used instead.

Rule:

<rule> ::= ***RULE (<formula>,<parallel_clause>) / RULE (<formula>,<parallel_clause_list>) /***
IF <formula> THEN <parallel_clause> / IF <formula> THEN <parallel_clause_list> */= infix notation*

Object:

<object_definition> ::= DEFINE_OBJECT(<object_name>,
{,OBJECTS(<object_declaration_list>)}
{,FUNCTIONS(<function_definition_list>)}
{,FACTS(<clause_list>)}
{,CONSTRAINTS(<clause_list>)})

<object_name> ::= <constant_symbol>
<function_definition> ::=
FUNCTION(<function_symbol>"["{<n_tuple_of_arguments>}"]",[15]<function_expression>) /
EXPORTED_FUNCTION(<function_symbol>"["{<n_tuple_of_arguments>}"]",[15]<function_expression>)
<object_declaration> ::= <static_object_declaration> / <dynamic_object_declaration>
<static_object_declaration> ::= OBJECT (<object_name>,<term>[16])
<dynamic_object_declaration> ::= OBJECT (<variable>,<term>,<predicate>)[16]

Scenario:

<scenario_definition> ::= DEFINE_SCENARIO(<scenario_name> (<n_tuple_of_arguments>)
{,OBJECTS(<dynamic_object_declaration_list>)}
{,FUNCTIONS(<function_definition_list>)}
{,RULES(<rule_list>)})

<scenario_name> ::= <lower_case_constant>

IDDL program:

<IDDL_program> ::= PROGRAM (DEFINE_OBJECTS(<object_definition_list>),
DEFINE_SCENARIOS(<scenario_definition_list>))

Notational conveniences:

List notation:
{<term>,<term>} is syntactically equivalent to ***.(<term>,<term>)*** and ***{<form>,<form>}*** is syntactically equivalent to ***.(<form>,<form>)***. ***{a,{b,{c,d}}}*** can be written as ***{a,b,c,d}***.

Meta convention:
for brevity the meta-names ***<xxx_list>*** mean ***{}***, ***{<xxx>}***, ***{<xxx>, <xxx>}*** etc. — i.e. a list which contains zero or more elements of ***<xxx>***. Thus we could say ***<term_list>*** instead of defining a ***<list_of_terms>***.

Clause notation:
"***<xxx>.***" is syntactically equivalent to "***∀()(<xxx>)***" — whatever ***<xxx>*** may be. A clause is therefore a claused formula.

Object definition
object...end object; stands for ***DEFINE_OBJECT(...)***.

Function definition

---

[15] no function expressions are allowed in the formal arguments, only ***<atomic_terms>***, ***<compound_terms>*** and ***<variables>***

[16] where ***<term>*** should be evaluable to an object or to a list of objects (i.e. a rigid designator of those objects) and ***<predicate>*** qualifies the ***<variable>***.

functions *term*$_{11}$ = *term*$_{12}$; *term*$_{21}$ = *term*$_{22}$;....; is equivalent to the function definitions

FUNCTIONS(***FUNCTION***(*term*$_{11}$,*term*$_{12}$),
FUNCTION(*term*$_{21}$,*term*$_{22}$),...)

Similar form exists for exported function definitions.

Object declaration in scenarios
objects *predicate*$_1$; *predicate*$_2$; *predicate*$_3$;...; stands for

OBJECTS(***OBJECT***(*var*$_1$,{},*predicate*$_1$),
OBJECT(*var*$_2$,{},*predicate*$_2$),
OBJECT(*var*$_3$,{},*predicate*$_3$),...)

where var_i is the free variable in $predicate_i$ and the second argument is (at the start) empty. The second argument is going to hold the instantiations of var_i.

Object declaration in objects
attributes *predicate*$_1$; *predicate*$_2$; *predicate*$_3$;...; has a meaning similar to the object declarations used in scenarios, but implicitly generates appropriate function definitions as well — as described in Section 2.2. The notation is often used in the definition of objects which stand for classes.

Scenarios
scenario ··· ***end scenario;*** stands for DEFINE_SCENARIO(···)

Rules
begin *rule*$_1$; *rule*$_2$; *rule*$_3$;...; stands for

RULES(***RULE***(*rule*$_1$),
RULE(*rule*$_2$),
RULE(*rule*$_3$),...)

Paper Session:

Interfaces

Multi-media Presentation in CAD Systems

Zs. Ruttkay

Multi-media presentation in CAD systems

Zsófia Ruttkay

Computer and Automation Institute, Hungarian Academy of Sciences,
1502 Budapest, P. O. B. 63, Hungary

Abstract: The paper discusses how knowledge-engineering can be used to define a user interface supporting multi-media communication with a CAD system. Object-oriented representation of the lexical and syntactic constituents, and rule-based production systems for the control of the specific user interfaces using one media will be described. The co-ordination of the usage of the different media is provided by a blackboard mechanism. The functions provided by the user interface are also dealt with.
Keywords: User interface, presentation, multi-media communication, object-oriented programming, rule-based programming.

Introduction

In research concerning user interfaces the emphasis has been on how to exploit physical devices and interaction techniques provided by them, focussing on the human factors aspect and implementational issues. Research devoted to the role of AI-based techniques has addressed first of all the questions of lexical and syntactic level of user interfaces.

Object-oriented programming has been used with success to define the user interface in terms of the visual appearance and functional description of its constituents: windows, menus, icons [4]. The content of the window was either generated by the application and could not be reached via the user interface [16], or was represented as objects whith more ore less attributes shared by the user interface and the application [2, 7, 15]. There were short communication cycles supported, the application's task was to execute actions initiated by the user, without checking its semantic correctness.

The use of production systems was proposed to define the dialogue component of interactive systems [5, 11], and experiments have been made since then [8, 12].

In our approach the main point of interest is the functionality of the user interface: its adequacy for the information to be exchanged and for the expectations of the user, both concerning utilities supporting humans' typical errors as well as personal features and a profession's practices of proved value in communication and presentation.

These questions are of basic importance in the case of CAD: the type (geometric and non-geometric), the quality (undefined, partially or fully defined) and the quantity of the data to be presented – just like the set of feasible further design steps – change during the design process [9, 14, 17].

There are well-established traditional ways – textual descriptions, sketches with symbols for parts, annotated drawings – to present different aspects of a design, using different media. These traditional presentations should be supported when using a CAD systems, and could be used not only to give information about a complete design, but to change and improve the design interactively in course of the design process[10, 18].

In our paper we define a user interface which is to meet some basic specific requirements of CAD systems. We give a conceptual basis for the user interface and show how it can be implemented using knowledge engineering tools: object-oriented representation of lexical elements and syntax, and a rule-based blackboard mechanism to control the course of communication. The correspondence of the presentations and the object being designed is maintained by the user interface. The use of different media for both input and output and dynamic adaptation of the interface is supported.

The paper is structured in the following way: In Chapter 1. the CAD system of which the interface we are going to deal with is presented. In Chapter 2. the architecture and implementational issues of the user interface is discussed. In Chapter 3. the novel functions supported – concerning communication actions as well as auxiliary services – are dealt with. Finally, in Chapter 4. some issues which we have not addressed are mentioned as further directions for research.

1. The CAD system

1.1. Why variational design?

Our approach was to build a user interface to such a specific CAD system which exhibits basic and common requirements of CAD, concerning

communication with the user.

We think that the interface issues are more involved in the case of apprentice type systems than autonomous systems [1]: the communication between the system and the user is user-driven or mixed-initiative, the user has much freedom in choosing what to do next, how to improve the design.

Throughout the paper, we use the notion design for the object being designed, design system (DS) for a system to perform/support the design process, and user interface(UI) for a system enabling the human designer to use a given design system.

We have chosen a variational design system as a design system to build a user interface for. Variational design – though often stated to be the less demanding type of design – operates with generic concepts of design: feasible design steps, consistency of the design, re-design [13]. On the other hand, as there is much left for the user in choosing the next action to be performed, much interaction is required, and there are long, embedded communication cycles to be handled by the user interface.

From the user interface's point of view, the design system responds to a user's action. We are not interested in why and how the design system derived its response(s), but in the consequences of the responses to the flow of the dialogue: how and when to display a response, when the user is allowed/forced to make a next step, how to inform him about his possible next inputs. In the rest of this chapter we discuss the design system from this point of view.

1.2. The representation of a design

There is a conceptual representation of the classes of design objects (e.g. parts) and the concepts (e.g. dimension, price) used in defining and evaluating designs. The classes are described by attributes and constraints attached to them and relations among them. A member of a class is represented as a 'filled in' class description. Partially defined instances – i.e. class descriptions with unbound attributes – represent subsets of the given class. Subclasses are defined according to the taxonomy of the object being designed. Inheritance of attributes, with default or compulsory values is supported. The value of an attribute can be an object itself, with its own attributes. The structure of the representation of objects provides a natural

hierarcy of the attributes: attributes of an object are refined attributes of the attribute which the object is the value of, see Fig. 1.We shall use the refinement concept for both the 'conceptual zooming' on parts and the focussing on specific aspects.

gear_pair_with_parallel_shafts
driving_gear:
*is_a:***gear**
driving_shaft:
*is_a:***shaft**
driven_shaft:
*is_a:***shaft**
driven_gear:
is_a: **gear**
ratio_of_teeth:
range:[1, 5]
driving_shaft:
*is_a:***shaft**
driven_shaft:
*is_a:***shaft**
distance_of_shafts:
*range:***standard_dist**
weight:
*derived:***summa(**weight of driving_gear, weight of driven_gear, weight of driving_shaft, weight of driven_shaft)

gear
number_of_teeth:
pitch_radius:
angle_of_teeth:
range:[0,15]
tooth_form:
is_a: **tooth**
tooth_strength:
weight:

Fig. 1.
Refined and derived design attributes

An attribute is derived from some other attributes, if its value is computed from the values of the given attributes by a derivation function, see Fig.1. A derived attribute can be computed only in one way. The derivation graph – which is supposed to be acyclic – assigns to each attribute the ones which are derived from the given and some other attributes. Derived attributes can be given values only by the application.

Constraints refer to an attribute value: the cardinality, the class(es), the range or set of the allowed values. Relations have at least two attributes as variables. The attributes in a relation can be those of the same or different classes.

The taxonomy of the design objects with the constraints and relations form the knowledge concerning the design objects. A design – i.e. a member of a class of the taxonomy – is consistent, if all the constraints and relations hold.

1.3. The design process

In course of the design process the user, aided by the system, instantiates the design, by refining the design according the class hierarchy, and defining values for member-specific attributes. The design process is constrained by prescribed precedences for the attributes: certain attributes cannot be given values before others. The precedences are given by an acyclic precedency-graph. The user initiates a design action, which is then executed by the system, with the implied further design actions.

The following design actions can be executed:

- giving value to a design attribute
- deleting an attribute value.

After giving value to an attribute, the following checks are performed:

- constraint-checks;
- relation-checks for all the relations which refer to the attribute and which can be evaluated;

The following value-propagations are fulfilled by the DS:

- whenever an attribute is given a value, all its derived attributes with a derivation function which can be evaluated, are bound;
- whenever an attribute's value is deleted, all of its derived attributes' values are also deleted.

The user can initiate any design action, then it is the DS's turn to check it and accomplish value-propagations. The DS performs related design actions (according to the prescribed precedences first and then the derivations) in a depth-first way. The DS reponse to a design action is one of the followings:

- further attribute has to be specified by the user;
- error, indicating the constraint which has been violated;
- conflict, indicating the relation not holding;
- a design action has been performed successfully.

The DS reports also the design actions which have been executed by the DS, and were not initiated by the user. The user is expected to react to the first three cases. His actions can be one of the followings:

- giving value to an attribute;
- quitting the design process;
- cancelling his previous design action which resulted the error or conflict reported;

- indicating attributes which he is going to correct in order to overcome the error or conflict situation.

A communication cycle is started by a design action initiated by the user, which is followed by a sequence of actions by the system and reactions by the user. The possible actions by the DS and the user listed above are the communication actions. A communication cycle is terminated either by the DS, indicating that the user-initiated design action has been executed successfully or by the user's quit action. Inbetween, if a value has been asked for by the DS, the user is forced to give it or to quit. If an error or conflict has been reported, usually there are more than one ways to eliminate it, it is up to the user, what he does.

The design is being changed during a communication cycle, and if it is terminated by the DS, then the result is a consistent design. Otherwise, the design is not consistent. The DS preserves the last consistent design version, that is the design before the start of the communication cycle.

2. The user interface

The user interface has double responsibility:

- to mediate between the user and the DS: to transform the user's input(s) into the appropriate communication actions, and to generate the necessary output action(s) to inform the user according to the system's actions and to indicate what next input actions he can do;
- to control the communication: in a given stage what next actions are allowed/expected by the user and the DS.

The user interface provides more flexible communication than a static mapping of i/o actions and communication actions, 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 DS's need. The user interface supports the communication of the user and the design system on three levels:

- on the device level, the i/o actions are executed;
- on the presentation level, the mapping of i/o actions and communication messages is fulfilled;
- on the communication level, the communication messages are collected, interpreted in the context of the previous

communication messages and the state of the presentations (discussed later), and forwarded to the DS or to the presentation level.

We are not dealing with the device level. We refer to windows and the common input functions (locate, choose, pick, type) and suppose, that the mapping of the physical inputs and the input functions and the accomplishment of outputs (redrawing parts of the screen, cursor-echoing) is provided by low-level utilities of a multi-window environment.

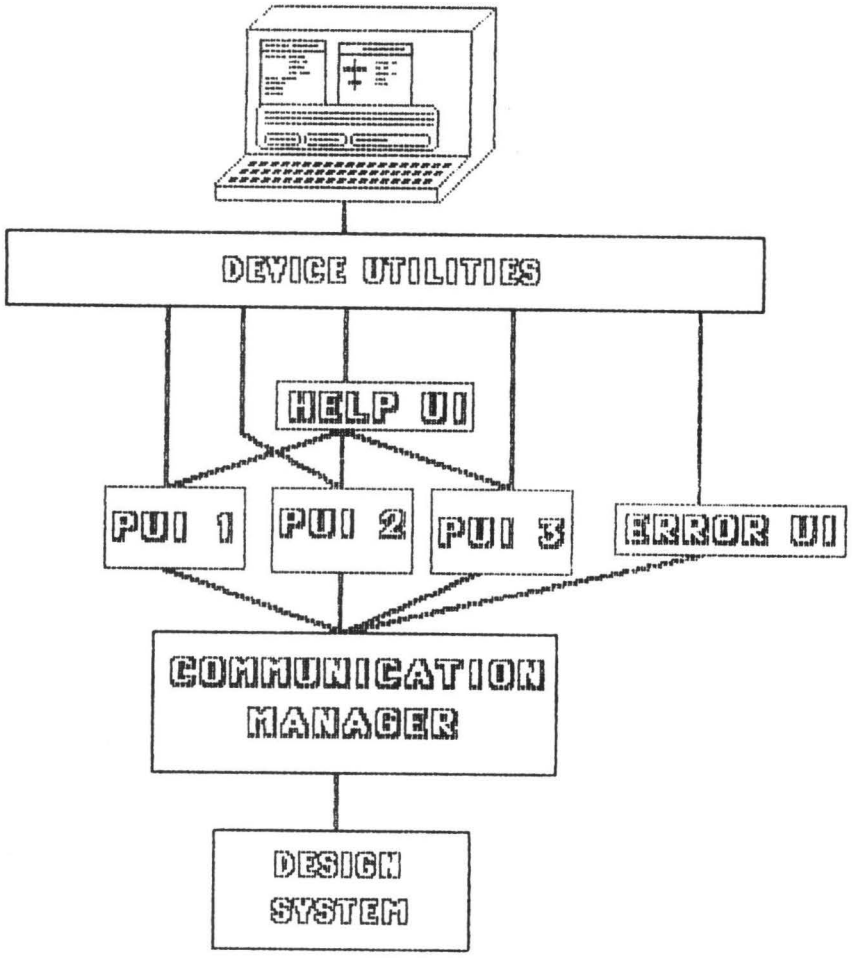


Fig.2.
Components of the user interface

The basic components of the user interface are as follows (see Fig. 2):

- the specific presentation user interfaces which support different presentations of the design and service interfaces used in course of

the design process (error, help). These interfaces rely upon local knowledge to manage interaction via the given interface only;
- the communication manager to co-ordinate the usage of the different presentation and service user interfaces and the design system, relying upon rules concerning the communication and the usage of the different presentations.

2.1. Presentation user interfaces

A presentation user interface (PUI) is used to exhibit and interactively edit a given presentation of a design. The user communicates with a PUI via a window on the screen. Different presentations of a design are managed by different PUIs. Actions common in all presentation user interfaces are also supported: open, close, alternative presentations, refined presentations, screen only/presentation only/design modes, cancel last action, help, quit (discussed later).

There can be several presentations open at a given moment, but there is always only one presentation active, the other presentations are passive. Input/output actions can be performed only via an active presentation, but the user has freedom in choosing the presentation from a set of candidates, i. e. from the ones not having been locked by the communication manager. On the other hand, for output it is the communication manager who decides which presentation should be activated next. In doing so, user-defined preferences and presentation-specific rules are taken into account.

A presentation user interface consists of the following components:
- the presentation state description (PSD) where the current state of the presentation of the design and of the presentation-specific and non-specific actions are given;
- the presentation working memory (PWM) for storing messages from the PSD, or from the device or communication level;
- the presentation rule memory (PRM) with the rules defining message generation and propagation.

Each PSD an instance of a presentation type. The instances of presentation-specific entities provide information about the design, while the ones common in all presentation types are used for actions concerning the communication and the usage of the presentations, see Fig. 3.

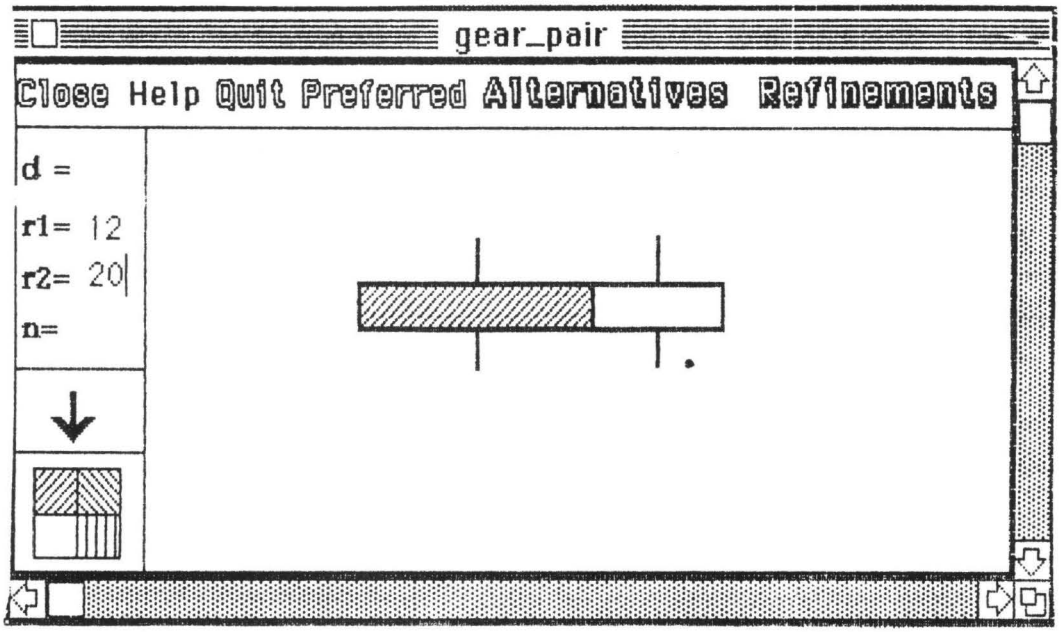


Fig.3.
A presentation window for a gear pair with parallel shafts

Further, by a presentation we mean the presentation-specific entities in the PSD. The entities common to all presentation types will be identified by referring to the presentation (e.g. status of a given presentation).

2.1.1. Presentations

A presentation provides information about certain attributes of the design, and offers dedicated editing facilities. A presentation is an instance of a presentation type. A presentation type defines the entity types which could be/should be used in the presentation of the design. The set of possible editing actions, the conditions when they can be performed and side-effects of the actions are also given by the presentation type.

Entity types are defined by attributes, with constraints and relations attached to them. Compound entity types with attributes, which have entities as values, can also be defined.

The changes in the visual appearance of an entity instance are caused by either the echoing of an editing action (chosen, last input) or changes in the design. The echoes are defined in an entity-aspecific way (chosen entity and the last input should be highlighted for all entities, an input error should be indicated by an error signal), while the changes in the visual appearance of an entity reflecting changes of the design are entity-specific. In the first cases, attributes defining the visual appearance of the entity are related to

some of its other attributes only (e.g. status), while in the second case they are related to an attribute of the design. This relation can be a one-to-one mapping (one presentation entity attribute value – one design attribute value); or a one-to-many mapping (the same presentation entity attribute value for a set of values of a design attribute), see Fig. 4. A presentation entity attribute can be related to at most one design attribute.

symbol_for_gear_pair_with_parallel_shafts

d:
default:10
d=distance_of_shafts/2
r1:
default:5
r1=pitch_radius of driving_gear/2
r2:
default:5
r2=pitch_radius of driven_gear/2
n:
default:
range:[1,5]
n=ratio_of_teeth
angle_of_teeth:
range:{parallel, less_then_90, more_than_90}
picture:
is_a:**picture_of_symbol**
parameters: d, n, angle_of_teeth

Fig.4.
Presentation entity for gear_pair_with_parallel_shafts

The semantic correctness of a presentation is provided via the relations coupling presentation entity and design attributes: whenever a presentation entity attribute has been changed by the user, the design system is informed about the implied change in the design, and the change of a design attribute is reflected by the change of the related presentation entity attribute(s). The constraints attached to entity attributes can prescribe only the type and set of values for the given attribute.

Two presentation types are alternatives, if the set of design attributes they refer to is the same. For a presentation type, the alternative presentation types are given. Preferences of presentation types can also be declared. The preference relation is allowed to be partially defined, but in a transitive way.

A presentation is a refinement of another one, if the design attributes the first refers to are refinements of some of the design attributes referred by

the second.

A design can be presented in different details and by different refined and alternative presentations, see Fig. 5.

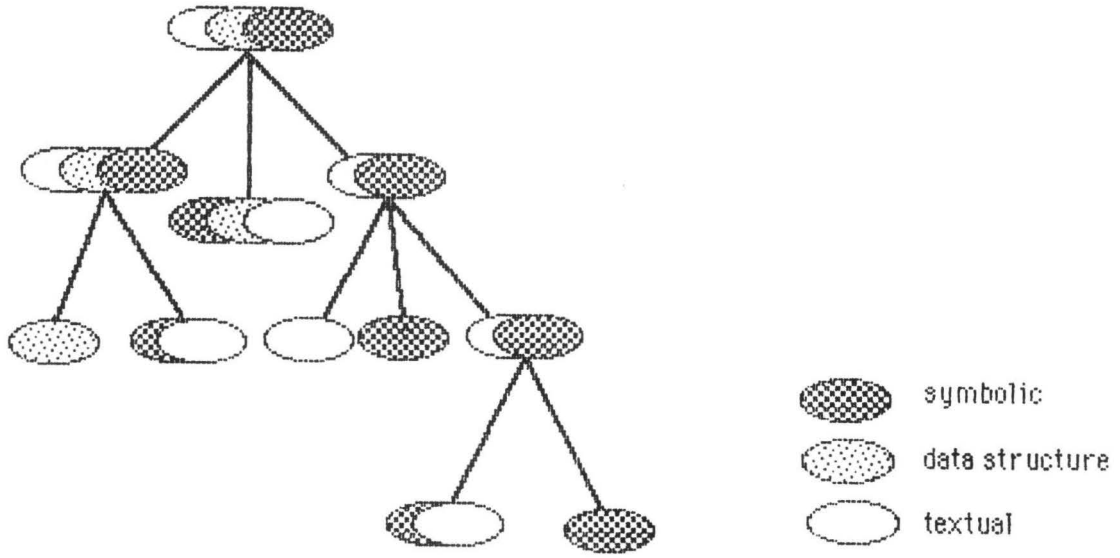


Fig. 5.
Refined and alternative presentations of a design

2.1.2. Control of the presentation user interfaces

The control of each presentation user interface is provided by a dedicated rule-based system, with a PWM and PRM. The mechanism is the same as for the communication manager, where it will be discussed in details. Here, we dwell on the control functions provided.

Whenever a presentation entity attribute concerning its visual appearance or related to a design attribute has been changed, or a constraint has been violated, a message is sent to the PWM. Changes in the presentation-aspecific attributes (state, preference) are also reported to the PWM.

Messages from the device level, from the entities in the PSD and from the communication level are processed according to the rule in PRM. The following functions are supported:

- The locate, choose, pick, type actions with references to window coordinates are resolved, and the action is forwarded to the

corresponding presentation entity instance; the messages from entities indicating changes in their visual appearance are forwarded to the device level.
- Messages from entities indicating changes in a design attribute are forwarded to the communication level, while a message indicating that a design attribute has been changed by the DS is forwarded to the related presentation entities.
- An error message resulting from violating a constraint of an entity's attribute is forwarded to the device level.
- Changes in the status of the presentation (open/closed; locked/unlocked), its usage (presentation only/design, preferred) or requests to open further presentation(s) (derived or alternative ones) are forwarded to the communication level, and the visual effects of such a change – initiated either by the user or by the communication manager – are forwarded to the device level.
- User's requests for quit and cancel are reported to the communication level, while for help is reported to the help user interface.

Messages to the communication level are forwarded immediately, while ones to the device level – excepting echoes – can be delayed by the state of the presentation (passive/active) or constraints concerning the frequency of window updates or priorities between the presentation and the device level.

2.2. Service interfaces

The user might need help about what input action he is allowed/forced to do in a given situation, especially in error situations. The user does not need help concerning the syntax of inputs: he is not offered input actions other than the ones which are syntactically correct, concerning the content of the PSD. This refers to presentation-aspecific functions (e.g. opening an alternative presentation) as well as ones specific to presentation entities (strings can be typed only to places corresponding to a string-type attribute of an entity instance).

In the case of semantic errors – indicated by the design system – the user needs support by explaining the error situation, offering possible alternatives to overcome it. In the case of uncertainties about how an input action changes the design and what values he can give to a presentation entity attribute, he needs textual help or an explanation of the allowed values. The error and help user interfaces provide such support.

2.2.1. The error user interface

The error user interface (EUI) receives messages from the communication manager indicating the error to be reported, and user responses from the device level. The error window contains, according to the message received from the communication manager:

- an error message indicating the type of the error: value out of range, or violated relation;
- the design attribute to whom the error message refers;
- the possible actions by the user: to quit the communication cycle, to cancel the last input, to choose some design attributes to be modified.

The first two actions can always be chosen, the last one only in the case when a relation does not hold, and all the variables in the relation are design attributes which have been given values in the current communication cycle.

The user has to choose one of the allowed actions to continue the communication. The action chosen by him is forwarded to the communication manager, where – with its side effects discussed later – it is executed.

2.2.2. The help user interface

The help user interface is used to:

- display textual description attached to a chosen presentation entity;
- display the entity type specification of a chosen entity instance;
- offer the choices for a chosen attribute of an entity.

The control is given to the help interface whenever requested by the user, after having chosen a presentation entity instance or an attribute in a presentation window. The appropriate help screen is generated by the help interface. The user can return to the presentation screen after having read the instruction or having chosen a value from the offered choices. In the last case, his choice is forwarded to the PWM.

2.3. The communication manager

The communication manager is responsible for mediating between the presentation user interfaces, the error user interface and the design system. It has the following tasks:

- to maintain the communication actions in context: to recognize the termination of a communication cycle; to generate further communication actions due to quit or error correction; to generate messages addressed to the DS, the PUIs or the EUI;
- to give the control to the DS, to a PUI or to the EUI;
- to maintain the correspondence of different presentations;
- to maintain the correspondence of presentations and the design.

The CM is being implemented in form of a rule-based production system: the flow of communication is defined by a static set of communication rules in the communication rule memory (CRM), which have on their left-hand side conditions concerning the content of the communication working memory (CWM), and prescribe actions to be performed on their right-hand side.

Records to the CWM are identified by time-stamp. The structure of records in the CWM are given in Fig.6.

record::=	**id** message
message::=	design_message | interface_message
design_message::=	message_from/to_PUI | message_from_DS | message_from_EUI
message_from/to_PUI::=	presentation **assign_value_to** attr value
message_from/to_PUI::=	presentation **remove_value** attr
message_from_DS::=	**value_assigned_to** attr value
message_from_DS::=	**value_removed** attr
message_from_DS::=	**ask_value** attr
message_from_DS::=	**error** attr constraint_descr
message_from_DS::=	**conflict** attr relation_descr list_of_attrs
message_from_EUI::=	**quit**
message_from_EUI::=	**cancel**
message_from_EUI::=	**modify_attributes** list_of_attrs
interface_message::=	presentation{**closed** | **passive** | **active**| **locked**}
interface_message::=	presentation **to_return_to**
interface_message::=	presentation **preferred**

Fig. 6.
The structure of records in the CWM

The CM is controlled by the content of the CWM, according to rules in the CRM. The syntax of rules in the CRM are given in Fig. 7. E.g. the following rules provides, that if the DS has asked for an attribute, then all the presentations which do not refer to the design attribute in question, will be locked.

ask_value P, not **influenced_pres**(Pres, Attr) , Pres **passive;**
Pres **locked, remove** Pres **passive, add** Pres **locked**.

ask_value Attr, not **influenced_pres**(Pres, Attr), Pres **active;**
Pres **locked, remove** Pres **active, add** Pres **locked**.

rule::= condition...condition; action...action.

condition ::= [not] record_mask | [not] pres_relation
action::= send_message | change_CWM

send_message::= presentation record_to_PWM
send_message::= **DS** design_message_to_DS
send_message::= **EUI** error_message_to_EUI

change_CWM::= {**add** | **remove** } record

pres_relation::= **influenced_pres** (presentation, attr)
pres_relation::= **refined**(presentation, presentation)
pres_relation::= **alternative**(presentation,presentation)
pres_relation::= **influenced_presentations** (list_of_pres, attr)
pres_relation::= **preferred_presentation** (list_of_pres,presentation)
pres_relation::= **alternative_presentations**(list_of_pres,presentation)

Fig.7.
Syntax of the rules in the CRM

On the left-hand side of a rule, there is at least one not negated record_mask condition. Variables can occur instead of constants, both in record_masks and the relations. The conditions in a rule are evaluated from left to right in the following way:

- if a record matches a record_mask, then the variables in the record_mask are bound to the value of the corresponding fields of

the record,
- if there are unbound variables in a relation, then they are bound in such a way that the relation holds,
- the variable-binding is propagated to all the conditions which have not been checked yet and to the right-hand side of the rule.

A list of records is called an instantiation of a rule, if the records match the required record_masks and the relations hold. Each rule can have different instantiations at a given state of CWM, and more than one rule can fire. The strategy of rule selection and firing is the one often applied in production systems: recency-based firing. [6] This firing mechanism provides, that whenever the content of the CWM has been changed –either by the CM itself or a PUI or the EUI – , the most recent rule instance will fire. This means that the recency (time-stamp) of the records in the CWM and the number of conditions in the rules will guide the rule firing: the rule-instance with the most recent record and with more records will fire.

3. Novel functions supported

3.1. Mixed initiative operation, flexible control

If not in a communication cycle, the user is allowed to do any input supported by the device level and the open PUIs.

An input action of the user can imply changes on the screen (e.g. changing the position of the presentation's window on the screen), changes in the UI (status/preference changes for PUIs), or changes in the design (design actions). In the latter case, the user has started a communication which then will be guided by the CM: the user can give further inputs only when requested by the DS. In such cases, the range of allowed next inputs is restricted, and the user is forced to give one of them. Quitting is allowed in all the cases.

In course of communication the different i/o actions should be interpreted by different PUIs or the EUI. The CM is responsible not only for mediating the messages between the DS and the PUIs, but to decide about when should the DS and the distinct PUIs and the EUI get the control. Rules prescribing the transfer of control always have priority to the others, as they are triggered by the most recent message in the WM. The condition part also can refer to the content of the recent message (e.g. Error, Done, Quit), to the state of the presentations, or to the time when the module was active for

the last time.

3.2. Alternative presentations for input

Before starting a communication cycle, the user is allowed to activate any open presentation, and to perform any input action allowed by the PSD. In the latter case, the presentation used by him will be preferred during the communication cycle, and after the termination of the communication cycle that presentation will be the active one.

Whenever during a communication cycle the DS is asking for an attribute value, all the presentations which cannot be used to define the required attribute will be locked, and the user can activate any of the unlocked and open ones and give the required input.

If all the unlocked presentations are closed, then the CM will open the preferred of the alternative presentations which can be used to define the required value.

If an input is requested by the DS, the all the unlocked presentations are informed about the request, and as a result, inputs specifying other attribute than the required one are not allowed.

3.3. Preferred presentations for output

When a communication cycle has terminated with success, the CWM may contain messages addressed to various presentations. It is the status of the distinct presentations and the preferences prescribed for them which define the next PUI to give the control to, after having forwarded messages to the PWM.

The presentation which was used by the user to start the communication cycle is always given the control first, unless there is no message addressed to it. The rest of the presentations which are not closed are given the control according to the preferences prescribed: preferred presentations are given the control first, then its alternatives. The sequence of presentations not related by preference prescriptions is not specified.

If a message is addressed to a closed presentation and at least one alternative of the addressed presentation is not closed, then the message is removed from the CWM, otherwise the presentation will be opened.

3.4. Correspondence of presentations and the design

In course of the communication a presentation on the screen – as the user can see it – is the presentation of a design. The presentation is used to indicate the user-specified changes to the last, consistent design version. All the design actions done by the DS are not forwarded to the presentation level before the communication cycle having been terminated successfully.

Here we don't discusse the issue of the impact of the limits of resources and the constraints rising from human's visual and cognitive capacities, but only the fact that outputs by the DS are responses to one of the user's previous design actions. It can turn out after several responses, that the user's input violates the consistency of the design. In such a case the user is either forced to redo one or more of his previous design actions or quit. In both cases, all those design actions and related outputs which followed the erroneous input are to be invalidated and the previous state of the presentation to be reconstructed. If we support the synchronization of the design and the presentation level, then output actions should be generated to re-establish the original value of the presentation attributes, for which the previous design attribute values should be asked from the application.

Besides the cost of the implementation, this approach can be criticised from the point of view of the designer: why should he be informed about inconsistent designs, not acknowledged changes? Thus the delayed propagation of changes to the presentation level is supported: the changes are propagated to the presentation level only after having been confirmed by the DS, i.e. if all the further inputs requested by the DS have been given by the user and accepted by the DS, and all the derived attributes have been given values by the DS with success.

The user can work on a presentation, make corrections for a while without the need for checking the consistency of the design. This is supported by the 'screen only' option. His inputs are stored in the PWM, and the corresponding design actions are not forwarded to the CWM. As soon as the user resolves the 'screen only' restriction, after having forwarded all the messages addressed to the CWM the control is given to the CM. The user profits from this facility, because:

- he can overdefine attribute values several times, only the last value given to an attribute is to be forwarded to the DS;
- the user is allowed to specify attributes in any order, he is not forced to give them when and in the order needed by the DS.

3.5. Error recovery

The user is allowed to terminate a communication cycle by the quit action. The DS is informed to refer to the consistent design version, as the current design. Then, all the previous design actions are undone: the delayed messages from the DS indicating design actions are removed from the CWM. The specifications received from PUIs are sent back with the original value of the attribute in question. The PSDs and the attached windows are updated according to the re-established attribute values.

If the user corrects one or more of his previous design actions, then all the chosen design actions are undone, and the first attribute chosen by the user for modification will be requested.

Undoing one given design action means to forward a value-removal message to the DS and to the corresponding PUI, removing the further design actions requested but not having been accepted by the DS, and the DS's design messages (if any) responding to the specification message.

Those previous design actions by the user which were acknowledged by the DS, will be re-used: when requested, they will be 'consumed' by the DS just as if they were the latest input by the user.

If the user decides to change an attribute's value which has not been given in the current communication cycle, then this implies a quit action.

3.6. Ease of customizing

The UI can be tailored to the user's needs in run-time, by prescribing preferences, specific usage for presentations, opening/closing/activating certain presentations.

By separating the lexical-syntactical elements and the rules concerning the control of the communication, the UI can be easily customized to fulfill requirements as follows:

- the lexical elements and their visual appearance (synonyms, icons, menu layout, echoes) can be changed by modifying defaults or constraints for attributes of presentation entity types;
- presentation types can be modified by adding new entity types;
- entire PUIs can be added/deleted;
- local behaviour of presentations can be altered, by changing rules in the PRM (e.g. immediate/delayed update of the screen);

- the service user interfaces can be altered to the user's need of support (e.g. indicating the error versus giving explanation in the case of constraint violation error).

4. Conclusions, further issues

It has been shown, how object-oriented and rule-based programming can be used as implementation tools for the lexical-syntactical and the semantical-conceptual definition of a user interface, in order to meet the following specific requirements of CAD:

- multiple presentations of the design, which can be structured according to the stages in the design process (refinements), the different media which can be used (alternatives) and their intended usage (preferences);
- the correspondence of the different presentations;
- interactive, incremental instantiation and modification of design attributes.

Other characteristics not restricted to CAD applications are:

- mixed-initiative communication, user-application symmetry;
- offering a choice of different media for output, not only for input;
- help and error indication services, undo and error recovery;
- adaptation to the state of the communication;
- run-time binding of the media to be used;
- ease of tailoring, both concerning the syntactical and the conceptual level.

All the same, we are aware of not having discussed many important issues, first of all those of the device level, especially the bottlenecks and conflicts (time of redrawing a screen, space requirements). A similarly subtle control of the replacement/resizing of windows is needed to support the efficient and comfortable use of the facilities provided.

At present, the error detection/recovery and cancel is restricted to messages still pending in the CWM. By storing the past communication cycles in a history-file, the scope of these services could be enhanced to a whole session with the design system.

We have addressed the problem of multiple presentations of one and the same design. A design environment allowing the comparison and parallel improvement of different versions of designs could be supported by allowing

the same type of presentations of different designs, sharing presentation instances, copying presentations physically or virtually.

Acknowledgments

The work reflects some ideas which raised during the inspiring discussions with A. Márkus about an early draft of the paper, and with G. Krammer about the topic of intelligent communication. Thanks to J. Váncza for his valuable comments on the paper.

References

1. **Arbab, F.**: A Paradigm for Intelligent CAD, in: Intelligent CAD Systems I, Theoretical and Methodological Aspects, P. J. W. ten Hagen, T. Tomiyama (eds.), Springer-Verlag, 1987. pp. 20-39.
2. **Barth, P. S.**: An Object-Oriented Approach to Graphical Interfaces, ACM Transactions on Graphics, Vol. 5. No. 2, April 1986, pp. 142-172.
3. **Card, S. K., Pavel, M., Farrell, J. E.**: Window-based Computer Dialogues, in: Human-Comuter Interaction, B. Shackel (ed.), Elsevier Science Publishers B. V., 1985. pp. 239-243.
4. **Densmore, O. M., Rosenthal, D. S.**: A User -Interface Toolkit in Object-Oriented PostScript, Computer Graphics Forum 6. 1987. pp. 171-180.
5. **Duce, D. A.**: Concerning the Specification of User Interfaces, Computer Graphics Forum 4, 1985. pp. 251-258.
6. **Forgy, C. L., McDermott, J.**: The OPS5 Users' Manual, Carnegie-Mellon University, Dept. of Computer Science, Technical Report, 1980.
7. **Gibbs, C., Won Chul Kim, Foley, J.**: Case Studies in the Use of IDL: Interface Definition Language, George Washington University, Report GWU-IIST-86-30, 1986.
8. **Harmelen, M., Wilson, S.**: Viz: A Production System Based User Interface Management System, Proc. of EUROGRAPHICS'87, Elsevier Science Publisher B. V., 1987. pp.331-345.
9. **Hatvany, J., Guedj, R. A.**: Man-machine interaction in computer-aided design systems, in: Analysis, Design and Evaluation of Man-Machine Systems, IFAC/IFIP/IFORS/IEA Conference, Düsseldorf, VDI/VDEGMR, 1982, pp. 265-272.
10. **Hatvany, J.**: Can computers compete with used envelopes? in: Preprints of the 9th World Congress of IFAC. Vol. 6. A Bridge between Control Science and Technology, J. Gertler, L. Keviczky (eds.), Budapest, IFAC,1984. pp. 93-98.

11. **Hopgood, F. R. A, Duce, D. A.**: in: Methodology of Interaction, R. A. Guedj et al. (eds.), Nort Holland, 1980.
12. **Hubner, W., Lux-Mulders, G., Muth, M.**: Designing a System to Provide Graphical User Interfaces: The THESEUS Approach, Proc. of EUROGRAPHICS'87, Elsevier Science Publisher B. V., 1987. pp. 309-321.
13. **Kimura, F., Suzuki, H.**: Variational Product Design by Constraint Propagation and Satisfaction in Product Modelling, Annals of the CIRP, Vol. 35/1/1986. pp. 75-78.
14. **Koegel, J.**: A Theoretical Model for Intelligent CAD Systems, in: Intelligent CAD Systems I, Theoretical and Methodological Aspects, P. J. W. ten Hagen, T. Tomiyama (eds.), Springer-Verlag, 1987. pp. 206-223.
15. **Lipkie, D. E., Evans, S. R., Newlin, J. K., Weissman, R. L.** Star Graphics: An Object-Oriented Implementation, Computer Graphics, Vol. 16. No. 3. 1986, pp. 115-124.
16. **Olsen, D. R., Dempsey, E. P., Rogge, R.**: Input/Output Linkage in a User Interface Management System, Computer Graphics Vol. 19. No. 3. 1985. pp. 191-197.
17. **Popplestone, R., Smithers, T., Corney, J., Koutsou, A., Millington, K., Sahar, G.**: Engineering Design Support Systems, Report DTOP/EXT/EDAI/09/1, Edinburgh University, 1986.
18. **Ruttkay, Zs., Allen, R. H., Laczik, B.**: A Multiparadigm User Interface for Intelligent CAD Systems, in: Intelligent CAD Systems I, Theoretical and Methodological Aspects, P. J. W. ten Hagen, T. Tomiyama (eds.), Springer-Verlag, 1987. pp. 242-255.
19. **Tanner, P. P., MacKay, S. A., Stewart, D. A., Wein, M.**: A Multitasking Switchboard Approach to User Interface Management, Computer Graphics Vol. 20. No. 4. 1986. pp. 241-248.

Interactive Design with Sequences of Parametrized Transformations

J.R. Rossignac
P. Borrel
L.R. Nackman

Interactive design with sequences of parameterized transformations

Jarek R. Rossignac, Paul Borrel[1], Lee R. Nackman

Design Automation Group
IBM Research Division
Thomas J. Watson Research Center
Yorktown Heights, New York 10598
JAREK@IBM.COM
(914) 945-3760

March 10, 1988

Summary: We present a paradigm for capturing the functional and relational aspects of a design as a sequence of parameterized unevaluated operations, which may, for example, correspond to the individual steps of a manufacturing process. A sequence that can transform a wide variety of models in a manner consistent with the user's intentions can be specified interactively using a single model as an example. The execution of a sequence to transform a particular model produces a result which may be interrogated to detect constraint violations and to locate operations responsible for creating or modifying specific features. An experimental implementation in an object oriented environment is described.

Keywords: CAD, Interactive Design, Modelling Operations, Parameterization, Unevaluated Representation, Functional Features, Object Oriented Systems.

[1] Visiting IBM on leave from LAMM, Montpellier, France.

INTRODUCTION

DESIGN PROCESS

Beginning with the formalization of a problem, the designer derives functional requirements for the physical realization of a solution, mapping the problem into one or more model spaces in which he can manipulate and analyze his design. Indeed, the design process can be viewed as an iterative transfer of information between two models: an **intentional** model, which captures the functional requirements, and an **extensional** model, which is a concrete realization of the functional requirements in a model space [1]. The intentional model is composed of abstract notions such as features, parts, and relations among them. The extensional model space may, for example, be a mathematical formulation of r-sets in E^3 represented as semi-algebraic sets [2]. A design activity aims at producing an extensional model that is consistent with the intentional model.

To cope with complexity, humans often use abstraction and hierarchical decomposition as part of the design process. Typically, the designer first lays out the characteristics of the major functions and components of the model. Then, through an iterative process, he designs each component individually and also in relation to the other components. A rough model of each component may be conceived first and details added later.

CAD SYSTEMS

The purpose of a CAD system is to provide throughout the design process a medium for expressing, interrogating, and modifying the current state of the model.

The "intelligence" of a CAD system seems constrained by the facilities it provides for representing and processing intentional information. The lack of such facilities makes automatic consistency checking impossible and forces the designer to remember the constraints that functional requirements impose on the extensional models and to test that his design meets all these constraints each time the design is modified. On the other hand, a complete representation of the intentional model permits, at least in principle, an automatic synthesis of a consistent extensional model.

Sequence of transformations

Commercially available systems typically do not attempt to capture the intentional model, but instead use a generate-and-test approach (see Figure 1.), in which the designer decomposes the functional requirements into statements, or commands, which are then automatically translated into actions that modify the extensional model. The actions are drawn from a limited set of primitive operations supported by associated underlying modellers. The available commands are typically few in number, but can be specified in the form of text input, menu selection, graphic interaction, or any other convenient way, and often accept parameters.

Geometric modellers offer good examples of such decomposition of the specification into individual commands. In CSG, solid models are constructed by defining primitive solids, combining sub-solids and moving them (see the PADL-2 modeller [3] for example). Euler operators, as described by Mantyla and Sulonen [4] and also by Krishnan and Patnaik [5], may also be used to construct geometric objects by specifying vertices, connecting them with edges, building faces from edge cycles, and finally specifying solids in terms of their bounding faces. Some systems accept higher level specification and provide more complex primitive shape modifications that create geometric features which have a specific function or correspond to a specific manufacturing process.

Consistency checks

Consistency between the model and the functional requirements can be checked by analyzing the resulting extensional model, i.e., by computing its properties and comparing them with the functional

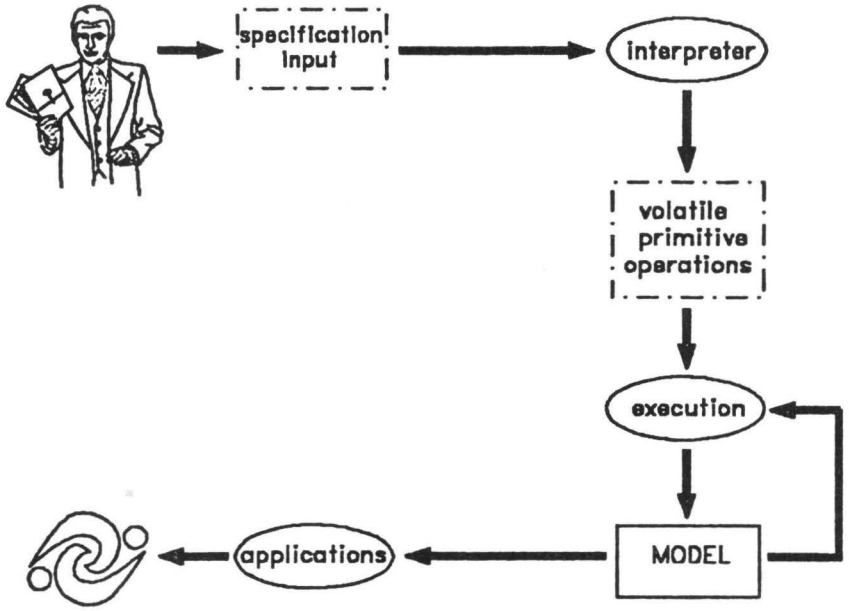


Figure 1. Conventional CAD systems: The user's specification is interpreted to produce a list of low-level operations which are executed to transform a model, then discarded. In some systems the specification is also discarded; in other systems, the specification is kept in a log-file, but no connection from the model to the specification is made.

requirements. This process is repeated until the designer is satisfied that the design, as captured in the extensional model, meets the functional requirements known by the designer but not by the system. Much can be gained by capturing the specification in the CAD system and by facilitating its modification and repetitive execution, as well as by allowing the user to interrogate of the result of these executions.

Representing relations

Often the problem expressed in the functional space can be hierarchically decomposed into subproblems that are loosely coupled and can be solved independently, producing subparts of the extensional model. The loose coupling can be captured as relations between these subparts. The general idea of using a graph, whose nodes represent solids, and whose branches represent constraints and relations that correspond to rigid motions, is described by Lieberman, Wesley, et al. [6,7]. Eastman [8] proposed a similar hierarchical location graph, where links correspond to relative positions between bodies and are stored as rigid motions. Lee and Gossard [9] extended this idea and described a data structure, which stores relations between components of an assembly, and can be created interactively. These systems do not address tighter couplings, such as relations between forms and dimensions of different subparts.

Guaranteed validity

Systems where users can produce invalid models add another layer of difficulties to the design process and are being gradually replaced by systems that guarantee that model validity is preserved by all

possible operations [3], and let users concentrate on the relation between the model and the intentional requirements without worrying about model validity.

Low level specification

A conventional CAD system has no common sense knowledge and cannot guess the designer's intentions from high level specifications, such as "make holes in this face to match the pegs of the other object when positioned such that this boss matches that slot". Consequently, users of conventional CAD systems must convert their high level specifications into low level data, such as point coordinates, angles, distances, and coordinate systems.

The low-level primitive operations available in many current systems are insufficient to capture the user's intentions regarding the function of parts and relations between them. Therefore, the user may have to provide an unnatural decomposition of his specification. This is time consuming and error-prone. Moreover, the loss of information inherent to this process complicates editing because the system does not provide information about the constraints that previously-defined commands satisfy. As a consequence, the designers correcting a specification that does not meet a particular requirement, may have forgotten why he made certain choices earlier and violate more functional requirements.

Capturing the specification

Certain CAD systems assist the user by providing construction tools that algorithmically translate high-level specification commands into a sequence of low-level operations, which are immediately executed and discarded. For example, some CAD systems provide many ways of specifying a circle, but store only its center and radius, discarding the relations from which they were derived.

Once a model was designed using a sequence of commands, it is important that these commands be saved for editing or later use. Several systems (see for example the GDP modeller [10]) offer log-file facilities for capturing commands in a file. A saved log-file can be edited or interrupted to permit user's interaction. However, if the commands are stored in a non-text form and if their meaning is context dependent, it may become difficult to replay such a file. Imagine specifying, as part of a sequence, a command which has as parameter a vertex of some object. The user may chose to pick the vertex graphically by pointing on the screen. The system is expected to interpret this selection and produce the appropriate vertex. What should be stored in the log file? The location of the cursor on the screen, the location of the selected vertex in the models coordinate system, a reference to the position of the selected vertex in the boundary representation of the model? None of these solutions is satisfactory, because, if an early part of the log file is edited in such a way that the model is not displayed at the same place, is moved in its own coordinate system, or has a different number of vertices, then the execution of the command that selects a vertex will fail to produce the expected result. Such situations are avoided if all commands in the specification are in text form independent of the model's particular state. Many changes to the design are easily done by editing the specification. However, it may be difficult to understand a specification without being able to interact with the corresponding model. Certain changes are best expressed by the user in terms of the model, for example, by pointing and dragging pictorial representations of some components of the model, in which case, to produce a context-independent specification, the system must translate the user's graphic interaction in a graphic-independent form. Such a transformation was implemented by Requicha and Chan in the VGraph extension of PADL-2 [11].

User friendliness

The selection of a CAD system may be dictated by its functionality and, to a lesser degree, its performance. However, its acceptance and widespread in industry is conditioned, not as much by its functionality, reliability, or performance, as by the facility with which users can specify and manipulate models. Indeed, it is important to reduce the design cost.

To address this problem much attention was given to "user-friendliness" and much effort was invested in the development of interactive front ends with tools for graphically specifying 2D and even 3D arrangements of shapes (see for example the work of Anderson [12] and Bier [13]). In spite of convenient graphic realtime interactions, these techniques offered limited help for designing models that satisfy any but the few standard constraints built in the system. No facility is provided for building constraints from other ones.

Re-usability

The necessity of further reducing the design cost and the fact that models of industrial parts are often built by combining simpler shapes, which may be grouped by similitude into categories, emphasized the importance of model re-usability. One could save the design time for a part if the design of a similar part was available and could easily be modified. Unfortunately, shape modification is not equally well supported by all modelling schemes. Consider the classical example of moving a cylindrical hole in a model of a 3D solid. Clearly, if the modification requires plugging the old hole and making a new one at a new position, as soon as the number of modification becomes important, using an old model becomes more expensive that starting from scratch. On the other hand, letting the user directly move the hole in the boundary representation may prove disastrous for the integrity of the model. Better results are obtained by limiting the user's interaction to a small number of numerical parameters which precisely define the geometry of the part. CSG schemes [3] provide simple facilities for constructing generic (parameterized) parts. Unfortunately it is difficult to design the specification of generic parts with input parameters that correspond to the functionally important characteristics of the part, such as specific dimensions or angles. This problem was addressed by constraint solving programs discussed in the next section.

AUTOMATIC CONSTRAINT SOLVERS

Algorithmic synthesis of the extensional model from the intentional model has been successfully implemented for "silicon compilation", where high-level functional specification is automatically translated into logic and layout, and, to a certain extent, for geometric modelling, where systems of simultaneous constraints are solved using inference engines or numeric iterations.

Numeric constraint solvers are limited to problems that can be formulated as a system of analytic (and often algebraic) equalities or inequalities. The logic programming approach can accommodate procedural solutions, but does not support cyclic set of constraints. Both approaches are typically unnecessarily expensive because they focus on solving a single global problem, and do not exploit the user's insight and ability to decompose the problem into smaller independent sub-problems, for which procedural solutions are often readily available.

Numerical constraint solvers

The idea of using a graph of constraints to represent geometry has received much attention. Most attempts are based on boundary representation and solve all constraints simultaneously, which results in a large system of (sometimes non-linear) equations.

In SKETCHPAD [14], Sutherland uses linear constraints between coordinates of line segments and circular arcs. Constraints are described by error expressions and solved by propagation of degrees of freedom, and by relaxation methods. The constraint-based graphic system JUNO, developed by Nelson [15], uses Newton-Raphson iterations to solve a system of constraints on coordinates of 2-D points, expressed in terms of distances between vertices, and parallelism relations between line segments. Constraints on coordinates were also used by Hillyard and Braid [16], who developed a general theory in which dimensions are used to specify algebraic constraints on the coordinates of vertices of a wire-frame representation. Instead of vertices, Lin, Gossard, and Light [17] propose to

use characteristic points of simple primitives bounded by planes, cylinders, spheres, and cones. The geometry of a part is specified by a fixed topology and a set of constraints (dimensions and angles) used to position the characteristic points.

Geometric constraints on curved surfaces can also be expressed explicitly without using topological information. The studies of Ambler and Popplestone [18], and of Lee and Andrews [19] address specifically the problem of converting constraints on surfaces into systems of equations. In both cases, the scope is limited to two constraint types: **against,** which matches two planes or achieves tangency between a cylinder and a plane [18], and **fit,** which aligns the axes of two cylinders. In Lee and Andrew's work, redundant equations are eliminated using a Jacobian matrix, and the remaining equations are solved by Newton-Raphson iterations, which may require a good guess for initial conditions in order to converge. Each constraint is converted into 16 or 18 equations and a typical 3-body configuration yields 100 equations with 84 variables. To improve the resolution process, Ambler and Popplestone developed techniques for separating the rotational and translational parts of rigid motions.

To support more general constraints, Borning developed THINGLAB [20], where constraints are defined by rules to test them and by methods (described algorithmically) to satisfy them. They can be incomplete, circular, or contradictory, and are solved in two steps. First, methods of propagating degrees of freedom or known states yield a plan for constraint evaluation and compile the result of a plan for incremental satisfaction of constraints into Smalltalk code. The code is used to solve, whenever possible, individual constraints in one path and relies on linear relaxation methods to deal with circularity. Constraints are not restricted to geometry. User-defined objects can be organized in a hierarchical manner, similar to CSG, allowing two objects to share common sub-parts.

The generality of the above global methods comes at a high price.

- The number of equations and variables grows fast with the number of geometric constraints and elements involved.
- The numeric methods require good starting points to converge and are often limited to linear systems.
- Furthermore, the declarative approach of a constraint-based specification gives little indication to the user as to what went wrong when the constraints are inconsistent.
- Minor adjustments of a specification may produce drastic changes in the solution selected by the system, thus constraint-based design is not well suited for interactive editing.
- The conversion between an intentional representation of a constraint to a set of equations may be more difficult than writing a procedure that transforms the model to satisfy the equations. Indeed, providing a sufficient set of constraints that capture various relations between the entities of a geometric model may be an overwhelming task.

Expert systems

Arbab and Wing [21] discuss how expert systems may be used to capture the functional constraints on geometric shapes, and even the assertions on how these constraints can be combined (for example transitivity relation) into rules. An inference engine is used to apply the rules whenever possible and derive abstract facts on shapes and relationships.

Bruderlin [22] describes a Prolog-based system which constructs 2D contours from geometric constraints, such as distances between vertices. Kimura, Suzuki, and Wingard [23] use first-order predicate logic to express implicit constraints on geometric entities. The topology of the part is assumed constant and the geometry is constructed by a sequence of transformations. An inference engine is used to determine the evaluation sequence. Similarly, Managaki and Kawagoe [24] achieve a parameterization of geometric models by storing relations between parts. Relations are specified implicitly by topological information and by limits on the values of variables.

These descriptive approaches suffer from the difficulty of computing the parameter domain over which the representation remains valid. These schemes are based on a boundary representation, which is composed of geometric descriptions of boundary elements (faces, edges, vertices) and a graph that captures their adjacency. Parameters are bound to the dimensions and relative position of boundary elements. Changing these parameters may produce boundary elements that are not consistent with the adjacency graph, and thus produce an invalid boundary representation. Since a boundary representation does not contain enough information to transform its adjacency graph that matches the new parameter values, these schemes are limited to parameter domains for which the adjacency graph is constant. To the best of our knowledge, no practical method has been reported to compute valid parameter domains, which may correspond to sets of higher dimensionality. To guarantee validity in current systems, a large number of additional constraints must be included.

Furthermore, expert systems do not seem to support incremental design and editing, because inference engines are not meant to cope with under-constrained problems. Moreover circular dependency among constraints cannot be resolved with logic deduction.

SEMI-AUTOMATIC SYSTEMS

Both previous approaches are computationally expensive, because they do not take advantage of the user's insight to break the problem into independent or loosely coupled smaller subproblems for which deterministic (procedural) solutions are available, or which can be solved by a specialized numeric method. Based on this assumption, Rossignac proposed a less general but more practical scheme, in which constraints involving the relative positions of curved surfaces are used to move primitives in a Constructive Solid Geometry representation [25]. Because unevaluated constraints are stored as an integral part of the CSG, the user can edit and parameterize his specifications as he does in the traditional CSG-based schemes. The system can inform the user that previously met constraints are violated, and even provide tools for satisfying certain new constraints without destroying of previously defined ones, but it is the user's responsibility to come up with an order of model transformations that will meet the functional requirements.

ORGANIZATION OF THE PAPER

The following section sets the scope and objectives for our research in the general framework described above, and presents the functionality of our experimental system, MAMOUR, on an intuitive level using a parallel with a human mechanics apprentice. Then, we describe the architecture and implementation of MAMOUR, explaining how each aspect relates to the issues discussed in the introduction. Finally we show some simple examples of how the current version of MAMOUR can be used in conjunction with a two dimensional geometric modeller specially developed for our experiments.

A MORE INTELLIGENT CAD SYSTEM

A basic premise of the work we describe in this paper is that despite much research on automatic synthesis of extensional models from intentional models (i.e., functional specifications), the generate-and-test approach will predominate in mechanical design for many years to come.

Our goal is to produce a CAD system, which integrates high level specification in terms of relations and constraints with the flexibility and reusability offered by parameterized representation. In addition, we focused on the convenience of editing the design and interrogating it, when it does not meet the functional requirements.

The purpose of our system is not to replace the user and automatically produce a solution to a problem that the user has formulated, but to collaborate with the user in the design of a solution. The user provides his global insight; the system takes care of tedious calculations, bookkeeping, actual model

transformations, and provides the user with valuable feedback. For global consistency between the intentional and the extensional model, we relay on the user, but provide him with high level concepts for expressing validity and with (automatic) checking tools.

DESIRED CHARACTERISTICS

A sequence of operations

We combine a declarative approach with a constructive (procedural) approach. As justified before, we expect the designer to break the specification into a sequence of transformations that can correspond to incremental shape modifying design operations, or can simulate manufacturing operations such as material removal with numerically controlled machine tools (as done by the system currently developed by Cutkosky and Tenenbaum [26]) or erosions and depositions processes occurring during the fabrication of integrated circuits (as done by the OYSTER system developed by Koppelman and Wesley [27]).

The sequence of transformations — much more than the result it produces — represents the design and captures the explanations of some of the user's decisions. As the user interactively designs a particular model, we capture his commands into a sequence of unevaluated specifications of operations. We provide tools for editing, parameterizing, and combining such sequences, and for executing them to transform different models (Figure 2).

Multi domain operations

To interface with various applications, the same sequence can be "executed" to produce different models with respect to different domains to account for aspects of the design that need to be analyzed. For example, a model for interactive design and engineering drawings would incorporate geometry, while a model for process planning would incorporate tooling requirements and cost estimates. The system supports queries of the specification and of the model. For example, a specification that combines drilling and painting operations may be interrogated, independently of any model, to collect a list of drill radii and paint colors. On the other hand, cost estimates, which may depend on the volume of removed material or on the area to be painted, will be different for each model and must be computed in conjunction with the geometry.

Extensible knowledge

The "intelligence" of a CAD system can be measured by the system's ability to understand higher level concepts and to execute tasks defined in terms of these concepts. The sets of all concepts and associated execution procedures define a language and its semantics. A flexible system of growing intelligence may be obtained by providing facilities for defining new concepts (and thus enlarging the vocabulary) dynamically. For example the users can attach names to elements of the extensional model or define new abstract concepts that can be used to interrogate or modify the model. The language grows rapidly if the concepts and procedures developed by one designer for his own purposes may be used by him or by other users to solve different problems and may even be combined to develop new concepts and more complex procedures. New concepts defined in a current language become part of that language.

Features

Features are abstract entities that combine functionally related elements of the model [28]. For example, the user might want to "drill a hole named HOLE1 at the center of SLOT1". A relation between two features, SLOT1 and HOLE1, is thus defined, and is stored as an unevaluated expression (e.g., "HOLE1=DRILL(SLOT1.floor.center)") which, when the specification is executed, computes the position of HOLE1 with respect to SLOT1 in the model. The constraints imposed by problem

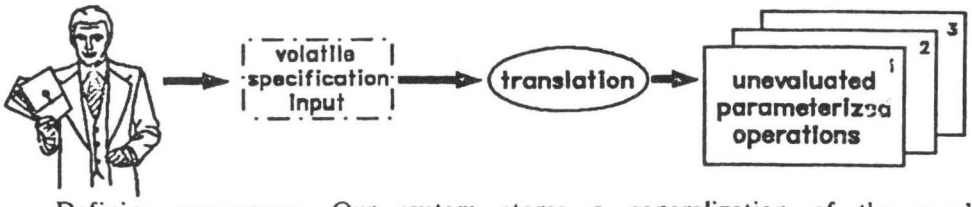


Figure 2. Defining sequences: Our system stores a generalization of the user's specification in a sequence of unevaluated parameterized operations.

requirements can be expressed in the model space as validity rules, provided either by the user or automatically by the execution of an operation. Here, the drill operation will modify the geometry and also create an abstract entity called HOLE1 which refers to some elements of the model's boundary. The validity of HOLE1 may be formulated as a requirement that the hole refers to a boundary element which is a specific unaltered cylindrical face. Some subsequent operations may cut a portion (or the totality) of that face. Such violation of the rules defining a valid HOLE feature should be automatically detected and responsible operations brought to the user's attention.

Editing

Engineering changes are simplified because the result of a particular execution can be interrogated in terms of the original specifications [29]. Consistency between the model and the specifications is automatically maintained during engineering changes. A good example is found in solid modelling: to move a misplaced hole the user can edit the specification step that defined the position of the hole instead of plugging the misplaced hole and making a new one somewhere else. The system can provide additional help, at any time, by indicating what part of the specification created the hole.

Reusability

The ability to use the same specification in different situations is achieved by storing parameterized sequences of unevaluated operations and by providing a mechanism for executing them in different contexts. A sequence of operations that represents a particular transformation of the model can be parameterized, archived, combined as a simple operation into other sequences and executed to transform (in a consistent manner) models that vary considerably. (For example, in solid modelling, a sequence that fillets all slots of sufficient depth can be applied to a variety of solid models, which can differ not only in dimensions, but also in shape and topology.) All models that can be processed by a sequence in a manner consistent with the designer's intentions form the family of models associated with that sequence.

However, the user may wish at first to concentrate on the product to be designed, and not on a parameterized specification that produces a whole family of similar products. On the other hand, if an early part of the specification must be altered by some editing operation and subsequent commands must be re-executed, the user expects these commands to perform in the new context in some reasonable manner and to produce the desired results. For example, if one command builds a slot at a given position and the next command drills a hole at the center of that slot, it would be inconvenient if the user had to redefine the second command just because he modified the position of the slot in the first command.

Generalyzing from an example

A specification typically involves features expected to be present in each member of the family of models for which the sequence of operations was designed. Since people find it difficult to design in

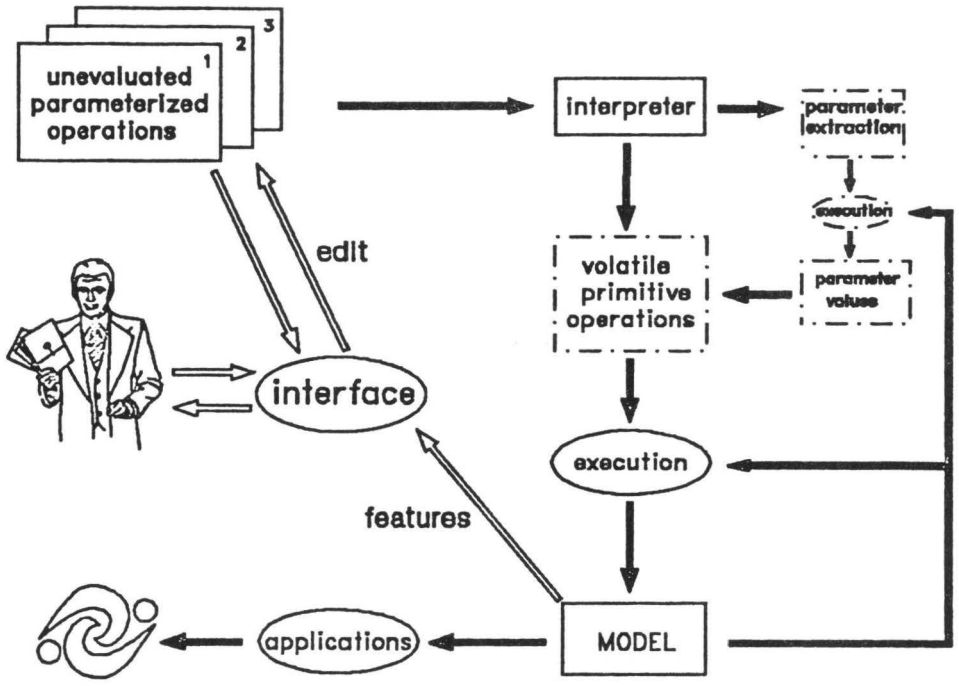


Figure 3. Executing sequences: Given any model of the family upon which the sequence can act, each operation of the sequence is executed to transform that model. Execution is done by calling primitive operations in a specific domain determined by the user. To evaluate the arguments of these operations, the system must first evaluate the expressions of the parameters of the high level operation. The evaluation may involve computing measures on certain features of the model in the current state. Through the interface, the user can query the resulting model in terms of features, constraints, and high level operations. He can also edit the high level operations using information supplied by the model.

the abstract, a representative example of the family of models is used to help the user specify the relevant features by graphic interaction. For example, an operation that drills a HOLE at the center of SLOT1 expects to find, in all the models it transforms, a feature named SLOT1. However, if the user wishes to drill a hole in a slot that has no name, he must first indicate which slot he means. The system converts such graphic specifications into names that are consistent with the origin of the graphically selected elements. Names — and not the graphic interactions — are stored in the specification, so that subsequent executions on an identical original model or on other models of that family will not require user's intervention.

Capturing relations and constraints

The user conceives the specification in terms of what it should do to the models. Therefore, it should be possible to incorporate measures — to be derived at execution time from these models — into the specification. User's specifications may also contain relations between various components of the models. These relations may be specified by expressions derived from some input parameters as in the parameterized solid modelling specification schemes [3] or through construction steps whose param-

eters are expressed in terms of attributes or measures derived algorithmically from certain features of the models.

EXPERIMENTAL IMPLEMENTATION

The ideas described in this paper are embodied in the prototype system MAMOUR [30], developed to support our study of the interactive design through sequences of operations. MAMOUR is implemented in the object oriented language AML/X [31]. The specifications of the parameters of these operations are stored as unevaluated AML/X expressions to be evaluated at execution time by the AML/X interpreter. Execution in the geometric domain is achieved by invoking functions of an independent geometric modeller with sufficient capabilities. Simulating the execution of the operations on a specific model produces a geometric representation, a list of features, and a list of non-geometric attributes. Constraints features may be tested automatically. Features and geometric elements may be interrogated to obtain the list of operations that produced or altered them, and a list of parameters whose values are directly or indirectly used by these operations.

AN INTELLIGENT APPRENTICE

As stated earlier, our approach is not meant to develop an autonomous CAD system that would generate the model from the specification of the functional requirements, but to provide an intelligent assistant. MAMOUR can be considered as the computerized version of a mechanics apprentice that can accurately perform well defined tasks. MAMOUR is given a set of parts (for example castings or machined parts) and instructions on how to process and transform these parts. These instructions are decomposed into simple operations. Some tasks and services provided by MAMOUR are listed below, using as example various processes involved with the fabrication of mechanical parts.

- Modify the part's shape through DRILLING, GLUING and other such operations.
- Recognize a feature defined by simple concepts, such as "the largest slot" or "the top face", or 'the hole labelled A".
- Understand the semantics attached to features of specific types (for example the WIDTH of a SLOT).
- Compute dimensions and other attributes of features.
- Derive measures from the geometric elements of the part and perform various arithmetic calculations to position points and directions with respect to the part.
- Paint faces and label them with names and annotations.
- Produce a list of features already defined on the part.
- Check whether these features satisfy the user's requirements or if they were altered by operations subsequent to their creation or identification.
- Use another finished part as an example to obtain specific dimensions upon which other shape modifications depend.
- Remember sequences of operations by name and execute them again on new unprocessed parts, which may be different.
- Learn new names, new concepts, and new operations.
- Be able to modify remembered sequences by changing certain parameters.
- Remember acceptable parameter ranges for operations (for example a tolerance on the radius of a cutter) and pick appropriate values given the available tools during execution.
- Show the resulting part after each operations.

- Answer questions as to which operation is responsible for creating certain characteristics of the part.

To educate MAMOUR, the user must demonstrate how to do a large variety of useful operations on a large variety of parts. Operations such as, "fillet concave corners of all SLOTs" can be composed from simpler operations ("S = all SLOTs" followed by "fillet S"). Such a sequence is very **explicit**, i.e., it does not depend on some parameter that may be implicitly defined by the user. On the other hand, "drill a HOLE in this face", where "this" is defined graphically by the user, does not provide an explicit means for capturing the user's intent. Such an ill-defined specification may lead to errors when it is executed on a slightly different part. We provide unambiguous ways to accurately express the user's wishes, but such verbose explanations are unfriendly and error-prone. The best is to let inexperienced users **show** graphically what they mean on an example, and later add more information if necessary. MAMOUR must understand how to adapt these graphically-specified operations to parts which may vary in size, shape, and even topology. To educate MAMOUR, the user will indicate what is to be done while processing a single part and pointing on the crucial geometric elements to measure or modify. When the same sequence has to be applied to a different part, MAMOUR tries to identify the crucial elements again on the new part. If it fails, the user has to be more specific and indicate how to identify these crucial elements on a larger class of parts. For example, say "take the bottom face of SLOT A" or "the face of the hole that you drilled at the beginning". It is the user's responsibility to make sure that MAMOUR knows exactly what to do, but *errare humanum est*, and the trial-and-error process may take place before MAMOUR is correctly *programmed*. However, correctness may be impossible to ensure, since the designer of a sequence of operations may be unable to envision all the parts to which the sequence may apply.

MAMOUR'S ARCHITECTURE AND IMPLEMENTATION

In this section we describe the current architecture of the experimental system MAMOUR.

Because geometric models have been precisely defined, are important in many applications, are associated with a clear intuitive interpretation, and provide convenient means for graphic interaction, we focused on the geometric domain in our experimental developments of MAMOUR, and illustrate the following description with geometric examples. However, we strongly believe that the mechanisms developed in MAMOUR are adequate for supporting applications in other domains.

PRODUCT DOCUMENTATION

The execution by MAMOUR of a sequence of user-defined operations with respect to a particular domain produces a complex structure, which represents the current state of the designed object and of the associated intentional and abstract information. This structure, called the **product documentation** (abbreviated PD), is a computerized generalization of the traditional blue print. It contains one or several extensional models, a list of features and a list of attributes and internal variables.

Extensional models

An extensional model (or simply model) is a complete representation of the product with respect to a specific domain (for example in the geometric domain, it can be a boundary representation for a solid).

Several extensional models may be simultaneously present in a PD. They may refer to different domains or to different representations in a single domain. Two refer to the geometry of a model and provide different representations (CSG and boundary). One refers to the dynamic properties of the product. Models produced directly by the execution of a sequence are called **primary**. (For example the CSG model.) Models that are derived from primary models are called **secondary**. (For example, the dynamic model is derived from a primary geometric model.) Note that boundary representation

could be obtained directly as a primary model or derived algorithmically as a secondary model from CSG; we used it as a secondary model in our experiments.

Secondary models are not evaluated until they are needed. (For example a boundary representation is only computed when the user wants a picture.) Consistency between primary and secondary models is maintained by setting a validity flag to TRUE when the secondary model is derived, and to FALSE when the primary model is altered.

Features

Features, which may correspond to functional entities of the intentional model or to subparts of a model that have particular significance for some application, refer to aggregations of entities of an extensional model — for example, a set of faces of a solid.

Features provide different abstract and compact views of an extensional model. For instance, a manufacturing view of a part (represented by an extensional boundary model) can be a list of manufacturing volume features (delta volumes) to be removed from the model of a raw stock of material in order to obtain the part. These volume-features can be categorized by type and conveniently represented by a few parameters. However, automatic process planning of numerically controlled machining operations may require additional information, such as feature accessibility, which can be derived algorithmically if the features contain references to elements of the parts. A different view of the part may be required for robot manipulation planning, which is concerned with inertial properties and grasping positions. Features also provide a convenient means for associating attributes to collections of part elements (depth of a slot, finishing tolerance for manufacturing, electric properties).

Features provide a means for associating attributes to subsets of an extensional model. (For example, a color associated with all the faces of a feature.) This association is achieved by attaching to each feature a set of variables (feature attributes) whose values can be interrogated and modified.

Users and application programs can manipulate features by name and access the feature attributes by the attribute name. (For example SLOT1.COLOR is the color of the feature SLOT1.)

Features may be assigned a type, which provides a way of associating some semantics with a class of features that exhibit certain common characteristics. (For example, all features referring to a planar disk-face connected to a cylindrical face in a geometric model would be of the type HOLE.)

Usually, to each feature-type is associated an agreed semantics understood by the system developer, the user, and the application programs. However, it is difficult to capture in a program the abstract semantics associated with a feature-type without using knowledge representation tools to model at least some fundamental principle of naive physics or its counterpart in the domain under consideration [32]. We only capture the part of this semantics that defines the interactions between features and users or application programs. These interactions are encoded in specific procedure (called *methods*) associated with a particular feature-type.

The internal variables of a feature are used in three different ways:

- references to entities of the extensional model (such as the boundary elements of a boundary representation of a solid) — they provide means for aggregation,
- discrete facts and continuous measures derived from the extensional model at a specific moment of the feature's life (such as the absence of the bottom of a HOLE in the final stage or the depth of a POCKET at its creation),
- feature attributes, which may not be derived from the extensional model (such as color or tolerances), but provide a means to associate an annotation with a particular part of the extensional model.

Features are related to the extensional models, through an aggregate of references to specific entities of the model. Such references, together with the feature's methods, can be used to conveniently access geometric elements that, accordingly to the understood semantics, play a specific role. For example, "HOLE1.BOTTOM()" refers to a particular subset of the parts boundary that corresponds to the common notion of the bottom face of a hole. This increased vocabulary provides a higher level specification language and provides means for capturing relations between various geometric entities.

When a secondary model (such as a boundary representation) provides a more natural arena for user interaction, features must guarantee access to entities of the secondary model. To this effect, when the secondary model is derived algorithmically, we associate to each one of its entities a unique name which is either explicitly set by an operation or derived algorithmically from the primary model. To ensure consistent behavior of a sequence, when executed to transform different PDs, it is essential that algorithmically derived names for elements of the secondary model be consistent with the relation of these elements hold to particular aspects of the primary model. (For example, in a boundary model, we associate with each boundary element an identification of the primitive or primitives of the CSG model that produced the element.) These individual elements of a primary or secondary extensional model can be accessed by their names in a manner similar to the way attributes and variables are accessed. On the other hand, they can be accessed through methods attached to features (for example FLOOR(SLOT1)) or by invoking procedures that will locate them automatically from some higher level description (for example the FIRST LEFT MOST EDGE).

To each feature type or instance can be associated one or several sets of validity rules, which are Boolean expressions combining the elements referenced by the internal variables of the feature. These rules may be evaluated automatically to determine if the elements referenced in the particular feature satisfy certain properties implicitly associated with features of that type. For example, the internal variables of a HOLE feature make reference to a set of geometric elements (such as a complete cylindrical face), which must be present in the final part, for the feature to be considered as a valid hole.

It is not required from a feature to be valid in any sense in order to to exist in the PD. Indeed, the validity may be defined by several sets of rules, applied in different domains of application. Also, it might be the user's intention to consider a part of the design as a feature even though it does not completely comply with the validity rules. Moreover, a feature may become temporarily invalid at an intermediate stage of the design, and then be reestablished (for example, it might be more convenient to apply a transformation to the geometric representation of the feature in several steps, even though the intermediate steps do not comply with the validity of the feature). Finally, it is more important to know why a feature is not valid than to just notice that it is not valid. This information can be derived from the interrogation of history of the elements referenced by the invalid feature.

Features of a given type are encoded (in the object-oriented paradigm) as instances of a particular class for which the internal variables and methods have been defined.

The concept of inheritance is extremely useful for defining new specialized features from existing ones. For example, a RECTANGULAR__SLOT may be a specialized version of a SLOT. It will have all the properties of a SLOT, will be associated with the attributes of SLOTs and with measures such as DEPTH that were already defined for SOLTs. All this semantics can be "inherited" from the definition of the SOLT concept by saying that a RECTANGULAR__SLOT is a SLOT. In addition, the RECTANGULAR__SLOT may have it's own abstract concepts and methods. Since class inheritance was only recently added to AML/X, our experimental version does not explicitly use it, but instead provides schemes for easily creating new features and new operations, without having to duplicate the methods and declarations that are common.

Since internal variables of a features are references to any part of the PD, they can refer to other features and be used to build a hierarchy of features or to represent compound features [1].

It is not necessary to link feature creation and feature recognition during the design process. The designer should have the choice to make the feature explicitly or to modify the shape through other means and then communicate to the system that he wishes some elements to be treated as a feature. Both approaches may lead to inconsistencies when used during an incremental design process.

A feature can be created in two different ways:

- A feature can be defined by an operation, as a reference to the result of its execution. For example, an operation of type DRILL creates in the product documentation a feature of type HOLE, and at the same time modifies the extensional model describing the geometry (for example the HOLE feature references the geometric elements of the cylindrical part subtracted from the model).
- A feature can be created "after the fact" by the user, in order to take into account side effects of previous operations or to capture some characteristic aspect of an extensional model. For example, if one or several sequences create two BOSSES on top of a part, the user may want to consider the space between these BOSSES as a SLOT, and interactively associate the corresponding elements of the model to the type of feature SLOT. This allows the user to add some part of his knowledge and interpretation to the product documentation, exactly as he would do it by annotating a blue print. In certain cases, as for example for studying the manufacturability of a part, a sequence of automatic feature extraction can replace the user in this task.

Attributes

Attributes play the same role as features in the product documentation in this sense that they encapsulate part of the knowledge the user has of the product, which is not represented in the extensional model. To the contrary of features, attributes contain no reference to the extensional model. Instead, they qualify the product or convey additional information, such as material properties of a solid part, some important recommendations regarding the product, or references to commercial standards that qualify the product.

Attributes are stored as name-value pairs in the PD and can be accessed by the user or by any operations using their name.

Variables

Variables are a specific instance of attributes and are used to pass information between operations, in the same way as global variables in a program. Variables may have any type (arithmetic, string, aggregate, object of a particular class defined for a particular application, procedure or method name). They are often used to store results of the analysis of the intermediate state of the product. Indeed, an important amount of time is spent in analyzing the results of the previous steps to guide forthcoming operations. For example, one operation may measure the depth of a pocket and store it in some variable that will be subsequently used to evaluate the parameters of several operations.

Similarly to attributes, variables may accessed by name and be used in parameter expressions of operations.

History

All entities (extensional model, features, attributes, and variables) are stored in an extensible structure which associates names, values, and history. Names are used to access the entities. Values can be objects or aggregates of objects of any type. A history retains references to the operations that created the entity or modified it.

SEQUENCES

A sequence is a user specified succession of operations, whose execution creates or transforms PDs.

The user may specify a new sequence similarly to a class in the object-oriented programming philosophy by providing the list of operations it contains. He can produce instances of that sequence which are differentiated by the values of their parameters. Instances of sequences can be executed to transform different PDs. The same sequence may be able to execute several "actions", which correspond to various domains of interest. Sequences can be stored and combined into other more complex sequences.

Executing a sequence consist in applying one of its actions to a product documentation. Since it is always possible to reconstruct a product documentation by re-executing all the sequences that led to its creation, these sequences constitute an unevaluated representation of the product.

The actions are combination of queries and modifications to be applied to the product documentation. Each of them typically addresses a particular aspect of the product documentation. They allow the user to interrogate and/or modify the product documentation in different domains of applications using the same predefined sequence.

For example, executing a sequence of drilling operations in a geometric domain would produce an extensional model that represents the final solid and a list of HOLE features. On the other hand, executing the same sequence on the same PD from a perspective of tooling requirements for manufacturing process planning may result in an extensional model that is a collection of (radius,axis) pairs — one per hole — and of features that aggregate these pairs by size and orientation. The manufacturing model could also be derived as a secondary model from the list of HOLE features associated with a primary geometric model.

It is not required that all the operations of a sequence can operate in the same domains. Executing an operation in a domain for which it was not designed results in no action.

Sequences can be edited by adding or deleting their operations or by changing the parameter expressions associated with each operation. Sequences can be archived and retrieved. A sequence can be viewed as a new operation (it has a type and a set of parameters). Consequently it can be combined with other operations or sequences to form more complex sequences.

Operations

The execution of a single operation defined by a command transforms the PD by modifying the extensional model, adding new variables or attributes, or modifying the value of an existing variable or attribute, or creating a new feature.

Operations are represented by AML/X objects, whose internal variables contain the list of actions, that particular operation may execute, and a list of unevaluated parameter expressions. The parameter expressions are stored in text format as internal variables. This text format reflects what the user typed, except for certain references to entities of the extensional model that were derived by MAMOUR from users' graphic selections (see below).

The parameter expressions can combine procedure calls, attributes and variables of the PD, reference to feature attributes, and measures to be derived from features (for example DEPTH(SLOT1)). Because any AML/X expression is acceptable, parameter expressions may contain conditional statements and various AML/X operators. Low level geometric entities and operators were added to AML/X [33] to provide an intuitive syntax for expressing geometric constructions. (For example, "(3*P+Q)/4" evaluates to a point between the points P and Q, and "4*R/M" is a product of a scaling by 4 with a rigid motion R combined with the inverse of a rigid motion M.) On the other hand,

commonly used AML/X constructs can be captured into procedures with mnemonic names, which would be more convenient for the novice user.

Methods specifying extensional elements

In this section, we illustrate the issues and techniques for selecting elements of an extensional model in the context of geometry.

MAMOUR provides two different primitive methods that can be combined for selecting geometric entities in the boundary graph: **explicit names** and **procedurally defined selection,** which is a non-evaluated descriptions of entities of the extensional model.

All entities of the extensional model have names, which may be used to consistently identify characteristic elements in each member of the family of products, regardless of the position, shape, or size of these elements. Consistency requires that each element keeps its name during the life of PD.

It would be too tedious to require from the users to assign names to all the elements to which they refer. MAMOUR provides an **automatic process** for assigning default names. Automatic names of geometric elements in a boundary representation are built using their provenance in the sequence. The provenance of each element (called its **history**) is composed of the name of the operation followed by a path.

A path is specified as follows. Operations that modify a geometric representation can add or subtract material to the part, move a sub-part, or change a boundary locally. To ensure that all operations produce valid geometric representations, we chose to support, as geometry modifying operations, only ones that can be expressed in CSG as additions or subtractions of material. Consequently, to each operation may be associated a boundary of the region to be added or subtracted from the representation of the model. If the region is represented in terms of CSG, a path to one of its boundary elements is composed of some encoding of how to reach a particular primitive from the root of the tree, followed by a number that identifies a boundary element of that primitive. If the region is defined in terms of boundary, the path is reduced to an element-number. In both cases, a path uniquely defines a primitive face in a CSG graph.

Such history names are independent of the primitives dimensions and positions and of the overall solid position. Moreover, the position of a primitive in a CSG tree reflects, in many situations, the user's hierarchical decomposition of the design problem, and thus is better suited to capture the user's intention than geometric location.

In order to disambiguate the situations where the automatic names would be meaningless (the same boundary may be produced using different CSG trees), the users (and thus the sequences) can override this process, and give explicit names to relevant elements of the boundary, by attaching names to CSG primitives.

As long as an element exists in the model, it will keep the same name. To meet this requirement, a single geometric element may have several names (when several different elements become coincident after the action of an operation) and a single name may be affected to several elements (when the action of a sequence "splits" an element into several pieces — each of the pieces can then be selected using the common name and an index).

Because the history contains a reference to the operation, the history of creation of the geometric elements of the boundary can be used for selecting them. For example, the statement:

```
EDGES.MADE_BY ( SEQUENCE.TYPE = DRILL )
```

refers to the edges of the holes made by any DRILL sequence (or what remains of them).

Instead of using names, the designer can invoke high-level, model independent methods for retrieving extensional elements. A predicate is used to decide whether a particular set of elements meets the requirements for selection. MAMOUR provides a method that searches two dimensional boundary representations and extracts sets of consecutive edges that satisfy certain requirements expressed in terms of rules (or predicates) which combine measures derived from the dimension and relative position and orientation of these edges. Rules defining feature validity may be used in this manner to extract features automatically. For example, a sequence that fillets the corners of all slots of a geometric model, could start by an operation that sets an internal variable to the result of a subroutine that locates slots. A subsequent operation can execute a FILLET operation on all the elements referenced by that variable. Such a sequence is independent of the number of slots and of their origin.

We have successfully experimented with facilities for extracting simple features in two dimensions, and we concluded that rules for common 2D features are very simple (we only support rules that refer to a fixed number of consecutive boundary elements). However, we realize that our approach has serious limitations; specifically finding features that contain a variable number of non-necessarily adjacent boundary elements in 3D models may prove complex and time consuming.

Executing a sequence

In MAMOUR, a sequence is executed on a product description in a given domain. The operations of the sequence are executed one at a time. The execution of an operation may change the extensional model, set or modify the value of an attribute or variable, create a feature, or do all of these things. The resulting PD is passed to the next operation. The names of the domains in which the whole sequence is executed is propagated to the operations, which in turn check if they should be executed in

To execute an operation, MAMOUR first checks if the particular operation has an action that pertains to the specified domain. If not, then no execution takes place for this operation. Otherwise, the appropriate method will be invoked, but first the values of the operation parameters must be computed by evaluating the appropriate expressions.

As pointed out before, such expressions may contain references to attributes and to measures to be derived from the model in its current state. Before MAMOUR can evaluate a parameter expression such as "DEPTH(SLOT1)+X", it must first parse the expression and extract the names of all variables (here SLOT1 and X). Then MAMOUR searches for these names in the sequence parameters and the PD.

If some variables are not in the PD currently processed, they can be provided in another "example" model selected by the user (which can be the model on which the sequence was defined in the first place). This mechanism provides a way for having a default setting for these variables. In the last resort, if any variable name remains unmatched, MAMOUR asks the user to provide the corresponding values (or optionally, raises an exception).

At the end of this matching process, each variable name is associated with a value. MAMOUR creates temporary local variables with these names and the associated values, and simply evaluates the whole expression.

The parameter expressions may contains procedure or method calls. A few commonly used methods have been developed with the system, (e.g., LENGTH of an edge, depth of a SLOT feature), others may be defined by the user, in a manner similar to subroutines and saved to increase the high level specification understood by MAMOUR. These procedures are executed and may derive certain characteristic measures from the extensional model. (For example, DEPTH(SLOT1).) Because variables and attributes of the PD may be set by one operation and used in the parameter expression of another subsequent operation, they provide a means for passing information between operations and to capture some relations between different characteristics or features. (For example, the width of a

slot can be set to be twice the radius of a hole by using the same variable X in MILL(width=X) and in DRILL(r=X).)

Once the parameters expressions are evaluated, the parameters values are stored for later interrogation and also passed to the method associated with the execution of the operation in a particular domain.

Specification on a characteristic example

Often a user defines a sequence because he needs to produce a particular product description. He does not want to address the problems of generating a sequence that can be archived and parameterized, that can be applied to transform a family of models, or even that can be edited. However, editing facilities are crucial to the design process and re-usability is an important economic factor. To reconcile interactive design with the construction of generic specifications, we propose a design environment, in which the user thinks he designs a model, while we save an adaptable sequence.

We experimented with facilities for extracting a parameterized and flexible sequences from specification processes that focus on the design of a single product description. The idea is to capture the user's intention, expressed to solve a particular problem, and to extrapolate it into a specification that will solve a family of similar problems. The general principle behind our approach is to let the user specify modification of a PD in terms of the content of the PD, and automatically derive a specification that is independent of the particular PD. For example, a boundary element referenced in a parameter expression of an operation, may be specified graphically by the user. However, in the specification, the history of that element will be captured, so that, when the sequence is edited and executed again, or applied to different products, an element with the same history will be used. The assumption we make, is that elements with the same history in different PDs are perceived by the user as playing the same role in its specification.

History-based specification may not correspond to the user's intent, in which case the user has the option to express his intents unambiguously and explicitly in text form, either in its initial specification or as a correction.

The variables referenced in parameter expressions used at this point should be present in all the PDs of the family (or at least be available in an example PD available during each execution), otherwise the user will be asked to provide a value, each time a name, for which a matching cannot be found, is used.

MAMOUR was designed to provide facilities for specifying parameter expressions that will evaluate to the desired result without ambiguity. When the command has to be executed on a variety of models, producing expression that will consistently evaluate to an "intuitively" correct result may prove difficult. The most delicate problem that we ran into deals with the selection of boundary elements, because the boundary geometry and even topology may vary from one model to another in a family.

We experimented with different tools. One is the feature location procedures discussed above. Another is to use names of boundary elements. These names may have been specified explicitly by some previous command are derived from the history of each boundary element, and thus unknown to the user, in which case they may be selected graphically. However, even in a simple 2D world, one may need to distinguish between several disconnected pieces of the same primitive edge, and correctly process overlapping edges. To deal with such problems, we expect the user to provide a parameter expression that will disambiguate the selection, but we help him specify the names of the boundary elements. The user types the expression and can use wild card characters, which will be replaced by MAMOUR by boundary element names derived from graphic selection. For example, the user can say explicitly that he wants the second connected segment of an edge that comes from a specific

primitive. All he has to say is "$(2)" and the system will replace the "$" sign by the name of an edge that must be graphically selected by the user. To obtain that name, the user is put in a graphic mode in which he can use the cursor to select edges and vertices, or traverse the boundary from one element to the neighboring one.

To design a sequence, the user needs not remember syntax nor the list of available operations. A user-extendible menu helps him select what he wants to do. Once an operation type is selected, MAMOUR prompts the user for the name of the operation and for its parameter expressions, explaining the significance of each parameter. The expressions are parsed as the user types them in, and references to unknown variables are trapped. The user is asked to provide values for these variables with the option to make them the parameters of the sequence.

After its specification, each operations is executed on the current PD. The result is displayed. For geometric applications, the graphic display may be used to interactively select geometric elements referenced in the parameter expression, as explained above.

Analysis

To check the correctness, the user can execute the same sequence on different members of the PD family and in different domains.

After each execution, the user can check the validity of the resulting PD by comparing the internal variables of operations, the variables, features, and attributes of the PD to values determined by functional requirements. Automatic validity testing can be obtained by capturing the functional requirements into feature validity rules, and have them evaluated automatically.

If the resulting PD does not meet the functional requirements, the user must change the sequence, by redefining the parameter expressions of an operation and by adding or deleting operations or sub-sequences.

Using features and associated measures, one can easily produce complex rules that will check whether certain requirements are met. To each requirement, one associates an attribute, which will be evaluated upon demand or automatically. Correct values for attributes will assess the validity of the design. It is the user's responsibility to correctly define the attributes. However, commonly used attributes may be predefined for each feature type. For example, the attribute ACCESSIBLE may be defined for a feature type HOLE, and a method for computing its value set-theoretic considerations may be provided. To drill a HOLE of radius R, one may need enough material around the hole (say an annulus of thickness E) to prevent the drill from breaking the part P. Such a naive requirement may be expressed by a rule specifying that the difference "HOLE('radius=R+E')-P" represents an empty set.

Attributes, whether attached to features or not, provide a simple way to query the model. These queries can be done by the user one at a time or organized into search-queries. For example, a process planner may want to know what are the DIAMETERS of all the HOLE features whose attribute ACCESSIBLE has value TRUE. Because these queries may be formulated by an application program, the can constitute and interface between the modelling system and application programs.

Errors in the design are detected by inspecting a particular model (either by hand or by automatically checking attributes). Engineering changes are often formulated in terms of measures of model elements or features. In order to correct errors in a design or to reflect engineering changes, one must correlate model elements or feature measures to the specification commands that are responsible for a particular situation. For example, if a HOLE feature is misplaced, one would like to find automatically which operation positioned the HOLE.

MAMOUR uses the HISTORY information associated with its variables, attributes, features, and elements of extensional models to help the user locate the operations that should be modified. (For example, if the boundary of a slot was disconnected by a drilling operation, the situation will be detected when the validity rules of the slot are evaluated. If this detection is done after the execution of each operation, then the execution can be stopped and the user warned. On the other hand, many such violations may be unimportant, and the user may choose to interrogate the resulting PD at the end of the execution. Checking the validity of the slot in the final PD will indicate that, for instance the bottom of the slot was cut by some boundary element which refers in its history to the DRILLING operation that produced the hole.

The same sequence can be executed on the same raw model for different purposes. If the sequence is to represent manufacturing operations, one purpose of the execution would be to compute the resulting geometry and produce a picture. Another purpose would be to collect tooling requirements and for example compute what are the radii of the holes. This latest query does not require the evaluation of the resulting geometry.

For example, a sequence may contain shape modifying operations (DRILL, MILL, MOVE...) and surface finish modifying operations (SAND, PAINT...). To find what paint colors are required, commands that modify geometry need not be executed. To find what is the volume of the resulting part, surface finishing operations need not be executed. Therefore, each command is associated with several execution routines that correspond to different domains. However, certain parameter or attribute of one domain may be expressed in terms of measures to be derived from other domain. For example, if one wishes to find how much paint will be needed, it is not sufficient to execute the painting operations of a sequence asking them to compute paint quantity. One must also execute the shape modifying operations, so that paining operations can compute the area to be painted.

Reuse of sequences

A sequence can be modified by changing its parameters. MAMOUR provides parameter specification by name-value pair, so that the user only needs to remember the name of a parameter and not its position in the parameter list. For example, "O1('r=3*a');" will produce a copy of O1 where the expression defining of the parameter named "r" is changed no matter how many parameters the sequence O1 may have. A sequence can be edited by deleting or adding operations, and by changing parameter expressions in a command.

The correct version of a sequence can be archived for later use. A library of parameterized sequences defines a set of new operations that makes MAMOUR more powerful and user friendly.

Sequences need not contain the specification of the creation of an **entire** product. They can simply modify existing products in a specific way. For example, FILLET all SLOTs. Sequences that perform such specific transformations can be parameterized in the same way individual operations are, and can be used in place of operations in other sequences.

MAMOUR also provides compound operations which execute a given sequence several times with different parameter values. For example

```
LOOP(SEQUENCE=fillet, NAME=slot, VALUE=all_slots)
```

will apply the FILLET operations to each SLOT in the aggregate variable ALL_SLOTS. MAMOUR also supports conditional operations which take as argument a Boolean condition and an operation, and execute the operation only if the condition is true. For example,

```
IF(CONDITION=depth(slot1) GT 2, THEN=fillet(slot1, r=2))
```

will fillet the feature SLOT1 if it is sufficiently deep.

Sometimes a sequence must be executed on a family of PDs that may have been produced by different sequences and thus contain extensional elements that play similar functional roles but have incon-

sistent names from one PD to another. At this point the user can either preprocess each PD and associate new names with specific elements in a consistent manner, or convert the extensional models into a standard representation, where the default names for elements are consistent with the role these elements play in the model. We used such an approach in the geometric domain by converting CSG extensional models into a different canonical CSG form, which is determined by the topology of the represented pointset. This process generates the same CSG graph structure for two models that have homeomorphic boundary graphs by identifying edge-loops and the inclusion relations among them.

Developing new operations

The amount of programming required to define new operation types has been reduced to a minimum by using techniques that mimicked class inheritance. The programmer must produce

- a declaration of the parameters names, default values, and comments to appear during interactive parameter specification to remind the user of the significance of each parameter.
- a declaration for the internal variables associated with the operation,
- a list of action names for which the operations is executable,
- the methods that will be called when the operation is executed.

We believe that in a given domain, the need for developing new types of operations will decrease significantly when a sufficient set of parameterized sequences, that can be combined to perform new functions, becomes available.

CONCLUSION

An interpreter of a high level object oriented programming language provides a very convenient environment for developing experimental CAD systems of increased "intelligence".

We implemented a domain independent system, called MAMOUR, which, if interfaced with an underlying modeller, supports interactive design, and automatically captures the user's specifications in a sequence of parameterized operations. Such sequences can be: (1) executed to transform different models, (2) edited accordingly to suggestions provided by an analysis of the resulting models, (3) parameterized, (4) archived for later use, and (5) combined with other sequences to define more elaborate transformations. To support our experiments, we have interfaced MAMOUR with a two-dimensional geometric modeller. Features of different types and methods defined for feature-types provide on one hand means for associating abstract concepts with aggregations of elements of the model, and on the other hand high level language constructs for specifying and interrogating the model. Relations between various components of the model can be specified by expressing the parameters of the operations in terms of measures derived from the model. The unevaluated expressions of these parameters capture a generalization of the user's specification, such that a sequence defined interactively to transform a specific model can be executed to transform other models in a manner consistent with the user's intentions.

Constraints imposed by the functional requirements on the model can be stored as validity rules associated with features, and evaluated automatically. To make engineering changes and editing operations more efficient, model elements referenced by rules that are flagged as invalid can be interrogated by the user in terms of the operations that produced them.

A sequence composed of operations that perform actions in different domains may be be executed with respect to a particular domain, and provide a (possibly more concise or discretized) view of the model — or of the sequence — that fits a particular application.

EXAMPLES

In this section we present a few simple examples to demonstrate how MAMOUR is used in the 2D geometric domain.

Notations

The sequence is presented using AML/X notation. The symbol "##" precedes comments on a line. System prompts (upper case) are indicated by "-->" and the user's response (lower case) is provided below the prompt.

Defining a sequence (mb) that builds an example (base1)

The following commented examples illustrate the interactive creation of sequences. The user first declares an empty PD called "base1". Then he declares an empty sequence called "mb". Finally, he invokes a method of the SEQUENCE class, available in MAMOUR, which provides an interactive environment for specifying the operations of a sequence.

```
base1: NEW assembly();        ## creates an empty product documentation

mb: NEW sequence();           ## creates an empty sequence

mb.make(base1,w);             ## to specify mb interactively
                              ## (w is a graphic window)

  --> NAME OF OPERATION OR QUIT
  01

  --> TYPE OF OPERATION (MILLE, DEFATT, SET_TREE...)
  set_tree                     ## to define a CSG tree

       --> DEFINITION OF A NEW TREE
       csg.b2(2,4)+csg.b2(L,H).trans(X,Y).turn(A)

       ## union of two rectangular blocks of sizes (2,4) and (L,H)
       ## the second translated by (X,Y) and rotated by A.
       ## This expression is stored as typed and will only be evaluated
       ## when the sequence is executed

       --> L BECOMES A PARAMETER OF THE SEQUENCE. INITIAL VALUE
       6
       --> H BECOMES A PARAMETER OF THE SEQUENCE. INITIAL VALUE
       2
       --> X BECOMES A PARAMETER OF THE SEQUENCE. INITIAL VALUE
       0
       --> Y BECOMES A PARAMETER OF THE SEQUENCE. INITIAL VALUE
       0
       --> A BECOMES A PARAMETER OF THE SEQUENCE. INITIAL VALUE
       0

  --> NAME OF OPERATION OR QUIT
  quit
```

The resulting model, "base1" is displayed in Figure 4. The new sequence "mb" is listed below, as it appears to the user. Note that the parameter expressions are stored in an unevaluated form, and that the sequence has five parameters, whose default values are indicated.

```
SEQUENCE mb
    PARAMETERS : L=6, H=2, X=0, Y=0, A=0

    OPERATIONS: 01 : SET_TREE (
            NEW_TREE=csg.b2(2,4)+csg.b2(L,H).trans(X,Y).turn(A) )
```

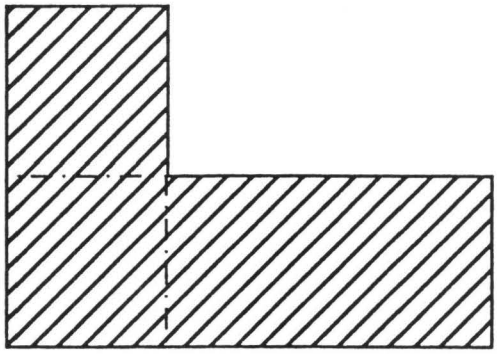


Figure 4. Result on executing MB

Defining, on the example base1, a new sequence (ms)

The result produced by "mb" is used to specify another sequence, called "ms", in a similar manner.

```
ms : new sequence();          ## make a new sequence called ms

ms.make(base1,w);             ## define it on the example base1

  --> NAME OF OPERATION OR QUIT
  01

  --> TYPE OF OPERATION (MILLE, DEFATT, SET_TREE...)
  mille                       ## predefined sequence which mills a
                              ## slot on an edge.

       --> EDGE               ## edge in which the slot will be milled
       $

       ## $ is a special symbol indicating that the edge
       ## will be defined graphically. The user is put in a graphic
       ## mode and uses the cursor to select an edge. The system will
       ## replace the edge by its identification (provenance).
       ## The user could type that provenance directly instead of $
       ## or type an expression that would evaluate to an edge
       ## provenance. For example he could use a procedure called
       ## LONGEST_EDGE(), which computes the identification of
       ## (one of) the longest edge(s). In our example, the user
       ## picked the top horizontal edge of the second block

       --> OFFSET             ## the distance between the beginning of
       long($)/4              ## the edge and the beginning of the slot
       ## the user will be asked to select the edge graphically and $
       ## will be replaced by the appropriate edge identification
       ## Here we picked the same edge as above. To avoid picking
       ## it twice, the user could define a symbol to be that edge
       ## and use it several times in the parameter expressions

       --> DEPTH              ## the depth of the slot
       long($)/2

       ## Here the user picked the right vertical edge of the second block
       ## The length of this edge is equal to the height of the second
       ## block, however, the user need not know about nor have access
       ## to the parameter H of the first sequence

       --> WIDTH              ## the width of the edge
       long($)/2

       ## picked again the same edge as the first picked one

       --> NAME OF THE PRODUCED FEATURE
       'sl1'

       ## the user could supply any expression that evaluates to a name

  --> NAME OF OPERATION OR QUIT
  quit
```

The resulting model "base1" is shown in Figure 5b. On Figure 5a, the arrows indicate the selected edges. The new sequence "ms" is listed below.

```
SEQUENCE ms
    PARAMETERS :
    OPERATIONS : 01 : MILLE (
                      EDGE   = edge_from_csg(R,3)
                      OFFSET= long(edge_from_csg(R,3))/4
                      DEPTH  = long(edge_from_csg(R,2))/2
                      WIDTH  = long(edge_from_csg(R,3))/2
                      NAME   = 'SL1')
```

Note that the graphic selection symbol (wild card) has been replaced by methods for retrieving edges from the product documentation.

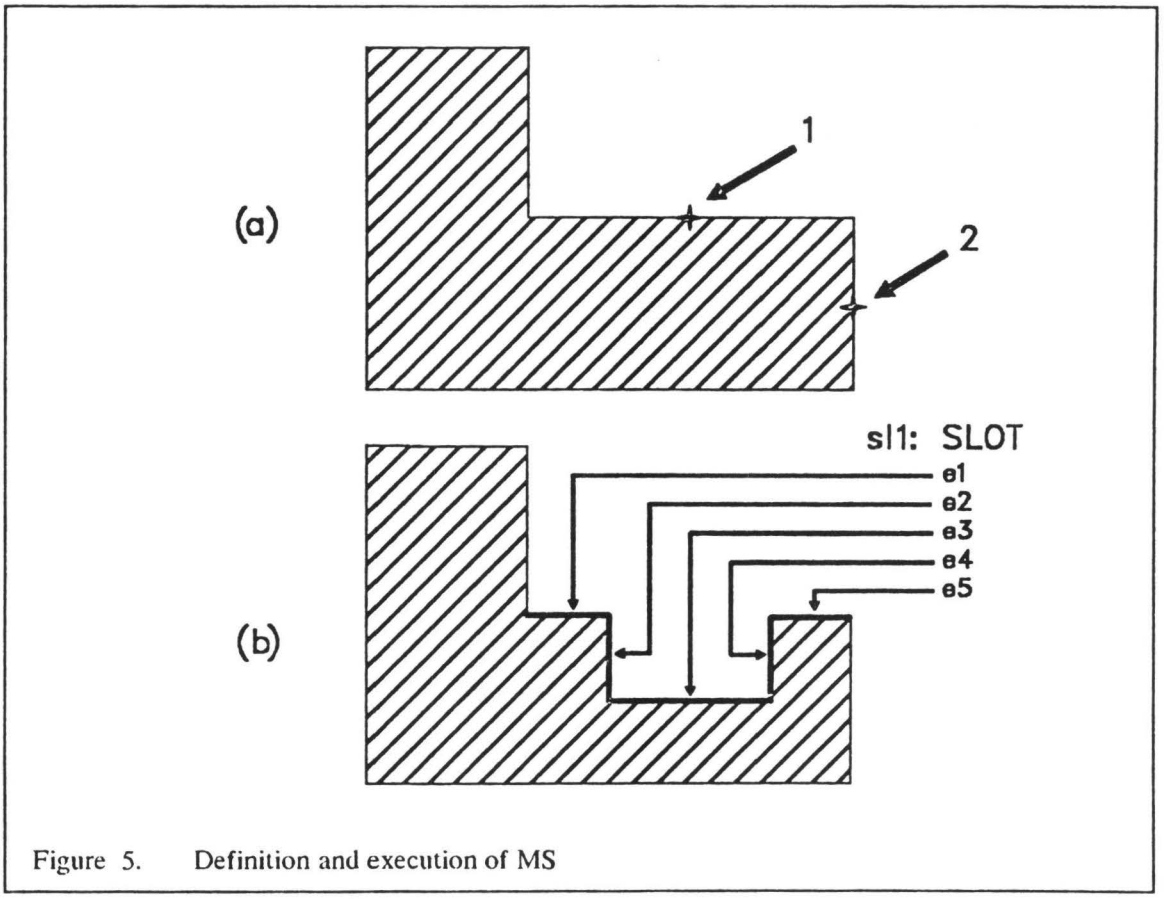


Figure 5. Definition and execution of MS

Creating a new product documentation base2 using mb

To show that the sequence "ms" can be applied to different models yielding consistent results, we first use "mb" with different parameter values to produce a different model, "base2". After what, we transform the result by "ms".

```
base2: NEW assembly();          ## define a new empty product documentation

mb: NEW mb(< 'X=-2', 'Y=1', 'L=7', 'H=1'>);

## changes the default values of three of the parameters
## note that the order of the parameters is not important

mb.exec(base2);                 ## transforms base2 by executing mb
```

The resulting base2 is displayed in Figure 6a.

Executing ms on the new product base2

```
ms.exec(base2);                ## executes ms on base2
```

The resulting "base2" is displayed on Figure 6b.

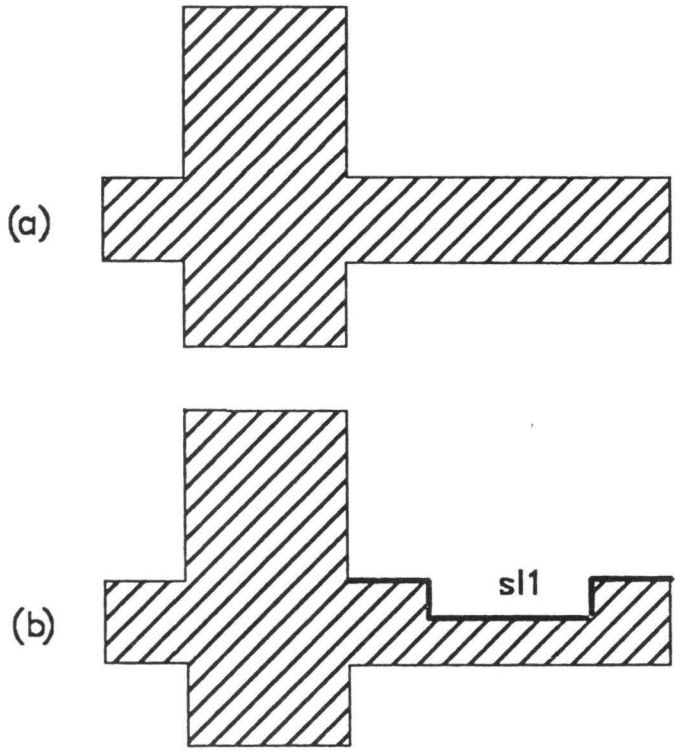


Figure 6. Execution of MB and MS on a new example.

Creating a new product documentation base3 using mb

```
base3: NEW assembly();         ## define a new empty product documentation
Yet a different set of parameter values are used below to produce
a third model.
mb(<'X=-0.5', 'Y=2', 'A=-30', 'L=6', 'H=2'>).exec(base3);

## produces a version of mb with new values of X and Y without
## changing the old version. The new version is executed on base3
```

The resulting "base3" is displayed on Figure 7a.

Executing ms on the new product base3

And again "ms" is used to transform the model.

```
ms.exec(base3);                ## executes ms on base3
```

The resulting "base3" is shown in Figure 7b.

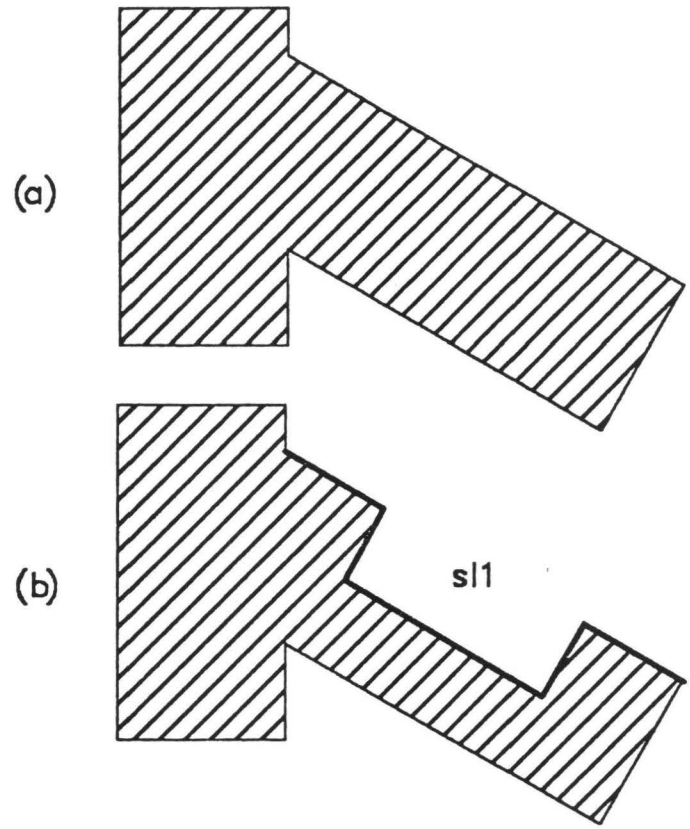


Figure 7. Execution of MB and MS on a new example.

Using names of features

ROUND__SLOT is a predefined sequence, that adds a fillet on a slot. It has two parameters:
SLOT: the name of the slot to be filleted
RADIUS: the radius of the fillet.

The instanciated sequence:

```
ROUND_SLOT ( 'SLOT = sl1', 'RADIUS=depth(sl1)' )
```

can be executed on any product documentation which has a feature named "sl1" of type SLOT.

Figure 8 shows some examples of execution of ROUND__SLOT on different products models.

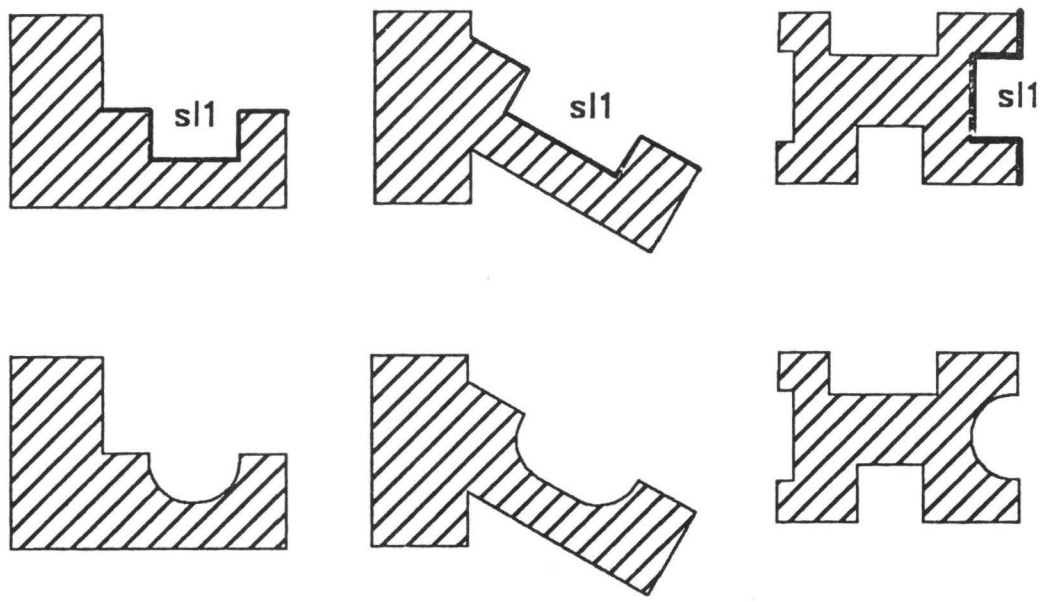


Figure 8. Execution of ROUND__SLOT

Analyzing a product

A sequence SETM, listed below, was created by aggregating two operations: FIND__FEATURE and DEFATT.

```
SEQUENCE setm
    PARAMETERS :
    OPERATIONS : 01 : FIND_FEATURE (
                        TYPE = slot,
                        NAME = 'all_slots'
                 02 : DEFATT (
                        NAME = 'm',
                        VALUE= min(width(all_slots)) )
```

FIND__FEATURE extracts all instances of features of a given type (in this case SLOT) in the geometric model of the product documentation. Rules defining a valid SLOT are associated to the SLOT type. They define the expected shape of this feature (i.e. the number of edges representing a slot and a set of geometric relations among them). FIND__FEATURE successively applies the rules to every set of 5 consecutive edges of the boundary of of the current model in order to locate the occurrences of the shapes they define. All the possible slots are then created and stored, as an attribute, named in this case "all__slots".

DEFATT creates an attribute in the model. The parameter is an expression composed of the MIN operator applied to an aggregate of measures. Each measure is obtained by applying the procedure WIDTH to the features aggregated in the variable "all__slots". The result of evaluating the parameter expression on a given model is the width of the narrowest slot. Figure 9 shows an example of execution of SETM on a given product.

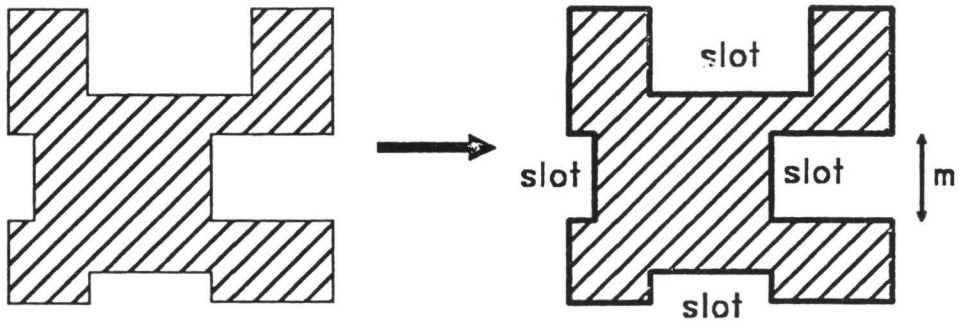


Figure 9. Execution of SETM

Combining sequences

In this example, we are combining the previously defined sequence SETM with an iterative operation, LOOP, that applies to the slots aggregated in the attribute "all_slots" a conditional operation, IF, which in turn applies a sequence ROUND_SLOT (defined elsewhere) to slots that are sufficiently deep.

IF and LOOP permit the conditional and iterative execution of other sequences. The parameters of IF are self explanatory. They are: CONDITION, THEN and ELSE. The parameters of LOOP are:

SEQUENCE: The sequence to be executed several times.

NAME: The name of a parameter of the previous sequence (or of a sequence called by this latter), whose different values are provided in VALUES.

VALUES: Aggregate of values successively taken by the above parameter. The size of the aggregate determines the number of iterations.

The new sequence, called SAMPLE, can be defined as follows:

```
SEQUENCE sample
    PARAMETERS :
    OPERATIONS : 01 : setm
                 02 : loop (
                        NAME     = slot,
                        VALUES   = all_slot,
                        SEQUENCE= if (
                                   CONDITION = depth(slot) GT m/2,
                                   THEN = round_slot(RADIUS=m/2)))
```

Note that the parameter SLOT of ROUND_SLOT will successively take the values aggregated in ALL_SLOT during the iterations.

The sequence SAMPLE fillets all sufficiently deep slots of the model with a radius equal to half of the smallest width to all the slots. Figure 10 shows examples of execution of SAMPLE on different products.

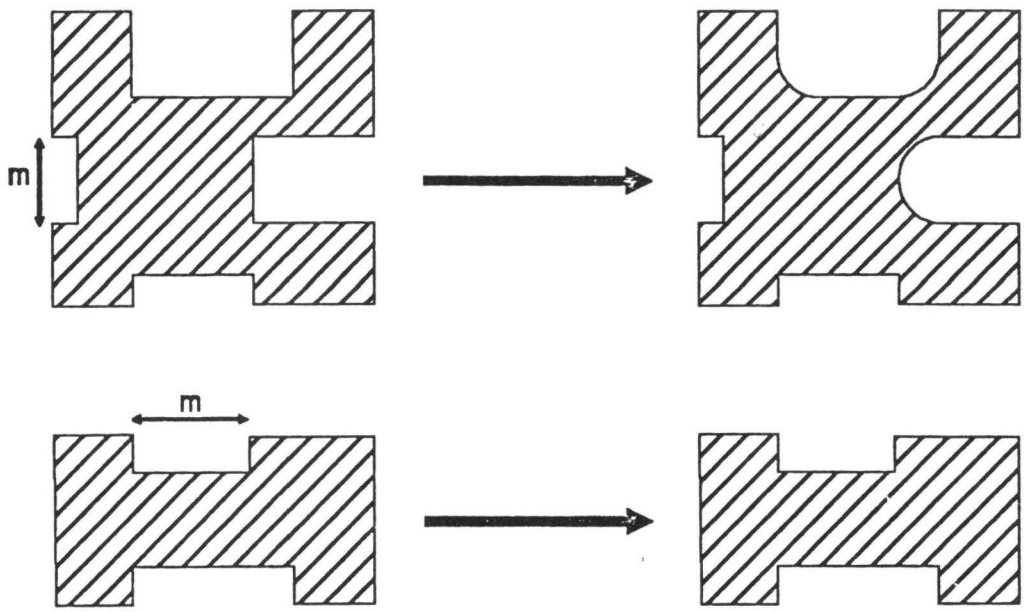


Figure 10. Execution of SETM

REFERENCES

[1] T. Tomiyama and P.J.W. ten Hagen "Representing Knowledge in Two Distinct Descriptions: Extensional vs Intensional" Report CS-R8728, Center for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands, June 1987.

[2] A.A.G. Requicha and H.B. Voelcker, "Constructive solid geometry", Tech. Memo. No. 25, Production Automation Project, Univ. of Rochester, November 1977.

[3] C.M. Brown, "PADL-2: A technical summary", *IEEE Computer Graphics and Applications*, vol. 2, no. 2, pp. 69-84, March 1982.

[4] M. Mantyla and R. Sulonen, "GWB: a solid modeller with Euler operators", *IEEE Computer Graphics and Applications*, vol. 2, no. 7, pp. 17-31, September 1982.

[5] D. Krishnan and L.M. Patnaik, "Design system using an entity-relationship model", *Computer Aided Design*, vol. 18, no. 4, pp. 207-218, May 1986.

[6] L.I. Lieberman and M.A. Wesley, "An automatic programming system for computer controlled mechanical assembly", *IBM Journal of Research and Development*, vol. 21, no. 4, pp. 321-333, July 1977.

[7] M.A. Wesley, T. Lozano-Perez, L.I. Lieberman, M.A. Lavin, and D.D.Grossman, "A geometric modelling system for automated mechanical assembly", *IBM Journal of Research and Development*, vol. 24, no. 1, pp. 64-74, January 1980.

[8] C.M. Eastman, *The design of assemblies*, SAE Technical Paper Series, Society of Automotive Engineers, Inc., USA, 1981.

[9] K. Lee and D.C. Gossard, "A hierarchical data structure for representing assemblies: Part 1" *Computer Aided Design*, vol. 17, no. 1, pp. 15-19, January/February 1985.

[10] R.N. Wolfe, M.A. Wesley, J.C. Kyle, F. Gracer, W.J. Fitzgerald, "Solid Modelling for Production Design" *IBM Journal of Research and Development*, to appear in 1987.

[11] A.A.G. Requicha and S.C. Chan, "Representation of geometric features, tolerances and attributes in solid modellers based on constructive geometry", *IEEE Journal of Robotics and Automation*, vol. 2, no. 3, September 1986.

[12] D.C. Anderson, "Closing the gap: a workstation-mainframe connection", *Computers in Mechanical Engineering*, vol. 4, no. 6, pp. 16-24, May 1986.

[13] E.A. Bier, "Skitters and Jacks: Interactive 3D positioning tools", proceedings 1986 Workshop on Interactive 3D Graphics, University of North Carolina, Chapel Hill, NC27514, F. Crow and S.M. Pizer Ed., ACM Press, pp. 183-196, October 23-24, 1986.

[14] I. Sutherland, "Sketchpad, a man-machine graphical communication system", PhD thesis, MIT, January 1963.

[15] G. Nelson, "Juno, a constraint-based graphics system", *SIGGRAPH'85*, vol. 19, no. 3, pp. 235-243, San Francisco, July 22-26, 1985.

[16] R.C. Hillyard and I.C. Braid, "Characterizing non-ideal shapes in terms of dimensions and tolerances" *ACM Computer Graphics*, vol. 12, no. 3, pp. 234-238, August 1978.

[17] V.C. Lin, D.C. Gossard, and R.A. Light, "Variational geometry in computer aided design", *ACM Computer Graphics*, vol. 15, no. 3, pp. 171-177, August 1981.

[18] A.P. Ambler and R.J. Poppelstone, "Inferring the positions of bodies from specified spatial relationships" *Artificial Intelligence vol. 6, pp. 157-174, 1975.*

[19] K. Lee and G. Andrews, "Inference of the positions of components in an assembly: Part 2." *Computer Aided Design*, vol. 17, no. 1, pp. 20-24, January/February 1985.

[20] 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.

[21] F. Arbab and J.M. Wing, "Geometric reasoning: A new paradigm for processing geometric information", Report TR-85-333, Computer Science Department, University of Southern California, Los Angeles, July 1985.

[22] B. Bruderlin, "Constructing three-dimensional geometric objects defined by constraints", proceedings 1986 Workshop on Interactive 3D Graphics, University of North Carolina, Chapel Hill, NC27514, F. Crow and S.M. Pizer Ed., ACM Press, pp. 111-130, October 23-24, 1986.

[23] F. Kimura, H. Suzuki, and L. Wingard, "A uniform approach to dimensioning and tolerancing in product modelling", *Proc. CAPE'86*, 1986.

[24] M. Managaki and K. Kawago, "Parametric man/machine interactions with semantic data", *Computer Graphics*, vol. 7, no. 3-4, pp. 233-242, 1983.

[25] J.R. Rossignac, "Constraints in Constructive Solid Geometry", proceedings 1986 Workshop on Interactive 3D Graphics, University of North Carolina, Chapel Hill, NC27514, F. Crow and S.M. Pizer Ed., ACM Press, pp. 93-110, October 23-24, 1986. Also available as IBM Research Report RC 12356, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, New York, September 1986.

[26] M. Cutkosky and J. Tenenbaum, "CAD/CAM integration through concurrent process and product design", submitted to the ASME Symposium on Intelligent and Integrated Manufacturing, Winter Annual Meeting, December 1987.

[27] G.M. Koppelman and M.A. Wesley, "OYSTER: A study of integrated circuits as three-dimensional structures" *IBM Journal of Research and Development*, vol. 27, no. 2, pp. 149-163, March 1983.

[28] G.E.M. Jared, "Shape features in geometric modelling", M.S. Pickett and J.W. Boyse, Eds., *Solid Modelling by Computers.* New York: Plenum Press, pp. 121-137, 1984.

[29] Y. Yamaguchi, F. Kimura, and P.J.W. ten Hagen, "Interaction management in CAD systems with history mechanism", proceedings of *EUROGRAPHICS'87*, G. Marechal (North-Holland), pp. 543-554, August 1987.

[30] P. Borrel, J.R. Rossignac, L.R. Nackman, "MAMOUR: A system for modeling assemblies and manufacturing operations by unevaluated representations", IBM Research, Yorktown Heights, NY, in preparation.

[31] L.R. Nackman, M.A. Lavin, R.H. Taylor, W.C. Dietrich, and D.D. Grossman, "AML/X: a programming language for design and manufacturing", *Proceedings of the IEEE Fall Joint Computer Conference,* Dallas, Texas, pp. 145-159, November 2-6, 1986.

[32] J. Cagan and A.M. Agogino "Innovative design of mechanical structures from First Principles", working paper 87-0801-2, Intelligent Systems Research Group, Department of Mechanical Engineering, University of California at Berkely, December 1987.

[33] J.R. Rossignac, "AML/X tools for primitive geometric calculations: Points, Vectors, Coordinate Frames, and Linear Transformations", IBM Report RA 189, Design Automation Group, IBM, T.J. Watson Research Center, Yorktown Heights, NY 10598, May 1987.

A Definitive Programming Approach to the Implementation of CAD Software

M. Beynon
A. Cartwright

A definitive programming approach to the implementation of CAD software

Meurig Beynon
Alan Cartwright

Departments of Computer Science and Engineering
University of Warwick
Coventry CV4 7AL

Abstract

This paper outlines an approach to the implementation of CAD systems that makes use of a programming paradigm based upon definitions ("definitive programming").

It departs from previous research on "pure definitive notations" - special-purpose notations for interaction - and proposes a general-purpose programming model based upon definitive principles. This model is examined as a possible basis for the development of an integrated framework within which to address the broader issues of a design support environment, including constraint handling and user-interface management. This gives a new perspective on the use of definitive principles for interaction in which the emphasis is upon interpreting a family of definitions as one of many possible "intelligent views" of an interactive system. It also establishes a closer relationship between the definitive programming approach to CAD and the study of CAD from an AI perspective than was previously evident.

The design of an appropriate definitive notation for geometric modelling is a fundamental aspect of the application of definitive principles to CAD software. An appropriate basis for such a notation is presented in an Appendix.

A definitive programming approach to the implementation of CAD software

Meurig Beynon
Alan Cartwright
Departments of Computer Science and Engineering
University of Warwick
Coventry CV4 7AL

Keywords: computer-aided design, human-computer interaction, user-interface management, geometric modelling

This paper - a sequel to [2] - outlines an approach to the implementation of CAD systems that makes use of a programming paradigm based upon definitions ("definitive programming"). There are two principal aspects to this research:

(a) the design of an appropriate definitive notation for dealing with the complex geometric problems that are centrally important in CAD applications;

(b) the development of an integrated framework within which to address the broader issues of a design support environment.

The paper is in two parts: an extended account - written by the principal author - of the research so far carried out on (b), and a jointly authored Appendix providing further details of (a).

Of the two aspects, the first is a direct extension of previous work on pure definitive notations for interactive graphics [2]. In contrast, the work that has been done under (b) - though consistent with the view of "dialogue over a definitive notation" as an appropriate intermediate code for user-computer interaction (cf [2]) - involves a significant development of the definitive programming paradigm, and introduces considerations far beyond the scope of [2]. These lead in particular to a new perspective on the use of definitive principles for interaction in which the emphasis is upon interpreting a family of definitions as one of many possible "intelligent views" of an interactive system. This appears to establish a much closer relationship between the definitive programming approach to CAD (cf §5) and the study of CAD from an AI perspective (cf [17]) than was previously evident.

The concept of using definitions as a basis for interactive software was advocated in [1], and is illustrated in the design of the ARCA and DoNaLD graphics systems [3,4]. The merits of using definitive principles as a basis for software for interactive graphics are described in detail in [2]. In particular, the advantages in respect of data representation and abstraction over traditional procedural or purely declarative programming paradigms are explained. It would be misleading to suggest that the application of these principles to the design of graphics for a CAD system is trivial however. This is illustrated by the design of an appropriate definitive notation for geometric modelling, as outlined in §1.2. The problems encountered in this generalisation have such interest and relevance for the entire project as to merit supplementary consideration in an Appendix.

The framework described in [2] is also limited in several other respects. A key idea behind definitive notations is that the current status of a user-computer interaction ("the state of dialogue") is effectively represented by a system of definitions resembling in essence the system of functional relationships between scalar quantities underlying a naive spreadsheet. Such a view of interaction is oriented towards a passive use of the computer in which the responsibility for changing the state of dialogue rests upon the user. In practice, there are many important issues in the design and implementation of CAD systems that demand a more general framework. Valuable though the passive use of sophisticated data description is in the early stages of the design process [20], there is also a very significant role for interaction in which the computer plays an active part (cf [21] p8). Traditional user-interface management issues come to mind in this context [12,14]: the animation of a dialogue through appropriate use of windows, menus, graphical displays, and the use of analogue rather than textual input. Of equal importance are issues such as the monitoring and maintenance of constraints [8,18,22,25]; an activity in which the computer must itself participate in

changing the state of dialogue.

The purpose of this paper is to argue that the essential concept of definitive programming (viz the representation of a state of dialogue by means of a set of definitions that is transparent to the user) can be developed to support the broader concerns raised by the implementation of sophisticated CAD systems. Some justification for regarding CAD systems based upon definitive principles as well-suited for the implementation of "intelligent" CAD software is also given. Indeed, the thesis that computer-aided design systems can be very effectively implemented using definitive principles in such a way that the user has a powerful abstract view of the stages of the interactive design process itself suggests a possible interpretation of "intelligent" computer aided "design".

The paper is divided into 5 sections.

§1 reviews some of the principal ideas discussed in [2] and gives a brief outline of a definitive notation suitable for geometric modelling combining some of the characteristics of both the ARCA and DoNaLD notations. The aim is to illustrate explicitly how the ideas in [2] can be applied to the description of geometric objects in many semantically different ways: as abstract complexes of simplices, as frames comprising finite configurations of simplicial elements, and as realisations of such frames as geometric objects defined by sets of points in space (cf [9,26]). In effect, the state of dialogue over such a notation is interpreted as a formal representation of the user's current model of a geometric object, and the design process is viewed as a process of refinement of this model associated with a sequnce of transitions through different dialogue states.

§2 focusses on some limitations of this "pure definitive notation paradigm" for design as a basis for developing CAD system software.Two principal issues are considered

1) the need in general for a framework of constraints within which the design dialogue must operate (§3),

2) the relationship between the semantics of the geometric design dialogue and the mechanics of the interaction between the user and the computer (§4).

Where 2) is concerned, it may be seen that no provision is made within the pure definitive notation design paradigm for representation of the current state of the device supporting the user-computer interaction. This is in some sense appropriate, in that the abstract design process has meaning independent of the ephemeral state of the screen during some specific design transaction; on the other hand, there is in general a complex interrelationship between the underlying semantics and the current state of the display interface. A major step towards the solution of this problem is the introduction of an auxiliary definitive notation within which to specify the screen display at any stage. The form of the user interface can then be specified as a set of definitions, either by the system or by the user, and subsequently driven by dialogue actions initiated by the computer.

An underlying principle of definitive programming, relevant to both 1) and 2), is that the way in which a transition between dialogue states is effected is not of primary significance; what is crucially important is that the current state of dialogue can be subtly represented, and is at all times transparent to the user. To exploit this principle fully, a richer programming paradigm is required, in which - in the conceptually simplest framework - several processes may participate in the design dialogue (see [5], and compare [16]). To this end, an extended form of definitive programming is described, incorporating definitions and user-defined functions (as in a pure definitive notation) together with actions that - in an appropriate dialogue state - are invoked to change this state (§2.2). §'s 3 and 4 are concerned with the way in which such a computational framework may be used to handle constraints and aspects of the user-interface management in a unified manner, and indicate some of the anticipated advantages of a consistent use of definitive principles.

The final section of the paper discusses the potential merits of a definitive programming approach to the development of an "intelligent" CAD system. A central theme is that the interpretation of a family of variable definitions as an "intelligent view" of an interactive system is a powerful abstraction implicit in many "intelligent" CAD systems based on more traditional paradigms.

§1. Applying definitive principles to CAD

In this section, the basic principles of definitive programming are briefly reviewed, and the design of a definitive notation that has been developed for geometric modelling is outlined.

1.1 Basic principles

A general account of the merits of using definitive principles for interactive graphics is given in [2], to which the interested reader is referred for further background. For convenience, a brief review of the basic concepts and terminology will be given here.

A **definitive notation** is specified by an underlying algebra comprising a set Δ of data types and a family Σ of operators that take the form of pure functions mapping between the data types. The underlying algebra is complemented by variables, whose types are in Δ, and whose values can be defined by algebraic expressions in terms of other variables and explicit values via the operators in Σ. Programming over a definitive notation consists of introducing a system of definitions of variables, and thereafter redefining variables or inspecting their current values. Such a style of programming is particularly well-suited to capturing the semantics of a user-computer dialogue [1]. In that context, the current system of variable definitions is referred to as "the state of the dialogue", and the computational step involved in the redefinition of a variable as "a dialogue action". Allowing the user to program autonomously using variable definitions to capture the current dialogue state is the simplest mode of definitive programming, and requires nothing more sophisticated than a pure definitive notation. In this paper, the same terminology will be adopted for a more general form of definitive programming, in which changing the dialogue state is not the prerogative of the user alone.

The basic technical problems in designing definitive notations, such as the methods to be used when defining variables of complex types that may represent values in many different abstract ways, have been quite fully researched and described elsewhere [2]. An account of the way in which definitive notations can be used to capture many different kinds of abstraction, as illustrated by the specific definitive notations for interactive graphics DoNaLD and ARCA, appears in [2]. The motivation behind the current research on definitive programming is to examine how far the principles developed in the study of pure definitive notations can be applied more generally. In part, this involves identifying those aspects of pure definitive notations that have yet to be exploited. For instance, the introduction of an abstractly defined underlying algebra - incorporating axioms governing the operators - will make it possible to use symbolic manipulation techniques for processing and evaluating definitions. Similarly, greater use of traditional functional programming techniques, such as the introduction of more sophisticated data types, and of higher-order functions as operators, will enhance the expressive power of definitions. To an extent, this paper will indicate respects in which such enhancement of a pure definitive notation can support more sophisticated applications (cf §1.2 below), but it will also be concerned with issues that cannot conveniently be dealt with within the pure definitive notation paradigm.

Each programming paradigm has its own characteristic approach to achieving computational abstraction. In functional programming, a program is regarded as being nothing more than the evaluation of an appropriately defined function, and the user is not encouraged to introspect about the computational process involved in this evaluation. In definitive programming, functional relationships between variables are viewed in a very similar manner: when one variable is functionally defined in terms of others, and its value is implicitly altered through a dialogue action, the mechanism used to update values is deemed to be irrelevant. In effect, any computation that might be involved in carrying out a transition between dialogue states is outside the programming model - or more appropriately in the context of this paper, outside the view of the active agent; the

values of variables are at all times assumed to be consistent with their definition. Updating a spreadsheet is the simplest illustration of this principle in action; the user ideally demands an interface that updates displayed values "instantaneously". In the context of the definitive notation for geometric modelling described below, such idealisation is clearly much further removed from realism - it is in general hard to update a display of a parametrised geometric model in real-time! At some level of abstraction, it is nonetheless appropriate to invoke such idealisation, and possibly to treat other aspects of the user-computer interaction in which computation is involved in a similar manner. It is equally necessary to understand how such an abstract perspective can be practically helpful when building a complete system for geometric modelling, and this concern supplies the context for the study described in §'s 2,3 and 4 below.

1.2 A definitive notation for geometric modelling

The use of a definitive notation for the representation of geometric models is central to the definitive programming approach to CAD software. Designing an appropriate notation is a major part of the conceptual design of the user-interface (cf [12] and §4 below). Pursuing the ideas in [2], it is to be hoped that definitive principles can in due course support feature-based description in parallel with geometric description, thereby avoiding the need for feature extraction for process planning (cf [21],p3). To achieve this, it is not enough to adopt a particular representation for geometric objects; it is necessary to develop ways of integrating many different characteristics of geometric models within a single unifying framework. This is a non-trivial task; as observed in [21], surface and solid models have yet to be successfully integrated in conventional CAD systems.

The relevant characteristics of geometric models include information about reference and construction points, labels, skeletal structure, and those geometric operations that are typically used to synthesise complex objects from simple components. The underlying algebra accordingly includes several different sorts of data, both combinatorial and geometric, together with a system of algebraic operators that can be used to establish relationships between these sorts. The variables over the associated definitive notation can then be used to define models of complex geometric objects, perhaps incorporating components specified using different modelling paradigms, and possibly incompletely specified.

The data types used to represent geometric objects have also been designed to capture the distinction between an abstract object, such as "the sphere with centre c and radius r", and a specific sphere, for which explicit values for c and r are known. The use of a definitive notation lends itself to the representation of such distinctions between partially instantiated and explicit objects, but in a manner that puts too much onus on the user to establish and maintain functional relationships. Using techniques resembling those introduced for moding variables in the definitive notation ARCA [2] it is possible to declare variables to represent objects of a generic form. This obviates the need for the user to repeatedly set up appropriately contrived systems of variables to define explicit objects within the same abstract class. In this way, the designer is able to describe an object in an internally consistent fashion without referring to its location in space, or physical dimensions. This solution to the problem of representing objects is similar in spirit to the use of classes in an object-oriented programming framework.

In essence, the underlying algebra incorporates three distinct sorts for describing geometric objects: **complex, frame** and **object**. The **complex** is a combinatorial structure - resembling the abstract simplicial complex of polyhedral topology - intended to capture the purely combinatorial ingredients of a geometric object. These include: reference points, abstract scalars needed to specify lengths and angles etc, and incidence information expressing the way that the object is abstractly synthesised from simpler components. To derive a **frame** from a **complex**, it is necessary to specify the dimension of the space in which the complex is to be realised, and to supply specific coordinates and scalar parameters corresponding to the abstract labels of the **complex**. An **object** is generally determined by a list of **frame**s, together with a function that takes the parameters of these **frame**s as arguments and returns the extent of the object. (More details of these data types are given in the Appendix.)

The motivating idea behind the choice of sorts is that an **object** is to be viewed as the realisation of a combinatorial structure, as represented by an underlying list of **complex**es. For instance, a spline is determined by a wire-frame together with an appropriate set of boundary elements. The wire-frame has two ingredients: a combinatorial structure, consisting of an array of labelled points with incidence relationships between them, and an associated array of coordinates. By specifying how the spline is abstractly defined in terms of the frame and the boundary elements, without regard for their specific coordinates and scalar values, the spline can be specified as an abstract object. That is to say, the abstract description of the spline takes the form of a function that takes as its arguments an appropriately typed list of **frame**s required to specify the wire-frame and its boundary and returns "the set of points that comprise the spline". By subsequently supplying parameters for such a function ie specifying a suitable explicit list of **frame**s, a spline is derived as an explicit object.

The design of variables for the definitive notation broadly follows the patterns established in ARCA and DoNaLD [2]. In particular, there is a concept of a variable **mode** - as introduced in ARCA - and of a subvariable - as in the DoNaLD **openshape**. (See the Appendix for more details.) There are also some significant unprecedented features. For instance, an "abstract object" is effectively a component of the **object** data type, so that appropriately moded **object** variables can be regarded as representing operators to realise **object**s from **frame**s.

Specifying the **object** data type also presents some novel problems. Since there is no reasonable general method of specifying an object explicitly (ie by enumeration of its points), the extent of an object is represented by a criterion for membership ie a function **vector_of real** → **boolean**. In practice, the relationship between such specifications of objects and the algorithms used to display and manipulate objects may be very complicated. For instance, it may not even be obvious how to apply the criterion for membership of an object in general, and there will be problems of numerical approximation to address [10,13]. There are several pertinent issues to be considered. It is anticipated that **object**s will generally be defined in terms of standard abstract objects (such as spheres or polyhedra) that are supported by built-in functions for display and manipulation in a solid-modelling system. There will be an important role for symbolic manipulation and transformation techniques in relating the abstract definition of a geometric object and its image in the display interface as conceived in §3 below. At the same time, the approach proposed here offers some advantages where the traditional problems of evaluation are concerned, since it is based upon the symbolic representation of objects.

§2 Generalising the definitive programming paradigm

This section examines the limitations of pure definitive notations, and indicates the most promising direction for generalisation of the definitive programming paradigm.

2.1 Limitations of the pure definitive notation paradigm

The use of a pure definitive notation for user-computer dialogue puts the primary emphasis upon representing the current state of the interaction by means of a system of definitions. In practice, this proves to be very effective in as much as the user can readily determine the current state of the dialogue at any time, and can predict the effect of any dialogue actions. However, the use of definitions can also be unnatural, since it requires an acyclic system of functional dependencies between variables.

If the criterion for a good representation of the state of an interaction is "predictability of response to dialogue actions", the restriction to acyclic systems of functional dependencies is superficially unnecessary. It is not even necessary to consider complex constraint relationships (cf §3) to appreciate this. For instance, it may be that two variables **x** and **y** are to be assigned the same value in such a way that if the value of one is changed, then so also is the other. This can be done within

the definitive programming framework - for example, by introducing an auxiliary variable **t**, together with the definitions: **x** = **t**, **y** = **t** - but only in such a way that an attempt to redefine **x** or **y** is construed as a redefinition of **t**. This solution can seem particularly uncomfortable in the context of interactive graphics, where it is congenial to use analogue input via the graphics interface, and direct reference to the internal definitions of variables may not be appropriate. These are the considerations that suggest that a pure definitive program is best viewed as a form of intermediate code for interaction [2].

Appropriate definition of variables can sometimes provide an alternative to hidden parametrisation, but leads in general to a form of over-specification that is somewhat inelegant, and puts an obligation upon the user to recall - or, to be more precise - to refer to knowledge of the representation. For instance, in DoNaLD, it is an easy matter to represent a unit square by defining **point** variables **a**,**b**,**c**,**d** to represent its four vertices subject to the relations:

b = **a**+[0,1]; **c** = **a**+[1,1]; **d** = **a**+[1,0].

To translate the square to another location whilst preserving its orientation, it suffices to redefine the variable **a**, but this is information that is hidden if the user sees only the display.

Such use of functional relationships contrasts with the use of equational relationships between variables commonly employed in a constraint-based graphics system. There are many issues to be examined here (see §3 below): the kind of enhancement of pure definitive notations that can support constraint-processing; the extent to which constraints can be accommodated within the definitive programming paradigm; the significance of representing a constraint using functional relationships.

The passivity of the pure definitive notation paradigm has other important ramifications. In practice, the implementation of an interactive system involves many dynamic elements required for the animation of the dialogue, such as windows, menus and graphical displays. The definitive notation for geometric modelling described in §1.2 can capture the stages of the abstract geometric design process effectively, but does not address the more immediate, if perhaps more ephemeral issues, concerning the current "state of the interaction" in its broad sense. These include considerations such as what windows are currently open, what menu options are currently available, and the form of the responses to input.

There are two aspects to this problem. On the one hand - as is argued in §'s 3 and 4 below - there is good reason to suppose that a system of definitions can be used to represent the entire suspended state of the interface during an interaction effectively. On the other hand, the transitions between dialogue states that are required to model the interface in this manner are very complex, and are in general to be carried out by the computer rather than by the user. To solve this problem, it is first necessary to re-examine the principle of "programming with definitions" to determine how several agents can be allowed to participate in a dialogue. A fuller discussion of the specific implications for user-interface management within a definitive programming paradigm appears in §4 below.

By implication, the generalisation of definitive programming envisaged differs very radically from the elementary use of pure definitive notations. Indeed, to handle the issues of user-interface management within a coherent framework of evolving systems of definitions requires a powerful and general programming paradigm far removed from the special purpose "definitive notations for interaction" described in [2]. An important concern in this paper is to motivate the introduction of a general purpose definitive programming paradigm - after all, other idioms, such as functional or object-oriented programming, have also been advocated for general purpose programming, and provide techniques particularly well-suited for operator specification and data abstraction. This paper argues that the distinctive feature of the definitive programming approach is effective support for the representation of interaction, whether internal or external to a system.

2.2 Enhancing the pure definitive notation paradigm

The method of enhancing pure definitive notations proposed here is based upon abstraction from a mixed programming paradigm first introduced in the EDEN interpreter - a software tool designed as

"an evaluator for definitive notations" [6]. The EDEN interpreter has built-in support for a definitive notation based upon list processing, but can also be programmed to perform traditional procedural actions that may be synchronised with changes in the dialogue state using triggering mechanisms resembling those used in object-oriented systems [8]. By translating definitions into the internal definitive notation, it is easy to represent the state of dialogue over any definitive notation. By using triggered actions, it is easy to make responses contingent upon the current state of the dialogue eg to ensure that a particular screen location in a spreadsheet is to be updated whenever the corresponding variable value is changed, or to register the presence of error conditions such as a constraint violation. EDEN has already been successfully used for the rapid prototyping of software based on definitive principles, including a prototype DoNaLD implementation [6], and is the practical programming tool that is currently being developed to support more ambitious CAD implementation.

In effect, EDEN makes it possible to link complex procedural actions and intricate systems of definitions: a very powerful mixed programming paradigm, but one that can also prove difficult to use and analyse. There is clearly a need to guarantee that triggered actions do not interfere, for instance, and to avoid infinite behaviour. For this reason, triggering has primarily been used to implement actions that have no effect upon the current state of the dialogue as represented by the internal system of variable definitions. Such limited use does not adequately support the direct participation of the computer in the user-computer dialogue that is required for general constraint and user-interface management however. The general definitive programming paradigm to be introduced to this end (cf §'s 3 and 4 below) can be most succintly explained in terms of an abstract machine model that can realise the capabilities of the EDEN interpreter without compromising clarity.

In outline, this abstract machine model consists of three components: a program store P, comprising a set of *entities*, a store D of variable *definitions*, and a store A of *actions*. Each entity comprises an abstractly specified block of definitions and actions, perhaps parametrised, that is superficially analogous to the declaration of a procedure in a conventional procedural language, or of an object in an object-oriented language. The variables whose definitions appear in D may be assumed to be of some basic data type, such as **integer**s or **list**s. At all times, the system of functional dependencies between the variables in D is acyclic, and the value of each variable is consistent with its definition. An action takes the form of a guarded sequence of instructions, each of which either redefines a variable, or invokes the introduction or deletion of a block of new definitions and actions into the stores D and A through the instantiation or elimination of an entity.

A computation consists of a sequence of parallel executions of appropriate actions. In a single computational step, the guards of all actions in the action store are evaluated, and the actions associated with true guards executed in parallel. This in general has the effect of changing the contents of the stores D and A by modifying the definitions of variables (possibly even those whose value is implictly defined by a formula), and may also lead to the introduction or deletion of blocks of definitions and actions. To admit redefinitions involving the evaluation of implicitly defined variables (as is appropriate for instance when referring to the current value of an attribute of an implicitly defined object), there is a mechanism for the evaluation of specified expressions in the same context in which the evaluation of guards is carried out.

In interpreting a user-computer dialogue within the framework of the abstract machine model, the roles of the user and of the computer are determined by entities that are instantiated according to the current context (cf §5 below). There will at any time be variables instantiated to represent new user-input via the keyboard, and to record the present position and status of the mouse, for instance. The abstract machine model then provides a framework that retains the characteristic feature of definitive programming viz the representation of the current state of the user-computer interaction by means of a system of variable definitions, but allows both the user and the computer to initiate dialogue actions to change this state.

Clarifying the abstract machine model is the first step towards the long-term objective of developing

CAD support systems that do not rely upon a semantically complicated mixed programming paradigm. The implications of using the "extended definitive programming idiom" for constraint and user-interface management in CAD applications are examined in more detail in §'s 3 and 4 below, but some of the motivating ideas behind the design of the abstract machine model above may be helpful. *Definitions* are the counterparts at a low-level of abstraction of the high-level definitions that establish functional relationships between variables that represent - for example - complexes of labels, geometric objects and screen displays (cf §1.2 and §4). *Actions* enable autonomous response on the part of the computer, as - for example - when undoing a user-dialogue action that leads to the violation of an imposed constraint, or when automatically invoking a new dialogue context for the user. *Entities* are introduced to meet the need for systems of variable definitions and associated actions that are to be temporarily instantiated as a unit. When declaring an object, an associated family of definitions to describe the associated screen display is required (cf §4), together with actions that are required eg to ensure that specified constraints upon components of the object (cf §3) are imposed.

§3 Constraints and constraint management in the definitive programming model

Constraints play a very important role in computer-aided design. The designer often needs to work in a context where a large number of interdependent constraints have to be met. The functional relationships between variables established using a definitive notation can be very helpful in imposing particular constraints, but there is a need both for additional techniques and for a better understanding of how such functional relationships are connected with general constraints. The purpose of this section is to examine how far constraints and techniques for constraint management can be accommodated within the abstract definitive programming model.

In the first instance, it will be convenient to consider to what extent constraints and systems of variables can be directly integrated, and to explore some elementary techniques that can be used to introduce constraints into the definitive programming model. In general, a dialogue action may have the effect of changing variable values in such a way as to violate a constraint. Even in the pure definitive notation paradigm, such violations can be monitored by using by introducing string variables to flag error conditions. Thus, if C is a constraint condition, then

E = **if** C **then** NULL **else** "C is violated"

defines an appropriate variable E to represent the error status. By displaying E in an appropriate fashion eg in a pre-determined field of an error monitor window, it is possible to carry out a rudimentary form of automatic constraint monitoring. In effect, the user is able to observe the consequences of adverse design decisions in so far as these can be represented using chains of dependent variables.

In the extended definitive programming framework, in which the computer can act to change the state of the dialogue, it is possible to develop this idea further, and provide constraints, that cannot be violated by a user action. To achieve this it is only necessary to set up a protocol whereby a user dialogue action leading to the violation of a specified constraint is automatically revoked. A yet more ambitious objective is the management of constraints in the spirit of a traditional constraint-based system: if the user performs an action that violates a specified constraint, then a predetermined sequence of redefinitions is initiated automatically until the constraint has been restored.

These techniques will be referred to as **monitoring**, **imposing** and **maintaining** a constraint. Both imposing and maintaining constraints require a user protocol that restricts the user's capability for action whilst the constraint is violated. Maintaining constraints of course additionally requires domain specific knowledge, but can be based around techniques that have been well-studied in other contexts [8], suitably adapted. As pointed out in [8], constraint maintenance calls both for declarative information ("what constraint must be met") and procedural information ("how is it to be maintained"). Though it might superficially appear appropriate to view the functional relationships established by definitions as a declarative formulation of a constraint, it will emerge later that they more fruitfully be regarded as concise ways of specifying the methods by which

constraints are maintained.

The critical reader will recognise that the above discussion fails to make a clear semantic distinction between general constraints and "functional relationships established through variable definitions". This is a naive perspective. As illustrated in §2 above, the specification of functional relationships that establish a particular constraint may involve an artificial choice of parametrisation that is aesthetically disturbing. What is more, constraints typically entail more complex relationships between variable values than can be conveniently expressed using definitions alone. As explained below, it is probably much more reasonable to regard a system of definitions as encapsulating one particular agent's view of how a given constraint is maintained (cf the methods considered in [8] §3.2). The techniques described above then become part of a more sophisticated framework for constraint management, in which there need be less emphasis upon constraint maintenance.

To illustrate the latter problem, consider a set of variables

point x, y, z, o; real a, b, c, r

constrained in such a way that **o** is the centre of the circle of radius **r** passing through the points **x**, **y**, **z** with polar coordinates (**r**, **a**), (**r**, **b**) and (**r**, **c**) respectively (see Fig 1). In a constraint-based graphics system, these constraints might be established in such a way that moving the point **o** translates the entire Figure parallel to the axes, that modification of **r** causes the circle to expand or contract so as to respect similarity and preserve the orientation of the triangle **xyz**, modification of the angles **a**, **b** and **c** leads to appropriate relocation of **x**, **y** and **z** on the circle, and relocation of **x**, **y** or **z** causes relocation of the circumcentre **o** of the triangle **xyz** and the associated redefinition of **a**, **b**, **c** and **r**. To represent the rich set of functional relationships implicit in these recipes for constraint maintenance is beyond the scope of an acyclic system of variable definitions. To adequately represent all the geometric transformations involved requires - for instance - functional definitions for **o**, **a**, **b**, **c** and **r** in terms of **x**, **y** and **z**, and definitions for the inverse relationship.

To describe such an interactive environment within the extended definitive programming framework requires a more careful appraisal of the significance of "definitions for interaction", and introduces new concepts to be elaborated in §'s 4 and 5 below. To appreciate the need for a new perspective, it must be remembered that the model of interaction required for extended definitive programming has to take account of several process views. The key idea is that a system of variable definitions can profitably be regarded as representing an interpretation (or perhaps even an "intelligent view") of an object, as observed and possibly manipulated by an agent participating in the dialogue. To be more explicit, the system of functional relationships between a set of variables perceived by a particular process encodes:

(a) the current state of an object,
(b) knowledge of how it can be modified,
(c) what the effect of such a modification will be.

For the object in Figure 1, there are several process views, corresponding to the various ways in which the object is modified by different actions on the part of the user. In this sense, the user acts in the role of several processes - as "the user who points at the centre of the circle" or the user who points at the point **x** on the circumference" etc. Within the extended definitive programming paradigm, this role changing on the part of the user is supported by the user-interface management system that interprets the screen position addressed by the user as a choice of context for the subsequent interaction. By pointing to **o**, the user invokes a state of dialogue comprising a system of definitions for **x**, **y** and **z** in terms of **o**, **a**, **b**, **c** and **r**, and is privileged to change the parameter **o**. By pointing to **x**, the user invokes a state of dialogue comprising a system of definitions for **o**, **a**, **b**, **c** and **r** in terms of **x**, **y** and **z**, and is privileged to change the parameter **x**.

The concept of "a definitive dialogue state as a profile of an intelligent agent" will be further examined in §5 below. It should be noted that the simple protocols used for illustration above are untypical of the context changes that might be programmed in general, and could be enhanced by using more sophisticated input mechanisms (cf §4). The most pertinent point to remark is that - despite the impression that naive enhancement of a pure definitive notation to support monitoring, imposition and perhaps even maintenance of constraints may give - acyclic systems of functional

relationships between variables have a semantics quite different from constraints, and convey information that is not expressed simply by using a constraint. As the above example shows, there are many "intelligent views" of Figure 1 that are consistent with the specified constraints, each corresponding to a different ingredient of the protocol for interactive constraint management. Another important consideration is that certain of these views conflict, in that the associated processes will in general interfere if they should execute concurrently. For instance, the order in which processes are invoked in order to change the parameters **r**, **o** and **a** is not significant - they are non-interfering, but it is not possible to move the centre of the circle and displace a point on the circumference concurrently.

The above discussion focusses on constraints as relationships that a designer may need to impose upon a specific object and its attributes. In a constraint-based programming paradigm, the use of constraints can serve many other purposes. Generic constraints might be used to express algebraic identities, for instance, or to define implicit functions. Notice that these can be modelled in other ways within the definitive programming paradigm, and respectively correspond to introducing axioms and additional operators into the underlying algebra.

§4 The user-interface in the definitive programming model

The abstract problem in designing a user-interface is to represent the current context for interaction to the user in such a way that both the status of the entire system and the options that are available to the user to change the state of the system are as easy to determine as possible. The work that has been done on pure definitive notations has focussed on developing such a representation for the conceptual aspects of the user-computer interaction, but there is no reason why the application of definitive principles should be restricted to such concerns. The purpose of this section is to indicate how the use of the extended definitive programming paradigm described above can in principle deal with all the concerns of the user-interface. Naturally, there are many technical issues to be addressed before this objective can be attained, but some of the key ideas will be outlined.

Following [12], an exchange of information between the user and the computer is conceived as having four ingredients: **conceptual** and **functional** elements concerned with its meaning, and **sequencing** and **hardware binding** elements concerned with its form. The conceptual elements of a geometric modelling system might correspond to the algebra underlying the definitive notation described in §1.2 - or perhaps more realistically to user-defined data types and operators over this algebra, and the functional elements to particular families of entities that are instantiated during an interaction implemented using the abstract machine model described in §2.2. To capture the entire functional design in this machine model, it is necessary to apply definitive principles to the specification of the display, so that the entities may include definitions of variables that are to be interpreted as representing the current state of the screen display. For sequencing purposes, the abstract machine model provides a state machine that has a richer structure than an augmented transition network, in that each state is represented by a family of definitions, and the state transitions can be very subtle and complex. The consistent use of a definitive programming paradigm to deal with interaction at the higher levels of abstraction has the unfortunate effect of highlighting the discrepancy between the software and hardware models; for this reason, there is little to say at present about hardware binding, and it can only be conjectured that the principles used in [14] - for example - may prove relevant.

The design of a definitive notation for screen layout is currently under development (an appropriate notation dealing with general textual displays has already been described). Such a notation will serve to address the issues that are conventionally handled using visual element editors, and provide the basis for the specification of entities that perform the role of the subroutines in the Interaction Technique Library [12]. The principal components in the underlying algebra include data types to represent character strings and graphics, boxes within which text strings are to be laid out or graphics displayed, windows comprising lists of boxes, and displays comprising families of windows. In effect, the current state of a screen display is abstractly described by a system of variable definitions, and it becomes possible to deal with issues of presentation and window

management (as is appropriate for instance when implementing the DoNaLD user-interface) without resorting to the inelegantly mixed paradigm of EDEN (cf §2.2). Perhaps the most exciting implication of using definitions to describe both the objects of the application domain and the screen display is that it readily becomes possible to establish relationships between the form of a display and its content. As a trivial example, the dimensions of a box can be defined to match its contents, in such a way - if necessary - as to reformat the display when exceptional output is to be displayed. Similar principles can be applied to provide dynamic functional feedback to the user (cf [12]).

Amongst the applications for the above method of display are generic methods for dealing with constraints that are to be monitored in the sense described in §3. It becomes directly possible to link the contents of particular "monitor" windows to boolean variables indicating the current status of particular constraints through an appropriate definition. The entities that would be required for this purpose would be easy to generate automatically in reponse to a declaration that a particular condition was to be monitored.

In principle, the methods for displaying information described above can also be adapted to cope with graphical input. As a philosophical point, it seems likely that some emphasis on textual rather than graphical input must be retained however (cf [8], p369), since variable definitions will presumably be hard to introduce using analogue techniques.

As the analysis of constraint-processing in §3 shows, even the conceptual aspects of the user-computer interaction are inadequately represented by a naive use of definitive principles. The user does not in general act within the framework of a single system of functional relationships between variables, but may "select the system of functional relationships" that is appropriate for an intended action. Strictly speaking, such a selection can be articulated directly by the user, who simply needs to preface an action (such as "redefining the parameter **o**" in Figure 1) by an appropriate series of variable redefinitions (such as "redefining the variables **x**, **y** and **z** in terms of **o**, **a**, **b** and **c**"). Though this illustrates how a pure definitive notation supplies a form of intermediate code for interaction, it is clearly inconvenient for the user. Within the extended definitive programming model, subject to introducing auxiliary variables to establish a protocol, the required transition between dialogue states can be handled automatically. To achieve this, the concept of the current state of the dialogue must be enhanced to include information about the user's present intentions, as represented by the definition of a control parameter (such as "the currently selected element" in Figure 1). As an aside, it is of interest to note the resemblance between the user's invocation of an appropriate context for an action, and the generally undesirable use of modes (cf [12] p8). It remains to be seen to what extent the transparent nature of the interface presented by a system of variable definitions alleviates this problem.

In principle, it is clear that dialogue sequence specification can be represented within the extended definitive programming paradigm. A system of variable definitions is a more powerful way to represent state information than an augmented transition network, and - as the above discussion indicates - there is considerable scope for introducing complex transitions between such systems. At this stage, there is much research still to be done into the most appropriate programming techniques for handling control in the abstract machine model of §2.2, but there are several promising indications. It should be easy to accommodate global commands for instance, to support undo actions, and to provide a context sensitivity that encompasses many orthogonal concerns.

Naturally, the use of definitive principles to represent user profiles is advocated as the appropriate approach to developing adaptive interfaces. To achieve this, it will be necessary to establish an "algebraic framework" within which user responses can be monitored and dynamically represented by an evolving system of variable definitions. Such definitions might take the form for instance of numerical data on the proportion of errors made, or commands used, together with boolean variables indicating skills successfully acquired. Notice in particular that the use of definitions avoids issues concerned with the order in which goals are accomplished. The appropriate dialogue context for computer responses can then be determined according to the current status of the user profile. Such a mechanism naturally complements the user's selection of a particular dialogue

context as described above, and requires no greater technical virtuosity.

§5 Definitive programming for intelligent CAD

It can be argued that the representation of state by systems of variable definitions captures an important constituent of human intelligence. When contrasted with procedural or functional models, definitive models appear to abstract relationships that match human conceptual processes more faithfully. Mechanical systems provide strong evidence for this point of view. As a simple example, it is much more appropriate to model an object such as a door as a parametrised system of variables representing the hinge, the handle and the catch than to conceive it as determined by a family of procedural variables that are independently updated in such a way the certain constraints are necessarily satisfied, or as a mathematical abstraction such as a functional programmer might employ that incorporates no concept of current state.

The naivety of proposing such pure definitive models for general systems becomes clear when more complex interactions between objects are involved. As illustrated in §3, systems subject to complex constraints can only be accurately modelled by implicitly introducing many different functional relationships between variables, not all of which can be consistent (ie acyclic). To accommodate this within a definitive framework, it becomes necessary to bind systems to the agents who can observe and act upon them consistently. Rather than modelling "the state of the system", it is then appropriate to model the views of a system as perceived by particular agents. From this perspective, a system of definitions is a way of modelling the view of one agent, who has (conditional) control over certain explicitly defined parameters, and can predict the effect of changing these parameters upon the dependent variables that are implicitly defined. It is in this sense that the definitive programming model of a door is a natural one; it simply expresses the way in which the door can be expected to interact with an agent.

It is of interest to examine more closely the idea - perhaps fanciful - that the concept of "intelligent views" can be an ingredient in a formal framework for studying "intelligence". Guided by analogy with the notion of "intelligent view", it will be appropriate to suppose that our perception of a system can be modelled by a family of variables, and that it is possible to observe the behaviour of the system through changes in the values of these variables with time. In such a context, intelligence about the system might be construed as "knowledge of universal relationships between system variables". There then appears to be a significant distinction between the kind of intelligence that is involved in the perception of constraints, and that involved in "intelligent views" as described above. In effect, there is the intelligence of an observer of the system, who perceives certain invariant constraints between variables (eg as in a system of collinear points A, B and C that autonomously changes so that at all times the distance between A and B exceeds that between B and C), and that of an agent, whose intelligence consists in knowing the effect of actions upon the system (eg if I increase the distance AB by d units the distance BC will increase by the same amount).

The key to unifying the intelligence of the observer and that of the agent appears to be to consider more restricted system models, and to confine the system changes that occur to those that can be brought about by participating agents. It is then no longer possible to speak abstractly of "changes of values of variables in the system with time", but only of actions performed upon the system by legitimate agents. The intelligence of the observer takes the form of consequential knowledge of global constraints upon the system behaviour.

The extended definitive programming model described in this paper seems to offer a most appropriate way to give formal expression to these ideas. A system will be represented by a set of variables that are perceived by various agents to satisfy functional relationships expressed in the form of an acyclic system of variable definitions. The system will evolve from state to state through changes made to parameters - subject to appropriate pre-conditions being met - by the participlating agents. The possibility of concurrent action of two or more agents is not discounted, but there will in general be a need to constrain the behaviour of agents to ensure non-interference, ruling out

concurrent action by two agents that would lead to inconsistent changes in the value of a variable. The global constraints on the system are the invariant relationships between variables - those that cannot be violated by the action of any agents. For convenience, such a system model will be termed a **definitive model**; it may be regarded as describing one possible intelligent interpretation of the system. The nature of the abstraction that is being made in definitive programming (cf §1) also becomes clearer in this perspective; it is not strictly necessary that the maintenance of functional relationships within an agent's intelligent view should be instantaneously updated, but only that these relationships can be guaranteed to persist as postconditions of any legitimate action on the part of the agent, and cannot be subject to interference through the concurrent action of another agent.

There are strong analogies to be made here with the scientific method. "Opening the door" is an experiment that confirms the thesis that the relationship between the essential parameters of the door is correctly perceived. The "intelligent view" of the agent formally represents an assumption that is properly seen as an article of faith, confirmed by experience but unproveable; it may be the case that one day the system will confound the agent's expectation, as when the door comes off its hinges. The quantum theoretic principle that every observer is necessarily an agent [7] also comes to mind.

To illustrate and elaborate the above ideas, it will be helpful to examine a simple example. To this end, suppose that the variables **x**, **y**, **z**, **o**, **r**, **a**, **b** and **c** in Figure 1 supply the basis for a system model. In §2, one possible definitive model consistent with Figure 1 was described. Within that model, the "intelligent views" of the agents were the systems of functional relationships between variables corresponding to the different expectations of the user on selecting different parameters for change. In that model it was coincidentally the case that the variables **x**, **y**, **z**, **o**, **r**, **a**, **b** and **c** were always subject to the constraining relationship graphically depicted in Figure 1.

Of course, there are all kinds of alternative definitive models that might be used to interpret Figure 1. It might be the case that the points and scalar parameters in Figure 1 happened to have the values depicted: in the view of any agent, all the variables would be explicitly defined, and there would be no functional relationships between the variables. This is very much the kind of mental model that underlies conventional procedural programming, and that the discipline of introducing invariants - or object-oriented programming methods - adapts for use in other contexts where there are constraints between variable values to be met.

There are also a number of interesting mechanical interpretations of Figure 1 that might be considered. It may be that **ox**, **oy** and **oz** are the spokes of a wheel; that they indicate the positions of the hour, minute and second hands of a clock; that **ox** and **oy** are diametrically opposite points on a wheel, and **oz** is the point of contact with the ground. To each of these there corresponds a definitive model that describes the effect of agents lifting or rotating the wheel, moving or resetting the clock.

Characteristic of the original model of Figure 1 (as discussed in detail in §2) is the richness of the functional relationships invoked by agents; something that is in general difficult to realise in a mechanical model. This probably accounts for the particular difficulties of programming a user-interface for interactive graphics and CAD software. There can nonetheless be a role in a mechanical system for "switching between different intelligent views" through a simple action similar to the selection of a point on the screen. Suppose for instance, that Figure 1 represents a wheel in which the spokes are bolted together but detached from the rim. Unbolting the spokes has the effect of radically changing the context for an agent who is privileged to be able to move a spoke. What is more, if there were to be an agent for each spoke, the effect would be to eliminate the interference between these agents. It is in the representation of such "intelligent" interactions that definitive principles offer most promise.

The approach to knowledge representation proposed above may be contrasted with the predominantly inference-based techniques commonly used in AI (cf [17]). This relationship has yet to be explored, but promises to be useful both for evaluating and extending the above ideas.

Concluding remarks

The work described in this paper has been motivated by three distinct concerns:

1) a pragmatic concern for developing and implementing sophisticated definitive notations as the basis for CAD software. This aspect of the work is for the present primarily concerned with finding better ways to describe the programming methods that have been used with considerable success for a prototype implementation of the DoNaLD notation (cf [6]).

2) a mathematical interest in developing an abstract programming paradigm that incorporates the principles used in pure definitive notations, but can be applied to more general problems. The abstract machine model described in §2.2 is what appears to be the most appropriate generalisation, in the light of experience with related work on EDEN [6] and LSD [5].

3) a semantic concern with fundamental principles of interaction, and the significance of definitive principles as a way of representing knowledge about systems in a psychologically convincing way. Where pure definitive notations are concerned, some of the most important issues have been already addressed (cf [2]), but the perspective presented in this paper suggests further potential within a less restricted programming model.

Respectively associated with these concerns, there are three broad objectives: implementing large software systems that exploit definitive principles, describing a satisfactory semantics for definitive programming, and developing new applications.

The implications of these three strands of research are at present only partially understood. The machine model described in §2.2 has as yet been little developed, for instance, and considerably more research is required before the methods currently being used for implementation can be recast in an abstract form. Work is in progress on the development of the definitive notation for geometric modelling (cf §1.2 and the Appendix), and on a definitive notation for describing presentation issues in the display interface (cf §4). The problems posed by concurrency, and the programming techniques needed to deal with synchronisation issues are also a particular focus of current concern. What is now clear is that there is no satisfactory way to compromise in the use of definitive principles; whatever the shortcomings of the extended definitive programming framework outlined in this paper, there is a strong motivation to fulfil the objectives set out above. As a footnote, it may be worth observing that the computational model that is perhaps most appropriate at the primitive hardware level, where the output of a gate is directly determined as a function of its inputs, closely resembles a definitive program!

The extended definitive programming framework described in this paper has something in common with both functional and object-oriented programming methods. The specification and augmentation of the data types and operators of the underlying algebra offers much scope for functional programming techniques such as are to be found for instance in [24]. There are clear connections between the methods for constraint processing and user-interface management discussed in §'s 3 and 4 and the application of object-oriented principles as described in [8]. Characteristic of the definitive programming approach is the explicit identification of relationships between variables as viewed by an agent; a significant form of abstraction that cannot be made explicit in object-oriented models but is often implicit in an object-oriented implementation. The use of trigger constraints by which "responses are keyed to particular message selectors to be received by specific *instances* " (identified as a powerful feature of the BALSA animation system in [8] p369) is one example of an implementation technique that provides convenient support for such relationships. It is of particular interest to see that variants of definitive programming have been implemented in many different programming paradigms: cf the data-base language ISBL [23] (in a PL1 environment), The Analytic Spreadsheet [11] (in an object-oriented environment), and the geometric design tool RELATOR [19] (in a Prolog environment). This may be construed as further evidence that the use of variable definitions to represent relationships is of significant interest, but outside the immediate scope of traditional paradigms.

Acknowledgements

The research described in this paper owes much to the support of Samia Meziani, Mike Slade and Edward Yung. We are also indebted to the University of Warwick Research and Innovations Fund, and to the Computer Science Department, for financial support.

References

1: W M Beynon, *Definitive notations for interaction*, Proc hci'85, CUP 1985, 23-34
2: W M Beynon, *Definitive principles for interactive graphics*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 1083-1097
3: W M Beynon, *ARCA: a notation for displaying an manipulating combinatorial diagrams*, Univ of Warwick RR#78, 1986
4: W M Beynon, D Angier, T Bissell, S Hunt, *DoNaLD: a line drawing system based on definitive principles*, Univ of Warwick RR#86, 1986
5: W M Beynon, *The LSD notation for communicating systems*, Univ of Warwick RR#87, 1986
6: W M Beynon, E Yung, *Implementing a definitive notation for interactive graphics*, Proc CG'88
7: Niels Bohr, *Atomic Physics and Human Knowledge*, Science Editions Inc, NY 1961
8: A Borning and R Duisberg, *Constraint-Based Tools for Building User Interfaces*, ACM Transactions on Graphics, Vol 5, No 4, October 1986, 345-374
9: U Cugini, *The role of different levels of modelling in CAD systems*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 881-898
10: R T Farouki and J K Hinds, *A Hierarchy of Geometric Forms*, IEEE Computer Graphics & Applications, May 1985, 51-78
11: J J Florentin, *The Analytic Spreadsheet of Objects*, OOPS 11: Beyond the Beginning, 1987
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: R Forrest, *Geometric Computing Environments: Some Tentative Thoughts*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 185-198
14: P ten Hagen, R van Liere, *A model for graphical interaction*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 517-542
15: K E Iverson, *A Programming Language*, Wiley 1962
16: M L Kersten, F H Schippers, *Towards an object-centered database language*, Report CS-R8630, CWI Amsterdam 1986
17: J Lansdown, Graphics, *Design and Artificial Intelligence*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 1153-1174
18: G Nelson, *Juno, a constraint-based graphics system*, SIGGRAPH '85, 235-243
19: M Y Rafiq & I A MacLeod, *Logic Programming for Technical Design*, Workshop on AI in Civil Engineering, AI Applications Institute, Edinburgh University, November 1987.
20: R F Riesenfeld et al, *Computer aided design*, UUCS-84-003, CS Dept, Univ of Utah 1984
21: T Smithers, *AI-Based Design v Geometry-Based Design*, Workshop on AI in Civil Engineering, AI Applications Institute, Edinburgh University, November 1987.
22: T Takala, C D Woodward, *Industrial design based on geometric intentions*, NATO ASI Series F; Computer and Systems Sciences, Vol 40, 953-964
23: S J P Todd, *The Peterlee Relational Test Vehicle - a system overview*, IBM Syst J 15, 285-308
24: D Turner, *An Overview of Miranda*, Bull EATCS, Oct 1987, 103-114
25: C J Van Wyk, *IDEAL User's Manual*, Computing Science TR #103, Bell Labs, 1981
26: K Weiler, *Edge-based data structures for solid modelling in curved surface environments*, IEEE Computer Graphics and Applications, 1986, 21-40

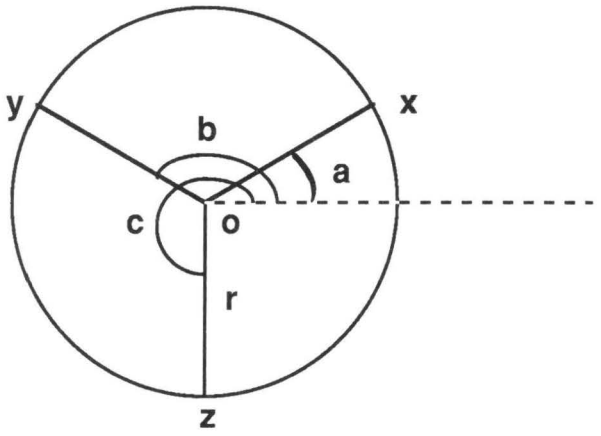


Figure 1: A system of constraints

MOLE: A Reasonable Logic for Design?

C. Tweed

A. Bijl

MOLE: A Reasonable Logic for Design?

Christopher Tweed
Aart Bijl

EdCAAD
University of Edinburgh
20 Chambers Street
EDINBURGH
EH1 1JZ
{chris, aart}@caad.ed.ac.uk

February 1988

This paper charts the development of a logic modelling system, MOLE (Modelling Objects with Logic Expressions). The system is intended to provide designers with a modelling environment in which to represent design objects and tasks. In this paper we trace the evolution of the underlying theory through experience of implementing and using prototypes of the system. In doing so our aim is to document the more important system design decisions.

INTRODUCTION

Developers of CAD systems can now choose from a wide range of techniques. Yet theories to develop systems from these techniques are lacking. As a result decisions tend to be based on arbitrary criteria, so opening the door to inconsistencies such that the finished system may fail to reflect the developers' original intentions. Explicit theories are needed as a framework for decision-making: to evaluate alternative techniques and assess their impact on objectives, and to guide the implementation of prototypes through to a working system. Implementing prototypes, however, often reveals weaknesses in the theory, which may prompt revisions and extensions. Theories must remain flexible to take account of practical considerations. Conflicts between theory and practice need to be resolved within a broader strategy. This paper describes the strategy for a logic modelling system, MOLE (Modelling Objects with Logic Expressions), the role of theory, and the interactions between theory, implementation, and use during its development.

MOLE was conceived as a general description system to enable designers to describe objects and tasks using words and drawings. The brief for MOLE emerged from a study of past CAD systems[1], which concluded that the definitions of objects and methods encapsulated in these systems often contradict individual designer's varied and changing intuitions. The problem is mainly the result of a component-based approach to design which fails to recognise or accommodate the evolution of design knowledge and practice. MOLE is intended to counter this problem of *prescriptiveness* by enabling designers to describe objects and methods as they perceive them. Hence the system must accept descriptions without conditioning their content.

The desire to minimise prescriptiveness, more than anything else, has determined our choices in developing MOLE.

The aim of this paper is to explain important decisions in the development of MOLE — with an emphasis on the problems presented by implementation, and on how these are resolved at the theoretical level.

The paper begins by setting out a broad strategy for MOLE which clarifies the starting point for development and presents the main objectives. This is followed by a brief report of our approach to developing the system, and provides the background to discussions of three key areas of MOLE's development: representation scheme, expressions, and design tasks. In each of these, the interactions between strategy, theory, and prototypes are discussed.

A DESCRIPTIVE STRATEGY FOR CAD

MOLE has been developed as a vehicle for exploring descriptive methodologies in CAD. In a companion paper[2], Bijl describes a broad strategy for CAD which advocates descriptive rather than prescriptive methodologies. The main points are summarised here.

The argument for descriptive methodologies in CAD rests on the belief that design is a highly idiosyncratic human activity, and that consequently CAD systems should not impose definitions of domain objects and tasks upon designers. By adopting a descriptive approach, we enable designers to represent their own perceptions of objects and tasks. A non-prescriptive system should:

a) accommodate users' descriptions without conditioning their content;

b) enable descriptions to be presented to the system either as text or as drawings, or as both; and

c) provide general functionalities independent of particular domains.

To realise these goals the system must reject any notion of semantics which map onto things in the users' world. In short, computers process symbols regardless of users' interpretations of those symbols. This has led us to view MOLE as an extended word-processor — the extensions enabling users' to structure their words to form descriptions.

METHOD OF DEVELOPMENT

Our method of developing MOLE has been to formulate a theory of descriptions based on an informal characterisation of design. This theory has since been expressed formally by Krishnamurti[3], and is to appear in revised form in a paper by Krishnamurti and Tweed[4]. From here we have advanced theory and implementation in tandem, so allowing them to interact, each informing the other. Implementation and use of the system expose omissions and flaws in the theory which have to be considered in the context of our strategy before they can be adopted in the theory. The process can loosely be described as a cycle, though in practice it is more complex. As the scope of the system has expanded so interdependence of theoretical concepts has increased: seemingly trivial changes reverberate throughout the entire theory. Maintaining a prototype consistent with the theory, therefore, demands much from the programming environment.

The choice of Prolog as a programming language has made the task of implementation much easier than it would have been with conventional procedural languages. The C-Prolog interpreter implemented at EdCAAD, together with the UNIX* operating system provided the programming environment for this work. As an approximation to first-order logic, Prolog encourages formal specification of theory, since theory expressed in logic can so easily be implemented as a program. Programs can thus be treated as proofs of the theory. Prolog's declarative approach also frees system developers from having to translate theoretical goals into machine instructions, allowing them to concentrate on the logic of the system instead. Moreover, the use of an interpreter permits rapid modification of prototypes to reflect changes to theory. As MOLE has been used to provide undergraduate students with a modelling environment we have had to develop prototypes that are capable of running modest applications at reasonable speeds. To improve performance I/O routines were written in the more conventional programming language C, and are invoked from within an extended version of the C-Prolog interpreter[5].

The remainder of this paper focusses on three key areas of MOLE's development:

- representation scheme: how descriptions are represented by a formal system;
- expressions: the form of expressions and their evaluation by the system;
- tasks: functionalities the system provides to enable designers to describe design tasks.

REPRESENTATION SCHEME

The choice of representation scheme is crucial to achieving the objectives for MOLE. The scheme must be capable of accommodating users' descriptions of objects, concepts, indeed anything a user may wish to describe, without conditioning the content of descriptions. Our theory of descriptions is based on criteria established by Bijl[6]:

- wholeness: the system should not see a whole as different to a part, nor a separate status for a root part, and any part may become a component of any other part;
- discreteness: there should be no notion of boundaries to parts or to assemblies of parts such that other parts are excluded, and changes to a description of a part should propagate to any other part that it describes;
- prior typing: the system should not depend on any prior definitions of types that have to correspond to types perceived in the user's world;
- correctness: the system should not imply any notion of an independent or absolute arbiter of correctness, it should reflect only what it has been told by users, and what it can infer from that.

The first task, therefore, is to devise a representation scheme that will satisfy these requirements.

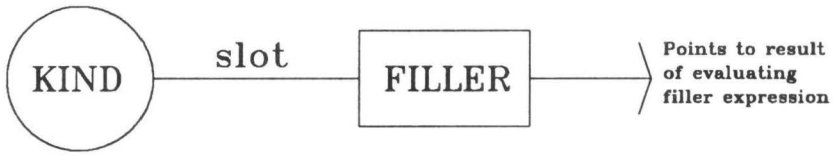
---

* UNIX is a Trademark of Bell Laboratories.

Kinds, Slots, and Fillers

We begin by assuming that we some "thing" we wish to describe, which we represent in MOLE as a *kind,* so named after "any kind of thing". A kind is therefore a type, but one which does not restrict what we can say about it. Kinds are described by their parts which are attached to them via *slots.* Slots may be filled with references to other descriptions. Kinds, slots, and fillers are the basis of MOLE's representation scheme. They support a *part-of* relationship between the kind and the expressions that fill its slots. Most importantly, the names of kinds and slots are entirely the choice of the system's users (with the usual provisos that they must consist of alphanumeric characters). The system treats these names purely as lexical tokens, such that the user is free to choose any names for these entities without the risk of clashing with system-defined keywords. (In earlier prototypes this was not possible because the system did use keywords to invoke various operations.) The *kind-slot-filler* relationship is depicted graphically in figure 1.

FIGURE 1: KIND, SLOT, AND FILLER RELATIONSHIP



Though the figure illustrates a single part-of relation, any number of (including zero) slots (filled or unfilled) may be attached to a kind, i.e. descriptions are infinitely extendible within the limits imposed by machine hardware. Kind-slot-filler relationships are represented by expressions of the form

$$K : [\ s_1 = F_1,\ s_2 = F_2,\ \ldots,\ s_n = F_n\ ]$$

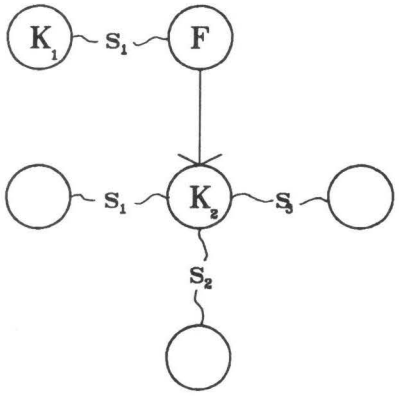
where K is a kind, s_1 to s_n are slots, each with names unique to this description, and F_1 to F_n are the fillers of these slots. The expression borrows Prolog's list notation to group parts of a description. Descriptions of parts are introduced by the ':' operator; the '=' indicates the connection between slots and their fillers.

Kinds as Fillers

A kind filling a slot in the description of another kind establishes a part-of relation between the two kinds. Thus, in figure 2, the description of K_2 is a part of the description of K_1 through K_1's s_2 slot.

Using kinds as fillers has implications we should consider. A kind may fill more than one slot such that changes to the description of that kind appear in every description that references it. But we can prevent kinds used as fillers from propagating changes to other descriptions, and thus support contextual variations, by allowing kinds to inherit all or part of their descriptions from other kinds.

FIGURE 2: KINDS AS FILLERS



Inheritance

Kinds, slots, and fillers support general *part-of* relations which include the description of one thing as part of the description of another. A thing is either part of a description or not. Inheritance offers an alternative to this "all or nothing" approach by allowing partial descriptions to be included within the descriptions of other things. We explored two types of inheritance as a means of extending MOLE's representation scheme: these are supported by *instances* and *variants* respectively.

Instances: are kinds that inherit all or part of their descriptions from one other kind. An instance is named after its parent kind, and may only appear as a filler within an existing description. The properties of an instance are:

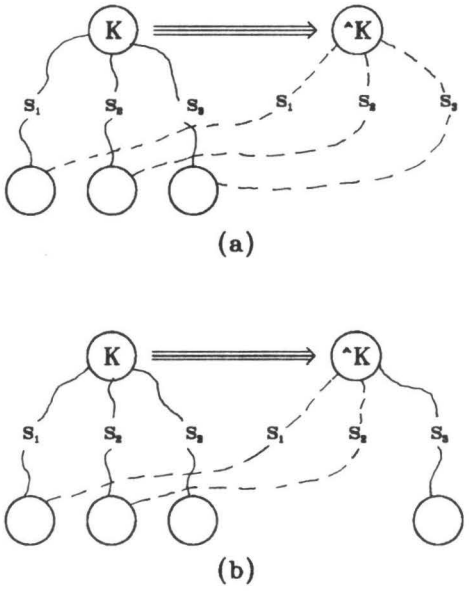
- it inherits slots from its (single) parent kind
- it inherits any slot belonging to the parent that is not *masked* by a slot with the same name directly attached to the instance
- it can have parts altered or removed without affecting the description of its parent kind
- it sees changes made to parts inherited from its parent — provided the slots are not masked

Figure 3 shows an inheritance relation between a kind *K* and one of its instances, ^*K*. Note that though there may be many instances of *K*, they all have the same identifier (^*K*), and can only be distinguished by their location as parts of other descriptions. In (a) all slots of *K* are inherited by ^*K*, whereas in (b) ^*K* has slots which mask those that would normally be inherited from *K*. In the figure, dotted lines indicate inherited slots, and solid line indicate slots local to the instance masking inherited slots with the same name from the parent.

Variants: are kinds that inherit from one or more other kinds. Conflicts between slots with the same name, inherited from different parent kinds, are prevented by tagging inherited slots with the names of their parents. Thus slots in descriptions of variants are displayed as

$$s/P = F$$

FIGURE 3: INHERITANCE RELATIONS



where *P* is the name of the kind *s* is inherited from. *F* is the filler, as before.

Variants have not survived to the current prototype. The complications of multiple inheritance outweighed its utility, but similar effects can be produced by using separate instances as parts of a description to inherit from different kinds.

Numbers, Character Strings, and Lists

To increase the generality of the representation scheme numbers and character strings were introduced to supplement kinds (and instances), and slots. Lists were added as a convenient method of grouping expressions. Further types — for example, ranges — were considered but rejected as being too specialised, as they can be represented using the types already defined.

Table 1 summarises the basic types and shows examples of MOLE naming conventions: kinds in upper case, slots in lower case, instances preceded by the '^' operator.

Table 1: Summary of basic types.

Type	Description	Example
kind	a name for 'any kind of thing'	*HOUSE*
instance	a kind which inherits parts from another (parent) kind	*^HOUSE*
slot	a name attached to a kind	*plan*
number	an integer or floating-point number	1.414
character string	a string of characters	"Marley and Co. Ltd."
list	a list of any of the above	[*HOUSE, LIBRARY, SHOP, OFFICE*]

Having introduced the basic types we can consider networks constructed from part-of and inheritance relations that are used to represent larger descriptions.

Paths and Context

Parts of a network are identified by paths. A path begins with a kind and continues with a sequence of one or more slots connected by the ':' operator, and ultimately leads to a filler, passing through connected descriptions. Since names of slots are unique to each description, a valid path will point to one filler only. For example, the path

HOUSE : *front_door*

names the *front_door* part of *HOUSE*. A path may traverse any number of kinds, as in

HOUSE : *living_room* : *carpet* : *supplier*

A path used as a filler is called an *indirection*.

Paths establish context by passing through descriptions in a specified order. Context consists of the names of kinds identified implicitly by the path, and the immediate slots and fillers in the descriptions of those kinds. In figure 4, *HOUSE* : *living_room* : *carpet* : *supplier* defines a context consisting of *HOUSE*, its slots and their fillers, *LIVING*, its slots and fillers, and *CARPET* and its slots and fillers. Slot-names used as fillers of other slots operate as paths, pointing the nearest slot in the context. Thus within *CARPET* a reference to (say) *width* will implicitly refer to the *width* slot in the description of *LIVING*. Since context includes the slots of the description in which the expression is being evaluated filling a slot with its own name will cause a loop, for example

CARPET:
width = *width*

We have introduced an operator to exclude the current description from the context search. To prevent the filler *width* from matching itself we say

CARPET:
width = *<: width*

A reference to a kind is treated in the same way as a reference to a slot, but will match any instance of a kind found in the context. In general, a reference matches the nearest kind, instance, or slot with the same name, i.e. the kind, instance, or slot most recently added to the context.

Context allows the initial kind name in a path to be omitted since the system can match slot names in context, though kind names are needed to identify a description without relying on context, or to explicitly override the order of a context search. Indirections, therefore, may be expressed in terms of slots, as in figure 5. Note that the indirection *worktop* : *width* always points to the *width* slot of whatever kind fills the *worktop* slot. Indirections thus maintain dependencies across descriptions.

Implementing Descriptions

Initially the representation scheme was implemented as a loose, i.e. unstructured, collection of Prolog facts

FIGURE 4: CONTEXT

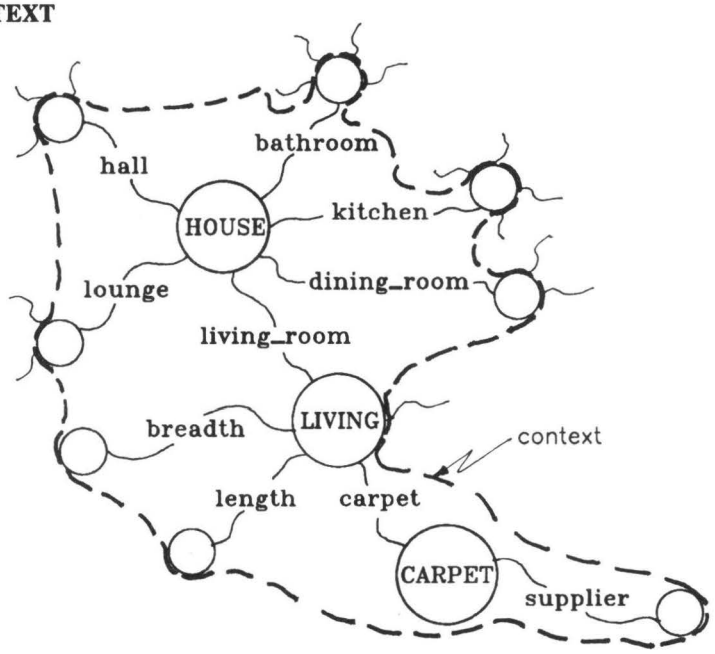
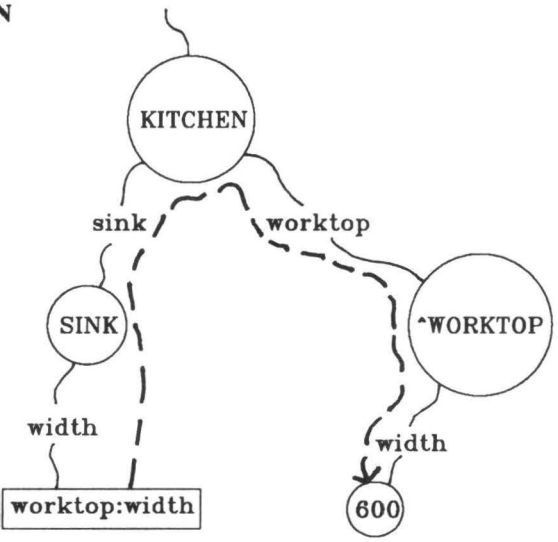


FIGURE 5: INDIRECTION



kind(K).
slot(K, s_1, F_1).
slot(K, s_2, F_2).
slot(K, s_3, F_3).

Views of descriptions were constructed only when requested by the user. This has the advantage of allowing immediate access to parts of descriptions without having to decompose structures to change one part.

The problem with this strategy was that, because of the ordering of the Prolog database, re-asserted slots would change their positions within descriptions, such that when a user viewed a modified description the parts were often displayed in a different order. The order of parts was

not important in early versions of the system, but when the scope of the system was extended to include descriptions of users' tasks a consistent approach to ordering description became important.

The concept of order has yet to be expressed formally in the theory, and for now is assumed to be dealt with by the order that facts are presented to the system in expressions. The problem may be overcome at the user interface by providing a description editor; this is a long term aim. But in the implementation the representation of descriptions has been changed to use more structured facts, which prevent unnecessary changes to the order in which parts are presented to the user.

description(K_1, [(s_1, F_1), (s_2, $F_{2)}$, ..., (s_n, F_n)]).
description(K_2, [(s_1, F_1), (s_2, $F_{2)}$, ..., (s_n, F_n)]).
...
description(K_n, [(s_1, F_1), (s_2, $F_{2)}$, ..., (s_n, F_n)]).

As far as the user is concerned, however, access to individual parts is not affected by this new scheme, but changes and additions to descriptions demand more of the system.

EXPRESSIONS

In general the task of developing a CAD system can be thought of as:

1) specifying a language to enable users of the system to access its functionality; and

2) defining the functionality of the system, i.e. the operations the system will perform.

This section describes the general form of expressions, and a treatment of names which allows several descriptions to be referred by a single name. This is followed by a discussion of the evaluation simple MOLE expressions.

Form of Expressions

It is often assumed that the most natural form of expression will provide the most effective means of communicating with a machine. Considerable effort is directed towards equipping machines with natural language understanding abilities. The problem with this approach is that users' natural language expressions are ultimately mapped onto logical representations via a restricted set of semantic primitives defined in a lexicon. The lexicon hides the system's logic from the user, and thus makes it difficult for users to identify boundaries to the system's "understanding". This can lead to misunderstandings analogous to people talking at cross-purposes.

In contrast to this approach, our goal is that users' expressions should map directly onto logical operations defined by the system. The form of expressions should unequivocally reflect the logic of the system. The view of expressions which has emerged is that they consist of user-defined words, which have meaning to the user, and system-defined operators that structure these words according to rules defined by the logic of the system. System-defined operators are always represented as non-alphanumeric symbols to clearly demarcate the responsibilities for interpretation in expressions. The visual appearance of operators, therefore, can be likened

to that of punctuation in written text.

Collecting and Selecting Parts

So far we have dealt with names which uniquely identify descriptions, but it is often useful to be able to collect things into groups. There are two ways to collect things in MOLE: first, things can be named explicitly in a list; and second, a list can be generated by using an expression that matches more than one name. This is done using the wild card character '*' to indicate that the word given may be completed by any sequence of characters. Hence "*" matches any word, whereas "wall*" matches any word which begins with "wall" followed by zero or more characters. More than one wild card may appear in the same word, fulfilling the same role in each occurrence: "*right*" matches any word with "right" in it.

The following two expressions are equivalent

$$K : * \quad \text{and} \quad K : [\, s_1, s_2, \ldots, s_n \,]$$

Wild cards offer a powerful way of collecting things based on lexical matching of names. Their introduction to MOLE came about as a result of working with prototypes, they were not accounted for in the formal specification of the theory, and tend to be regarded as a convenience rather than as a formal device. Their use, however, is consistent with the extended word-processor analogy. Where other systems might collect things according to type, wild card matching is based solely on the lexical properties of user-defined words.

Conditions

We may want to be more selective in collecting things. By applying conditions to expressions we can select on the basis of other criteria, such as the relationships of parts, numerical comparison, etc. Conditions have a filtering effect on lists, rejecting members of a collection that fail to satisfy the condition/s. The syntax is

$$< path : > [\, expr_1, expr_2, \ldots, expr_n \,] \{ conds \}$$

Note that the leading path or paths, enclosed in <>'s, are optional.

Within the braces, one or more conditions can be supplied, separated by commas, or semicolons to indicate conjunction and disjunction respectively. Thus

$$\{ cond_1 , cond_2 ; cond_3 \}$$

reads as "condition 1 and condition 2 or 3" — conjunction takes precedence over disjunction. Parentheses may be used to obtain "condition 1 and 2, or 3" as in

$$\{ (cond_1 , cond_3) ; cond_3 \}$$

Individual conditions may apply to the fillers of paths in the preceding list, parts of those fillers, or indeed any thing described in MOLE. A leading '=' indicates that the following condition is applied to fillers, whereas a leading ':' indicates that it is applied to parts.

$$K : [\, s_1, s_2, \ldots, s_n \,] \{ \; : \; cond_1, = cond_2 \}$$

applies one condition to the parts of $s_1, s_2, \ldots, s_n$, and one to their fillers. Conditions may be negated using the '~' operator:

: ~*cond* and = ~*cond*

To conclude this section table 2 gives examples of selection expressions. To shorten explanations in the table it is assumed that "rooms" refers to slots that match *room**, likewise "windows" and *window**, etc.

Table 2: Examples of Selection Expressions.

Expression	Selects
HOUSE : *room**{ : *window** }	rooms with windows
HOUSE : *room**{ : *window**, : *orientation* = *SOUTH** }	rooms with windows **and** a southerly orientation
HOUSE : *room**{ : *window** ; : *rooflight** }	rooms with windows **or** roof lights
HOUSE : *room**{ = *main_space* }	rooms with a filler the same as the filler of *main_space*
HOUSE : *room**{ : *area* > 5 }	rooms with an area greater than 5
HOUSE : *room**{ = *LIVING* }	only those rooms filled by *LIVING*
HOUSE : *room**{ = *LIVING* ; = *DINING* }	only those rooms filled by *LIVING* **or** *DINING*

Operating on Descriptions

In this section we consider two main types of expressions and the operations they invoke:

- statements — which create and modify descriptions; and
- queries — which examine and generate views of descriptions.

Parts of expressions are subject to a general set of evaluation rules. Evaluation is the general term we use to describe processing of users' expressions. Since all expressions are evaluated we must define the results of evaluation over all types and relations. The rules of evaluation for the basic types are:

- character strings, and numbers are treated as constants;
- a kind on its own is treated as constant, but may point to the nearest kind or instance with the same name, within a given context;
- the result of evaluating a path is the filler at the end of the path;
- a slot on its own is treated as a path which must be resolved by context;
- the result of evaluating a list is a list of results of evaluating each element in the list; and

- all expressions are evaluated until they produce a constant, or, in the case of lists, a list of constants.

Creating and Modifying Descriptions

Descriptions are created and modified by users' input expressions. There is no distinction between expressions that create a new description, and those that extend or modify an existing description — all such statements are treated as assertions. The effect of an assertion depends on what has previously been stated, such that an assertion contradicting previous statements will modify a description accordingly.

The simplest assertion is a kind-name, which creates a kind with that name, if one does not already exist. Slots may be attached (asserted) singly as in

$$K : s = < F >$$

or in lists

$$K : [\ s_1 = < F_1 >, \ldots, s_n = < F_n >\ ]$$

(Note that <>'s are used here to indicate that the fillers are optional in this form of expression — they are not part of MOLE syntax.) If a filler is omitted the slot is asserted but not filled.

Assertions may be nested to specify parts for more than one kind in the same expression

$$K_1 : [\ s_1^1 = K_2 : [\ s_1^2 = < F_1^2 >\ ]\ ]$$

In the above the kinds used as fillers may be omitted

$$K_1 : [\ s_1^1 : [\ s_1^2 = < F_1^2 >\ ]\ ]$$

giving rise to two possible results of evaluation:

a) if a slot is already filled by a reference to a kind then the expression updates (extends or modifies) its description;

b) if a slot is not currently filled by a kind the system will generate a kind with the specified description, and fill the slot with a reference to it. Names of system-generated kinds are of the form *SYS#id* where *id* is a unique identifier for each of these kinds.

The system always responds to assertions with the list of kinds modified by the expression. Hence the following dialogue might occur:

MOLE> *ROOF* : *pitch* = 30.
== *ROOF*

MOLE> *ROOF* : [*covering* : *min_pitch* = 22.5, *purlin* : *depth* = 200]
== [*ROOF*, *ROOF_COVER*, *PURLIN*]

Parts may be removed by negated assertion, prefixing the part with the negation operator, '~'. Hence to remove a slot we say

$K : \tilde{} s$

and to remove a list of slots we say

$K : \tilde{} [s_1, ..., s_n]$

Kinds and instances can only be removed by negating all references to them. Unreferenced descriptions are then deleted by the system.

Creating Instances

Instances are created by prefixing the name of the parent kind with the instance operator, '^', to give a general form of expression:

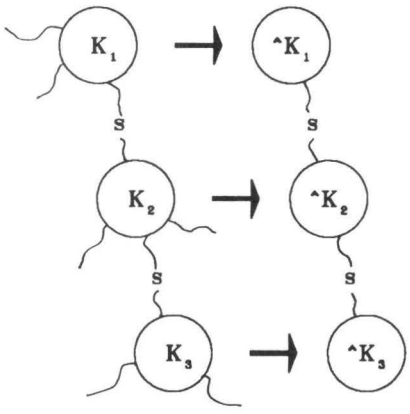
$\hat{} K$

If the parent kind does not exist then it will be created before the instance.

In earlier prototypes instances were created automatically when a part was modified in context, i.e. the target of an assertion was a path. Every filler along the path was instanced to prevent changes from propagating to other descriptions. When changes were required to be global an "all" operator qualified the assertion. More recently we have decided that instancing should be in the control of the user, thus instances are only created now when a user explicitly requests it. The instancing mechanism has also changed to make changes within the instance, not along the path to the instance.

Instancing is recursive such that when a kind is instanced all its parts are instanced, and their parts and so on. This prevents changes to any part of an instance from altering the description of its parent kind. Instances, therefore, provide a means of restricting change to a particular context. The principle is shown in figure 6.

FIGURE 6: CREATING INSTANCES



An instance can exist only within the context of another description; hence instances are always fillers. The creation of instances as fillers depends on the evaluation of assertions, i.e.

when is the instance created?

Controlling Evaluation of Fillers

By default fillers are evaluated, and what is stored, therefore, is the result of that evaluation. But we can prevent the system from evaluating a filler expression by quoting it as in:

$$K : s = '\hat{}F'$$

If $\hat{}F$ is not quoted an instance is created to fill the slot when the assertion is evaluated. Note that the same applies to indirections. If we want the indirection to be evaluated each time the slot that it fills is referenced then we must prevent it from being evaluated when the slot is filled. Hence

$$K : s = s_1 : s_2$$

will fill K's s slot with whatever $s_1 : s_2$ points to, whereas

$$K : s = '(s_1 : s_2)'$$

will store the unevaluated indirection as the filler.

Examining Descriptions

The ability to query, view, and project what the system may know is crucial to our claim that the user should be able to retrieve parts of any expression previously given to the system. We have already seen that certain expressions may be evaluated to produce results, but often the results may either include information we don't want, or provide too little information.

Views are edited versions of the system's responses. However, views do not invoke evaluations of expressions in descriptions; if this is required it must be requested explicitly by the user. Views, therefore, can be used to build expressions which may then be evaluated. Views are called for by qualifying an expression with the query/view operator '?'. Note that in some cases the response will be same for a view as for an evaluation. Hence

$K?$ and K

produce the same system response, K.

Views

In early prototypes views of descriptions were generated as responses to evaluations. For example, the evaluation of a kind was its immediate description — its slots, and their fillers. With the introduction of strict evaluation rules, required to accommodate descriptions of users' tasks, the whole notion of querying and viewing descriptions had to be rethought. The method of viewing descriptions is now a product of the application of wild cards as collective names. The immediate description of a kind can be generated by expanding

$$K : * = *$$

From this we can define expressions to retrieve any part of a description. The response can also be tailored to produce only those parts of the description that the user requires. In database terms, this is known as projection. We use the braces here to denote a viewing context in expressions, where this context is needed to evaluate the complete expression, but is not required as part of the result. Table 3 shows some typical projections of descriptions.

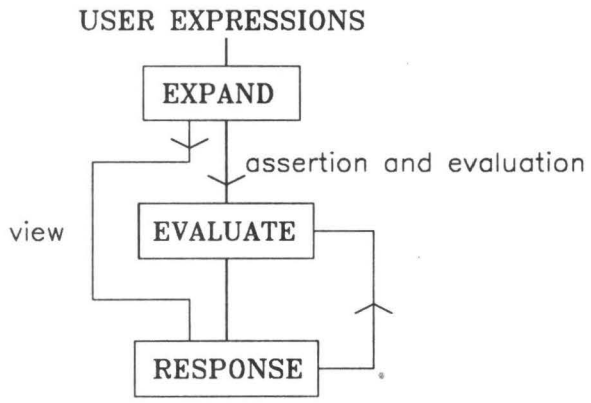
Table 3: Projections of MOLE Expressions.

Expression	View
$K : * = *?$	$K : [s_1 = F_1, s_2 = F_2, \ldots, s_n = F_n]$
$\{K : \}* = *?$	$[s_1 = F_1, s_2 = F_2, \ldots, s_n = F_n]$
$\{K : * =\}*?$	$[F_1, F_2, \ldots, F_n]$
$K : *?$	$K : [s_1, s_2, \ldots, s_n]$
$\{K : \}*?$	$[s_1, s_2, \ldots, s_n]$
$K : * = * : *?$	$K : [s_1 = F_1 : [\ldots], s_2 = F_2 : [\ldots], \ldots, s_n = F_n : [\ldots]]$

Processing User Expressions

Each user expression is processed first to expand all wild-carded names and remove conditions. If the expression is a query or projection evaluation stops at this point and the result is the expansion of the input expression. For other types of expression the expanded expression is handed to an eval predicate implemented in Prolog. The sequence of operations is illustrated in figure 7.

FIGURE 7: PROCESSING USER EXPRESSIONS



By directing the response of the system back through the evaluation, different types of statement can be combined to produce different results. For example to copy the description of K_2 to the description of K_1 we first generate a view of K_2's description — without the kind name — using the projection

$$\{K_2 : \}* = * ?$$

which produces

$$[\ s_1 = F_1,\ s_2 = F_2,\ \ldots,\ s_n = F_n\ ]$$

This can be used as a sub-expression in

$$K_1 : (\{K_2 : \}* = * \ ?)$$

The parentheses force the sub-expression to be evaluated first such that its result replaces the entire parenthesised expression. Thus the expression to be evaluated becomes

$$K_1 : [\ s_1 = F_1,\ s_2 = F_2,\ \ldots,\ s_n = F_n\ ]$$

which is K_2's description projected onto K_1.

TASKS

The evaluations defined so far have been limited to those that manipulate and query users' descriptions. A design system, however, must be more than a passive repository for information; it must also be capable of evaluating tasks useful to designers. However, as with object description, methods provided by the system must not condition what users may later describe to the system as tasks.

Early prototypes of MOLE offered users a functional language with an interface to MOLE's database. This language was conceived independently of MOLE and bore no similarity in concept or in syntax. These prototypes were used to provide undergraduates with a general modelling environment in which to develop simple architectural applications, and are described in[7]. This solution was far from satisfactory, and led to much confusion among users, because they had to learn and operate two different systems. This prototype is documented in[8].

Design tasks are now fully integrated within MOLE and are represented in the same way as objects. Tasks, therefore, can be instanced to respond to their context of use. To incorporate design tasks we have extended the concept of context, presented earlier, to include a single kind to describe the system and to which we can attach system-defined tasks.

System Context

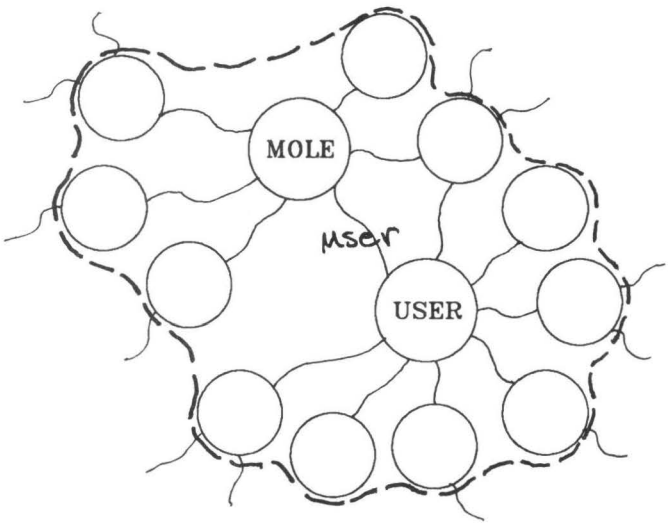
In MOLE statements are always made in the context of each user, and the 'world' provided by the system — other users, other machines (e.g. a drawing machine), and system functions. Initially, therefore, context is provided by:

a) the system kind *MOLE* which describes aspects of the system, and

b) *user-kinds* for each user of the system, under which users' descriptions may be referenced.

Apart from the system kind, *MOLE*, kinds are unique to each user. The figure below shows the initial context for one user's statements. Note that the kind *USER* represented in the figure will be replaced by a unique kind name for each user of the system.

FIGURE 8: SYSTEM CONTEXT



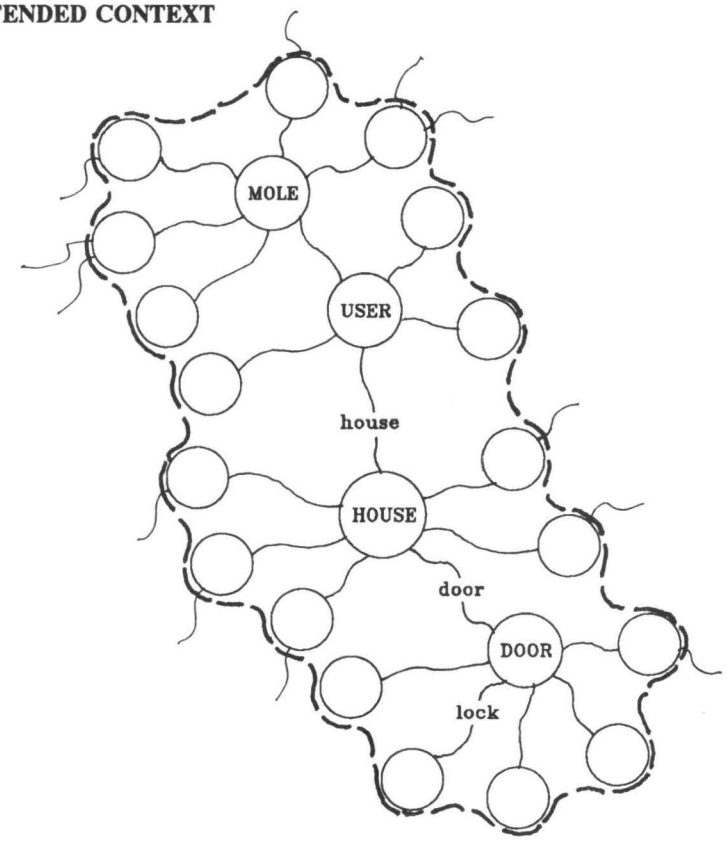
When a user creates a kind, the system attaches that kind to the user's world via a slot which takes the same name as the kind. For example, a user's kind called *HOUSE* will be attached to their user-kind via a slot *house*. In theory, therefore, it is possible to accommodate multiple views corresponding to different users' perceptions of objects and tasks. However, a user may refer to another user's definitions.

The context created by paths, therefore, begins not with the initial kind in the path but with the system kind, *MOLE*. Figure 9 shows the full context for the path *HOUSE* : *door* : *lock*.

Invoking Tasks

Tasks may be invoked anywhere in MOLE provided they are given a context in which they can resolve their references. For example, a simple task *AREA* might require values for *length* and *breadth*, and return its result through a slot *result*. Typically this task might appear within a larger description

FIGURE 9: EXTENDED CONTEXT



RECTANGLE :
length = 4
breadth = 5
area = *AREA* : *result*

such that *length* and *breadth* are defined within the context *RECTANGLE* : *area*. In some cases, however, we may wish to perform this task outside a specific context, which creates the problem of how to provide values for *length* and *breadth*. There are two ways to achieve this: first, we can add *length* and *breadth* to the user's context, i.e.

MOLE> [*length* = 4, *breadth* = 5].
== *USER*

or second, we can include references to positional arguments in the description of *AREA*:

length = #1
breadth = #2

which allows us to invoke the task as

MOLE> *AREA*(4, 5) : *result*.
== 20

This does not exclude the previous method of invoking the task because if the actual arguments are missing the system will search the context for values. In all cases, arguments are evaluated before they are passed to a function.

System Functions

The extended system context presented earlier reflects the need to provide specialised functions beyond the general scope of MOLE. External "machines" can be provided to perform whatever tasks a user may wish. We recognise, however, that certain built-in functions are needed to make the system useful; for example, mathematical functions. We will use the "mathematics machine" here to illustrate the concept of system provided functions.

When extending MOLE we are faced with a choice: do we use a single name to access the added functions or do we attach functions directly to the system kind? For the general purpose mathematics machine we decided to attach its functions directly to avoid having to include the name of the machine when invoking its functions. Thus mathematical functions are addressed via paths of the form

MOLE : function–name

for example, *MOLE : plus.*

At one level system functions appear much as user's functions and can be invoked as described above. Here is a description of the multipication function provided by the "mathematical machine" in MOLE.

PRODUCT :
  *op*1 = #1
  *op*2 = #2
  result = < system call >

A consistent naming convention is observed for all mathematical functions such that the result of the function is always obtained by examining a slot called *result.* The result must ultimately rely on system definitions of these operations. The modified description of *AREA* becomes:

AREA :
  length = #1
  breadth = #2
  result = *PRODUCT*(*length, breadth*) : *result*

which invokes *PRODUCT* to compute the result. There are other valid descriptions of this same task which will produce the same result, e.g.

AREA :
  *op*1 = #1
  *op*2 = #2
  result = *PRODUCT* : *result*

and

AREA :
  result = *PRODUCT*(#1, #2) : *result*

Sequence of Evaluation

Deciding the sequence of evaluation of a task description touches on the problems of ordering mentioned in the section discussing the representation scheme. In the tasks presented above we assume that the sequence of evaluation is controlled by dependencies established by expressions, such that references to other expressions indicate that they must be resolved

before the current expression can be evaluated. For example, given

area = *AREA* : *result*

and

volume = *PRODUCT*(*area*, *height*)

within a description (say) *ROOM*, the system will have to resolve the reference to *area* before it can compute *volume*, as *area* is explicitly referred to in the expression filling the slot *volume*.

Conditional Fillers

To complete the functionality required to provide a useful task description environment for users we have added one further mechanism: conditional fillers. These provide a means of controlling the path of an evaluation by choosing the filler of a slot based on tests applied to known values at the time of evaluation. The general form of all fillers becomes

$F_1 < \{ conds_1 \} < ; F_2 \{ conds_2 \} ... > >$

where a filler is optionally followed by a condition and more (optionally conditional) fillers separated by disjunction. The principle is best illustrated by example.

Suppose we want to provide a general description for *DOOR* which includes properties specific to internal and external doors. Conditional fillers can be used to choose the correct propertiies depending on the type of door.

DOOR:
  width = 800 { *DOOR* : *internal* } ;
  900 { *DOOR* : *external* } ; 750

In the above the type is determined by the presence of slots *internal*, *external*, or neither. The result of *DOOR* : *width* will either be 800, 900, or 750. This operates as a replacement rule.

The implementation of tasks as described here has yet to be completed, and so we expect to make some changes to the theory. We expect problems to arise in dealing with the order of evaluation, and in defining adequate control procedures for task execution. We intend to allow recursive definitions, but this obviously increases the complexity of implementation, since the system must be able to recover from infinite recursion by recognising non-terminating task specifications.

A FINAL WORD

In this paper we have described key features of the MOLE system and highlighted some of the major decisions. MOLE has evolved over a period of 5 years and is now in its final stages. We have gained much from the experience and thus have satisfied our primary goal of exploring descriptive techniques, using MOLE as a vehicle. Our intentions for the system have never ventured beyond this goal, as we have felt that too many fundamental questions remain unanswered. MOLE is just a starting point.

ACKNOWLEDGEMENTS

Work on MOLE takes place within the total research environment at EdCAAD, we are thus indebted to all our colleagues for their ideas and criticism. The work described in this paper has been supported by the Science and Engineering Research Council of the United Kingdom.

References

1. A Bijl, D Stone, and D Rosenthal, "Integrated CAAD Systems", EdCAAD Final Report, Edinburgh (March 1979).
2. A Bijl, "Strategies for CAD", pp. 2-19 in *Intelligent CAD Systems 1: Theoretical and Methodological Aspects*, ed. P.J.W. ten Hagen and T. Tomiyama, Springer-Verlag, Berlin (1987).
3. R Krishnamurti, "Representing Design Knowledge", *Planning and Design*, ().
4. R Krishnamurti and A C Tweed, "Theory and Practice of Descriptions", *Environment and Planning B*, (forthcoming 1988).
5. A C Tweed, "Dynamic Loading in C-Prolog", EdCAAD Technical Report (1985).
6. A Bijl, "An Approach to Design Theory", *proc. IFIP WG 5.2 Working Conference on Design Theory for CAD*, (Oct. 1985).
7. A C Tweed, "A Computing Environment for CAAD Education", *Proc. eCAADe Fourth European Conference on Teaching and Research Experience with CAAD*, (Sept. 1986).
8. A C Tweed, "MOLE User's Manual Version 1.0", EdCAAD Working Paper, Edinburgh (Aug. 1985, revised Mar. 1986).

Paper Session:

Application Modules and System Architecture

The Power of Physical Representations

V. Akman
P.J.W. ten Hagen

The Power of Physical Representations

Varol Akman
Paul J. W. ten Hagen
Department of Interactive Systems
Center for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract Commonsense reasoning about the physical world as exemplified by "Iron sinks in water" or "If a ball is dropped it gains speed" will be indispensable in future programs. We argue that to make such predictions (viz. envisioning), programs should use abstract entities (such as the gravitational field), principles (such as the principle of superposition), and laws (such as the conservation of energy) of physics for representation and reasoning. This is along the lines of a recent study in physics instruction where expert problem-solving is related to the construction of physical representations that contain fictitious, imagined entities such as forces and momenta [cf. Jill H. Larkin, "The role of problem representation in physics," pp. 75-97 in *Mental Models*, ed. D. Gentner and A.L. Stevens, Erlbaum, Hillsdale, N.J. (1983)]. We give several examples showing the power of physical representations.

Categories and Subject Descriptors I.2.4 [Artificial Intelligence]: Knowledge Representation Formalisms and Methods

Additional Key Words and Phrases Envisioning, Physics laws, Physical entities, Physical representations

1. Introduction

Leibniz's *An Introduction to a Secret Encyclopedia* includes the following marginal note [18]:

> Principle of Physical Certainty: Everything which men have experienced always and in many ways will still happen: e.g. that iron sinks in water.

In our daily lives we use this principle routinely. Thus, we know that we can pull with a string but not push with it, that a flower pot dropped from our balcony falls to the ground and breaks, that when we place a container of water on fire, water may boil after a while and overflow the container.

The origin of such knowledge is a matter of constant debate. It is clear that we learn a great deal about the physical world as we grow up. However, even philosophers were always tricked by the mechanisms which achieve this [10]:

> All our knowledge begins with sense, proceeds thence to understanding, and ends with reason, beyond which nothing higher can be discovered in the human mind for elaborating the matter of intuition and subjecting it to the highest unity of thought. At this stage of our inquiry it is my duty to give an explanation of this, the highest faculty of cognition, and I confess I find myself here in some difficulty.

In this paper, we shall argue that some difficulties regarding commonsense reasoning about the physical world can be overcome by using fictitious entities, laws, and principles of physics. This is along the lines of a recent study in physics instruction where expert problem-solving is attributed to the construction of physical representations that contain imagined entities [17].

1.1. Mental Models

Why do we advocate that a theory of commonsense reasoning about the physical world should be firmly based on physics? While we see the psychological literature on this subject as a valuable source of information, studies in learning show, maybe expectedly, that human subjects have fuzzy and even wrong ideas about the physics of everyday life. DiSessa [4] found out that a group of elementary school students learning to control a computer-simulated Newtonian object invariably had the wrong Aristotelian expectation that bodies must move in the direction they are last pushed. Another similar study by McCloskey [21] reports that assumptions of the naive theories of motion are quite consistent across college students. It turns out that the theories developed by different individuals are best described as different forms of the same basic theory. What is striking is that this basic theory is highly *inconsistent* with the fundamental principles of classical physics. McCloskey shows that this naive theory of motion is similar to a pre-Newtonian physical theory: the medieval impetus theory that the act of setting a body in motion "imprints" in the object a force, or impetus, that keeps the object in motion. Figure 1 illustrates a case examined by McCloskey. Asked about how a metal ball put into the end of the tube and shot out of the other end at high speed would behave, most subjects have drawn the incorrect path.

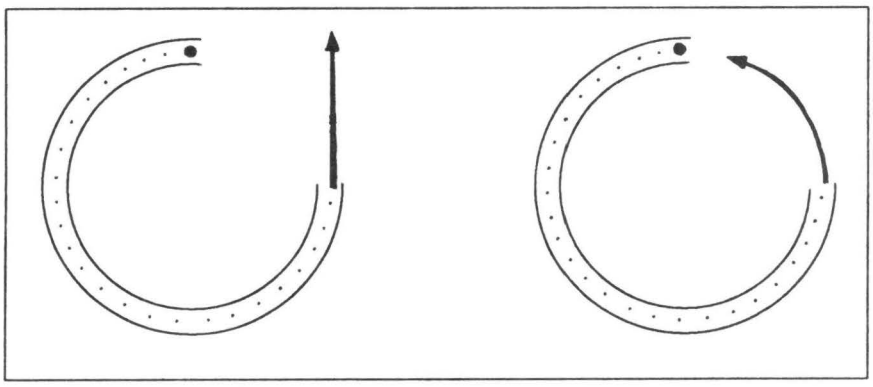


Figure 1. Correct and incorrect responses for the spiral tube problem

1.2. Motivation

Why is commonsense physical knowledge useful? In this paper we shall be concerned with mechanics. Therefore we may answer this question from a viewpoint focusing on mechanical design. We identify mainly two aspects:

- Mechanical design results in physical objects. After production a design object is left in the physical world. From that moment on it will interact with an environment where there are physical notions like force, motion, collision, etc. in existence. If we are interested in reliable products which continue functioning correctly under various disturbances, we must take heed of these notions and their effects on our design objects. Thus we may want to know, during the design stage, what happens to a nuclear reactor when a pressure regulator starts to malfunction. Depending on the outcome of such simulations we may embed more security checks and redundancies in our designs.
- All mechanical inventions are firmly based on a deep understanding of the physical world and its laws. If we want to design new devices automatically we need a design system

which has an appreciation for physical phenomena. For example, even a simple can opener is a device unifying diverse physical notions such as friction, force, rotation, and cutting. When human designers invent a new device they use their physics knowledge in a fundamental way to reason about the functioning of the device under consideration.

2. The Nature of Physics

Observing that all of our ideas in physics require a certain amount of commonsense in their application we see that they are not purely mathematical or abstract ideas. In fact, nearly every page of [5] has a caveat to this effect as the following excerpts show:

> This system [a system of discourse about the real world] is quite unlike the case of mathematics, in which everything can be defined, and then we do not *know* what we are talking about. In fact, the glory of mathematics is that *we do not have to say what we are talking about.* The glory is that the laws, the arguments, and the logic are independent of what "it" is. If we have any other set of objects that obey the same system of axioms as Euclid's geometry, then if we make definitions and follow them out with correct logic, all the consequences will be correct, and it makes no difference what the subject was.
>
> · · · [W]e cannot just call $F = ma$ a definition, deduce everything purely mathematically, and make mechanics a mathematical theory, when mechanics is a description of nature. By establishing suitable postulates it is always possible to make a system of mathematics, just as Euclid did, but we cannot make a mathematics of the world, because sooner or later we have to find out whether the axioms are valid for the objects of nature. Thus we are immediately get involved with these complicated and "dirty" objects of nature, but with approximations ever increasing in accuracy.

Let us now study in detail why we cannot make mechanics a mathematical theory.

2.1. The Meaning of Physical Laws

Centuries of scientific activity gave rise to an enormous body of physical knowledge which can be found in text books. The aim is to provide an account of how the physical world behaves. Theoreticians found out that mathematics is an excellent tool for physics since all laws can be written in symbolic form with absolute clarity and economy. Yet physical formulas by themselves do not provide enough insights [5]:

> Although it is interesting and worthwhile to study the physical laws simply because they help us to understand and to use nature, one ought to stop every once in a while and think, "What do they really mean?" The meaning of any statement is a subject that has interested and troubled philosophers from time immemorial, and the meaning of physical laws is even more interesting, because it is generally believed that these laws represent some kind of real knowledge. The meaning of knowledge is a deep problem in philosophy, and it is always important to ask, "What does it mean?"
>
> Let us ask, "What is the meaning of the physical laws of Newton, which we write as $F = ma$? What is the meaning of force, mass, and acceleration?" Well, we can intuitively sense the meaning of mass, and we can *define* acceleration if we know the meaning of position and time. We shall not discuss these meanings, but shall concentrate on the new concept of *force*. The answer is equally simple: "If a body is accelerating, then there is force on it." That is what Newton's laws say, so the most precise and beautiful definition of force imaginable might simply be to say that force is the mass of an object times the acceleration. Suppose we have a law which says that the conservation of momentum is valid if the sum of all external forces is zero; then the question arises, "What does it *mean*, that the sum of all the external forces is zero?" A pleasant way to define that statement would be: "When the total

> momentum is a constant, then the sum of the external forces is zero.'' There must be something wrong with that, because it is just not saying anything new. If we discovered a fundamental law, which asserts that the force is equal to the mass times the acceleration, and then *define* the force to be the mass times the acceleration, we have found out nothing. We could also define force to mean that a moving object with no force acting on it continues to move with constant velocity in a straight line. If we then observe an object *not* moving in a straight line with a constant velocity, we might say that there is a force on it. Now such things certainly cannot be the content of physics, because they are definitions going in a circle. The Newtonian statement above, however, seems to be a most precise definition of force, and one that appeals to the mathematician; nevertheless, it is completely useless, because no prediction whatsoever can be made from a definition. One might sit in an armchair all day long and define words at will, but to find out what happens when two balls push against each other, or when a weight is hung on a spring, is another matter altogether, because the way the bodies *behave* is something completely outside any choice of definitions.

For instance, the important thing about force is that it has a material origin. If a physical body is not present then a force is taken to be zero. If we discover a force then we also try to find something in the surroundings that is a *source* of the force. Another rule about force is that it obeys Newton's Third Law: the forces between interacting objects are equal in magnitude and opposite in direction (cf. §4.2). These concepts we have about force, in addition to a mathematical rule like $F = ma$, are the key elements in solving physics problems. It is only through combining the mathematical equations with concepts experts attempt to solve physics problems [17].

Consider, for example, the use of constraints to model physical laws. A constraint such as $f(x_1, \cdots, x_n) = 0$ can be used to determine any x_i if all the others are given a value. However, as de Kleer [11] points out:

> There is much more information in an expression! If all we are interested in is solving a set of equations, looking at constraint expressions may be a valid perspective. However, we are solving a physical problem in which a duality exists between the mathematical structure of the equations and the actual physical situation we have thrown away most of the information. To the sophisticated student this duality is very clear and the mathematical equation is far more than a constraint expression. For him, the expression encodes a great deal of qualitative knowledge and every mathematical manipulation of the expressions reflects some feature of the physical situation.

For example, we shall see in §3.1 that whenever we calculate an imaginary number from a velocity equation, we decide that the body under consideration is not able to reach to a certain point. Similarly, we treat vector equations in a careful manner so that the signs of the involved quantities make sense. If a force is trying to prevent the motion of a body, its direction is taken opposite to the direction of movement. The issue of what signs should be assigned to the quantities under consideration is a first sign of a correct attempt to solve a physics problem.

But there is a more important kind of knowledge encoded in a physics formula. Considering again $F = ma$, F and m should be placed, during problem-solving, at the same conceptual ''layer'' since they can be determined independently from any other quantities [24]. However, a is placed at a higher layer since it is determined by the quantities in the previous layer. That acceleration causes force, although implicit in the above formula when one reads it as ''one unit of acceleration gives m units of force,'' is not meaningful in physics. Quantities like F and m are allowed to be manipulated only externally (exogenous variables) whereas a is derived (endogenous). The only intended meaning of $F = ma$ is that at any given moment, the net force on a body is equal to the product of its mass and acceleration.

2.2. Essential Attributes and Influences

Let us think of a simplifying way of looking at physics problems. We have usually various objects under consideration and we state several things about them. We can use predicates to talk about objects; this follows the advice of Hayes [9]. Thus, $p(a)$ means that an object a has an attribute p. To denote that object a has no attribute p we shall write $\bar{p}(a)$. Predicates can be one-, two-, ... place depending on how many objects are required to form them. Thus the predicate "x is frictionless" is one-place, "x is longer than y" is two-place, "z is faster than x but slower than y" is three-place, and so on. We write, for the preceding examples, in predicate notation $frictionless(x)$, $longer(x,y)$, and $inbetween(z,x,y)$. In general, $p(a_1, \cdots, a_n)$ means that an n-tuple of objects $a_1, \cdots, a_n$ has an attribute p; in other words, $a_1, \cdots, a_n$ such that p holds.

By *abstraction*, we signify the mapping of a set of objects to a new set of objects which have some of their attributes ignored. For instance, we may take a block and map it to a point mass by simply ignoring its spatial dimension. Similarly, we may map a surface to a frictionless surface, a gas to an ideal gas, and a fluid to an incompressible fluid. By doing so we are creating predicates which are unverifiable, e.g. $frictionless(s)$, $ideal(g)$, $incompressible(f)$ denote attributes which are not possible to observe. The objects denoted by abstract terms do not exist empirically because of the way they are constructed [26]; we *decide* to take certain objects and neglect some attributes of them.

Our view is that this is how physics experts start solving problems. That is, the mapping of empirical objects to abstract objects is an important ingredient in solving a physics problem since the physical laws in general are stated over abstract entities and as state transformations. Using the attribute viewpoint above, it is easy to write down transformations. Assume that s_1 is a state of an object at a certain moment and s_2 is a state at a later moment. This transformation will be denoted by $\rightarrow$. To denote that object a exists we may write $E(a)$. Thus $\bar{E}(a) \rightarrow E(a)$ denotes the creation of object a whereas $E(a) \rightarrow \bar{E}(a)$ denotes the death of a. We may write similar statements about the attributes. Thus $\bar{p}(a) \rightarrow p(a)$ stands for acquisition of an attribute while $p(a) \rightarrow \bar{p}(a)$ stands for loss of an attribute by object a. It is messy to tell when an object is created or annihilated. Considering a drop of water, if heat is given to it, the drop will acquire the attribute *hot* and lose the attribute *cold*. However, if enough heat is applied, the drop will turn to steam. There are two possibilities: either this is the same individual which was liquid before or this is a new individual.

By *essential influences* we shall mean those influences which play a central role in a situation. Consider the example of a pendulum clock [5]. If a pendulum clock is standing upright it works as expected, but if it is tilted nothing happens. Something else, something outside the machinery of the pendulum clock, is involved in the operation of the clock. This is the earth which is pulling on the pendulum. Once we encode this effect through the vector of gravitational force, we may take the earth out of the picture and consider our pendulum embedded in a field of gravitation — the essential influence for this problem. It is consequently an easy matter to deduce that e.g. the pendulum will tick with a different rate on the Moon.

Shape is a difficult attribute to deal with. Considering a rectangular block sliding along an inclined plane, one may use the abstraction that the block is a point mass. However if a ball is rolling down on an inclined plane, a point mass viewpoint may lead to an incorrect perspective since e.g. the angular momentum due to rotation is no more accountable. For the sliding block problem we take the mass of the block as a constant since we make the reasonable assumption

that it is going at a low speed (compared to the speed of light) and its mass is independent of time. Similarly, for a block attached to a spring with stiffness constant k, we normally think that if the spring is stretched by x then we can find the force on the spring using $F = -kx$. This is based on the assumption that for small x, k is constant. Note that the fact that the spring can break if the force is exceedingly large is not implicit in this formula.

Another way of looking at physical influences is to regard them as functions. Thus by (fx) let us denote that function f takes object x and gives fx as a new object. This is an intensional definition of a function as opposed to the usual, extensional way. Functional equivalence then becomes $(fx)=(ghx)$, i.e. apply g on the result of applying h to x to obtain a summary effect equivalent to (fx). This notation helps one to formulate state transformations. Consider object x which is a block of ice. Now (fx) where f is heat renders a new object fx which is equal to water. Let us call this object $\bar{x}$. Applying f on $\bar{x}$ gives a new object which is equal to steam. (Thus self-application of functions is sometimes meaningful.)

3. Envisioning

In envisioning we deliberately ignore the values of the problem variables — we let them take any value. This permits us discover all the possibilities for a given problem. Only one of these possible answers is observable for a given set of initial conditions. The idea of envisionment will now be made clearer by studying a pioneering program.

3.1. NEWTON

The first attempt for predicting the interaction of moving objects in a simple, idealized world has been made by Johan de Kleer and a program called NEWTON was implemented. NEWTON works in a 2-dimensional world called the "roller-coaster" world. We start with a classical problem used by de Kleer [11]. In Fig. 2, a small sliding block (idealized as a point) of mass m starts at point c_1 along a frictionless surface consisting of three parts: s_1 and s_2 which are concave, and s_3 which is straight. We want to know if the block will reach point c_4.

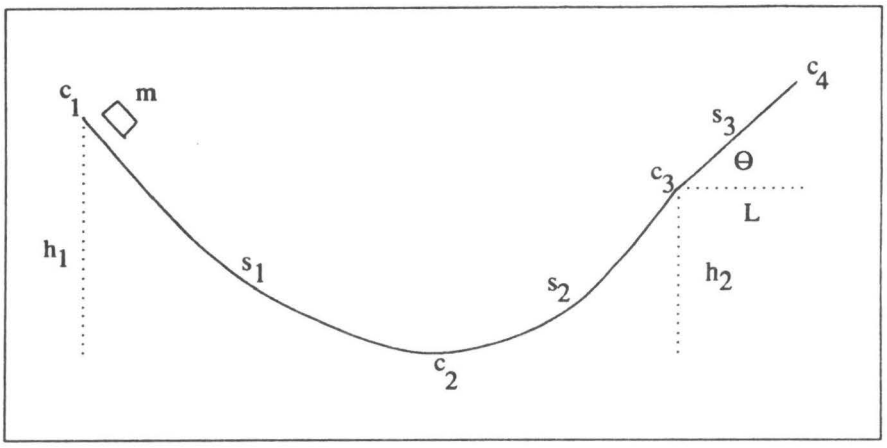


Figure 2. The sliding block problem

In NEWTON, a production rule system looks at the local geometric (topological) features to predict what might happen next. The left hand side of a production describes the features of the environment (e.g. the shape of the paths, the velocity of the block) and the right hand side lists

the consequences (e.g. sliding, falling). A closed world assumption[1] is in effect: the actions on the left hand side never produce a change in the features (e.g. a block sliding on a segment does not cause a change in the shape of the segment).

We may summarize the reasoning suggested by de Kleer as follows. Without falling off or changing direction m will start to slide down the surface. After reaching the bottom c_2, it will start going up. It still will not fall off, but it may start sliding back. If m ever reaches the straight section (i.e. the segment s_3) it still will not fall off there, but it may change the direction of its movement. To determine exactly whether m reaches c_4 we must study its velocity. The velocity at the bottom can be computed by using the conservation of energy: $v_{c_2} = \sqrt{2gh_1}$. Similarly, using v_{c_2} and conservation of energy we can set up an equation which can be solved for the velocity v_{c_3} at the start of the straight section: $\frac{1}{2}mv_{c_3}^2 = \frac{1}{2}mv_{c_2}^2 - mgh_2$. If v_{c_3} is imaginary, we know that the straight segment is never reached. At the straight section we would use kinematics to find out whether the block ever reaches c_4. The acceleration of the block along s_3 must be $a = g\sin\theta$. The length of the straight segment is $\frac{L}{\cos\theta}$, so using the kinematic equation relating acceleration, distance, and velocities we get: $v_{c_4}^2 = v_{c_3}^2 - 2Lg\tan\theta$. Again, if v_{c_4} is imaginary, c_4 is not reachable.

This line of reasoning can be conveniently depicted in the *envisionment* tree of Fig. 3. The branches of the tree correspond to the different situations that may arise depending on the solutions of the equations mentioned above. Basically, we have a set of worlds W and a set of times T. There is a linear ordering $<$ (meaning "prior to") over T. We can then think of a 2-dimensional space where we have differing worlds as we move along one axis and differing times as we move along on the other axis. Any particular point in this space can be thought of as a pair of coordinates $<w, t>$ for some $w \varepsilon W$ and $t \varepsilon T$. This index denotes a point whose location is determined by which world it is on on one hand, and by what time it is on the other hand. The "possible worlds" in Fig. 3 are then as follows, in a left-to-right order:

w_1: Slide until at most c_3, slide backwards, oscillate around c_2.

w_2: Slide until at most c_3, slide backwards, fall off from c_1.

w_3: Slide until after c_3 but before c_4, slide backwards, oscillate around c_2.

w_4: Slide until after c_3 but before c_4, slide backwards, fall off from c_1.

w_5: Slide until c_4, fall.

For example, denoting the moments that m passes through the points c_i by t_i, it is seen that m will be at c_2 at t_2, at some point of s_2 for any t such that $t_2 < t < t_3$, and then for all $t > t_3$ on s_1 or s_2, if it follows the prediction denoted by w_1.

[1] Davis gives the following justification for the closed world assumption [3]: "The "frame" or "persistence" problem of determining what remains true over time requires no special treatment in our logic. We can avoid this problem, not by virtue of any special cleverness on our part, but by virtue of the structure of the domain. The predicates in the domain are divided into two classes. The first class includes predicates which depend on position and velocity of objects. These are not assumed to remain constant over any interval unless they can be proven to be so. The second class includes structural predicates, depending only on the shape and other material properties of the objects. These are defined to be constant over the problem, and so are defined atemporally. Similar considerations would seem to apply to any closed world, complete physical theory; it is not clear why the frame problem should ever create trouble in such a domain."

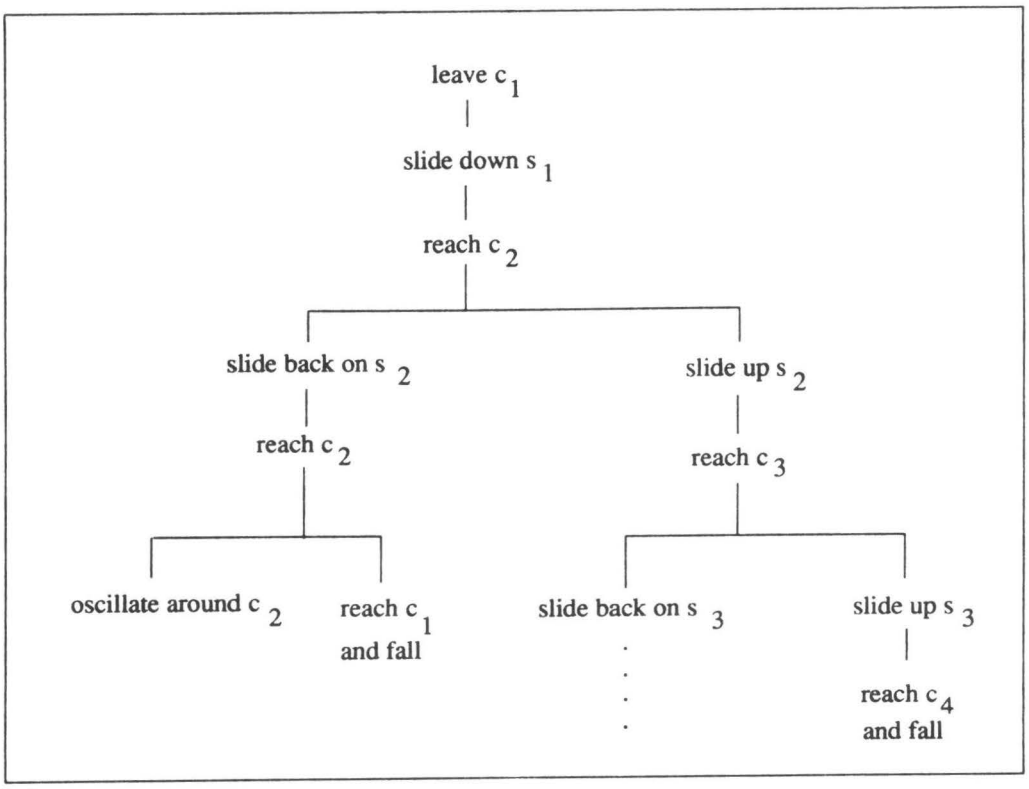


Figure 3. The envisionment tree for the sliding block problem

At each fork of the envisionment tree there are several possibilities. A block would take one of the branches always and would go down the tree, ending up in a leaf. The forks correspond to the points where we have to *disambiguate* the ambiguities whereas leaves correspond to particular behaviors. Disambiguation is achieved by solving some equations. Looking at the formula giving v_{c_2} we see that this velocity can never be imaginary. Thus the block would always reach c_2; it would never stop there since it would, at that point, have a positive velocity. Let us look at the next equation giving the value of v_{c_3}. It can be rewritten, after manipulation, as $v_{c_3}^2 = 2g(h_1 - h_2)$. This equation has a simple message. If c_3 is located higher than c_1 then there is no way that m would reach c_3. Until now we did not really need any numerical knowledge other than $h_1 > h_2$ to decide that m will arrive at c_3. However, after c_3 we need to know the values of L and θ to discover that c_4 is reachable. Assume that m has made it to c_4 barely, i.e. $v_{c_4} \geq 0$. Then the inequality $h_1 - h_2 \geq L \tan\theta$ should be satisfied.

3.2. Naive vs. Physical Representations

NEWTON has been a corner-stone in the search for knowledge representation and reasoning methodology for physical domains. A later work on mechanics problem-solving in the style of NEWTON is MECHO [2]. This program tries to improve the envisioning process by making it more goal-directed. The main improvement is that MECHO does not generate the envisionment tree but only the parts needed[2] to answer a question. Thus, for a suitable set of values of the problem parameters above MECHO would only generate the real path that would be taken by the block, say the left-most.

[2] It is arguable that an envisioner should generate only the relevant parts of an envisionment tree. Goal-directed envisioning is fine as long as questions like "What happens next?" are avoided.

Larkin [17] points out that NEWTON has a naive way of representing physical knowledge. In particular, it has an internal representation which contains direct representations of the visible entities mentioned in the problem description. As a result, it performs simulation — NEWTON's inferencing via an envisionment tree follows the direction of time flow. What is more serious is the lack of deeper physics principles in NEWTON. For instance, Larkin notes that a physicist encountering the above problem about the sliding block might reason as follows [17]. The energy at c_1 consists of kinetic energy, zero because m is at rest, and potential energy determined by h_1. At c_2 the potential energy is zero, because the block is at the bottom and the kinetic energy is unknown because the speed is unknown. At c_3 the potential energy is determined by h_2 and the kinetic energy is still unknown. At some point c (which may be above or below c_4), the block stops[3] and the kinetic energy is again zero. Writing down the equation $0 + mgh_1 = 0 + mg(h_2 + X\sin\theta)$, where X is the distance m travels along s_3 after point c_3, immediately leads to a solution. (Note that Larkin writes $\frac{X}{\sin\theta}$ which is wrong.) Briefly, if $X > \frac{L}{\cos\theta}$ then m will reach c_4 and fall off.

Compared to NEWTON's solution, there are good insights in this solution. The fact that a body has speed is identified with its having kinetic energy. That a body is at a given height is identified with its having potential energy. In writing down the preceding equation, the absence of friction (and hence no loss of energy due to heat dissipation) is used to state a simpler law of conservation of energy. The fact that potential energy and kinetic energy are convertible to each other is used implicitly. The expert is also aware that m, which is initially at rest, is brought to motion by the gravitational field of the earth which is constant and equal to g in distances not too far away from the surface of earth. As another pointer to the use of deep knowledge, consider the decision of the expert to select c_2 as a "standard" point for potential energy calculation. If he had used any other point, say c_3, instead of c_2, it is obvious that the potential energy is changed only by the addition of a constant. Since the energy conservation law cares only about *changes,* it does not matter if a constant is added to the potential energy.

NEWTON's use of quantitative knowledge (represented as frames) to disambiguate ambiguities remains as a very important contribution. Frames in NEWTON are not procedures but describe dependencies among variables. They are similar to Minsky's frames [22] in that they are used to chunk physical formulas of the same nature. Since there are many different equations applicable to a problem, grouping equations in frames help isolate the relevant ones with less effort. Dependencies among variables are then searched for a solution (the goal variable). For example, a kinematics frame[4] knows about the usual equations [12]:

frame kinematics **of** *object*, *surface*, t_1, t_2
variables:
(v_1: **velocity of** object **at time** t_1,
v_2: **velocity of** object **at time** t_2,
d: **distance of** *surface*,
t: **time between** t_1 **and** t_2,
a: **acceleration of** *object*)

[3] It is incorrect (contrary to what [17] claims) to think that the particle stops. Since there is no friction the particle may oscillate indefinitely. Thus, an expert would only consider the extremal points of the oscillation where the kinetic energy is indeed zero.

equations: $(v_2 = v_1 + at;\ v_2^2 = v_1^2 + 2ad;\ d = v_1 t + \frac{1}{2}at^2)$.

Frames use two kinds of variables: the names of the objects and their essential attributes (such as velocity or acceleration). Each equation of a frame referencing the goal variable is a possible way to determine that variable. On the other hand, unknown variables in equations referencing the goal variable must be given a value (possibly using other frames or asking the user) since every unknown in the equation must be determined before achieving the goal.

4. The Content of Mechanics

This section gives, rather tersely, the building blocks of classical mechanics. It is neither complete nor uniform as it stands. However, it should give an idea about what kind of knowledge should be formalized for a deep coverage of mechanics. It is noted that we are not yet concerned with a concrete knowledge representation scheme. This is thought to be in agreement with [9] where the building of mini-theories (clusters) are given priority over favorite notations: ''Initially, the formalizations need to be little more than carefully worded English sentences. One can make considerable progress on ontological issues, for example, without actually formalizing anything, just by being *very* careful what you say.''

4.1. Fictitious Entities

Description of Motion: Speed as a derivative $(v = \lim_{\Delta t \to 0} \frac{\Delta s}{\Delta t})$, distance as an integral $(s = \lim_{\Delta t \to 0} \sum_i v(t_i)\Delta t)$, and acceleration as the derivative of speed are the basic notions. We distinguish velocity from speed, which is the magnitude of velocity. (We shall sometimes denote vectors with bold letters.)

Pseudo Vectors: Ordinary vectors are e.g. the coordinate, force, momentum, velocity, etc. Vectors which are obtained as a cross product are artificial, e.g. torque τ, angular velocity ω, and angular momentum $\mathbf{L}$. The corresponding equations are $\tau = \mathbf{r} \times \mathbf{F}$, $\mathbf{v} = \omega \times \mathbf{r}$, and $\mathbf{L} = \mathbf{r} \times \mathbf{p}$, respectively, where $\mathbf{p}$ is the momentum and $\mathbf{r}$ is the radius.

Work: If a body is moving along a curved path, then the change in kinetic energy as it moves from one point (*1*) to another (*2*) is equal to $\int_1^2 \mathbf{F}.d\mathbf{s}$.

Torque: Torque bears the same relationship to rotation as force does to linear movement, i.e. a torque is the thing which makes something rotate or twist.

Energy: In mechanics problems, energy takes basically the following forms: gravitational energy, kinetic energy, heat energy, and elastic energy.

Power: Power equals work done per second. In other words, the rate of change of kinetic energy of a body is equal to the power used up by the forces acting on it.

Field: We need two kinds of laws for a field. The first law gives the response to a field, i.e. the equations of motion. The second law gives how strong the field is, i.e. the field equation. In other words, ''One part says that something *produces* a field. The other part says that something is *acted on* by the field. By allowing us to look at the two parts independently, this separation of

[4] De Kleer published a revised, shorter version of [11] later in [12]. An interesting decision has been made in the second version to call the RALCMs (Restricted Access Local Consequent Methods) of the first version FRAMEs.

the analysis simplifies the calculation of a problem in many situations'' [5].

The idea of a field is closely associated with potential energy. We note that the gravitational force on a body is written as mass times a vector which is dependent only upon the position of the body: $\mathbf{F} = m\,\mathbf{C}$. This lets us analyze gravitation by imagining that there is a vector $\mathbf{C}$ at every position which effects any mass placed there. Since the potential energy can be written as $m\int(\textbf{field}).(d\,\mathbf{s})$, we find that the potential energy of a body in space can be written as mass times a function Ψ, the potential. To get the force from the potential energy we use: $F_x = -\frac{\partial U}{\partial x}$ and similarly for other directions. To get the field from the potential we do the same thing: $C_x = -\frac{\partial \Psi}{\partial x}$ and similarly for other directions. More succinctly, $\mathbf{F} = -\nabla U$ and $\mathbf{C} = -\nabla\Psi$.

Pseudo Force: A well-known pseudo force is what is often called *centrifugal* force. If we are in a rotating coordinate system, we experience a force throwing us outward. Using the pseudo force we can explain several interesting problems (Fig. 4). In this figure, adapted from [5], a container of water is pushed along a table, with acceleration a. The gravitational force acts downward but there is in addition a pseudo force acting horizontally. The latter is in a direction opposite to a. As a result, the surface of the water will be inclined at an angle with the table, as shown in the left part of the figure. If now we stop applying a push, the container will slow down because of friction, and the pseudo force will reverse its direction, causing the water stand higher in the forward side of the container.

Among the forces that are developed in a rotating system, centrifugal force is not the only one. There is another force called *Coriolis* force. It has the property that when we move a body in a rotating system, the body seems to be pushed sidewise. If we want to move something radially in a rotating system, we must also push it sidewise with force $F_c = 2m\,\omega v_r$. Here ω is the angular velocity and v_r is the speed the body is making along the radius.

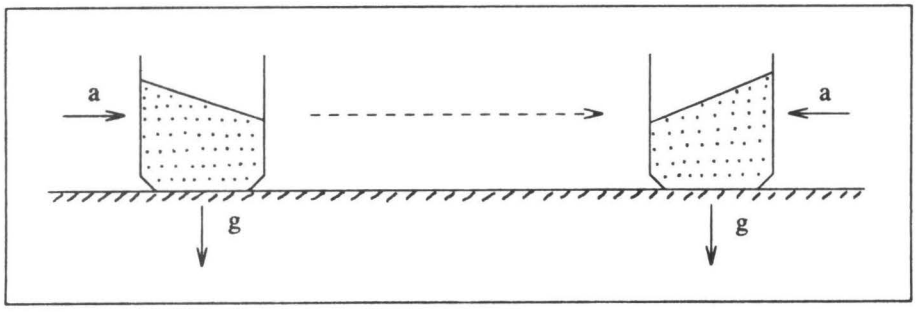


Figure 4. Pseudo force acting on a container of water

Conservative Force: If the integral of the force times the distance in traveling from a point to another is the same (regardless of the shape of the curve connecting them, and for this pair of points on every curve) then the force is *conservative* (e.g. gravity).

Center of Mass: Given a rigid body there is a certain point such that the net external force produces an acceleration of this point as if the whole mass is concentrated there. The point does not have to be in the material of the body but can lie outside.

4.2. Laws of Mechanics

Hooke's Law (The Law of Elasticity): The force in a body which tries to restore the body to its original condition when it is distorted is proportional to the distortion. This holds true if the distortion is small. If it gets too large the body will be torn apart or crushed.

Newton's Laws of Dynamics:

First Law (Principle of Inertia): If an object is left alone (not disturbed), it continues to move with a constant velocity in a straight line if it was originally moving, or it continues to stand still if it was standing still.

Second Law: The motion of an object is changed by forces in this way: the time-rate-of-change of momentum is proportional to the force.

Third Law ("Action equals reaction"): Suppose we have two small bodies and suppose that the first one exerts a force on the second one, pushing it with a certain force. Then simultaneously the second body will push on the first with an equal force, in the opposite direction. These forces effectively act in the same line.

Conservation of Linear Momentum: If there is a force between two bodies and we calculate the sum of the two momenta, both before and after the force acts, the results should be equal.

Conservation of Angular Momentum: If no external torques act upon a system of particles, the angular momentum remains constant. In other words, the external torque on a system is the rate of change of the total angular momentum: $\tau_{ext} = \frac{d\,\mathbf{L}_{tot}}{dt}$.

Conservation of Energy: There is an abstract quantity that does not change in all the natural phenomena which the world undergoes: energy.

(Galilean) Relativity Principle: The laws of physics look the same whether we are standing still or moving with a uniform speed in a straight line.

Work Done by a Force: If the force is in one direction and the object on which the force is applying is displaced in a certain direction, then only the component of force in the direction of the displacement does any work: i.e. physical work is expressed as $\int \mathbf{F}.d\,\mathbf{s}$.

Work Done by Gravity: The work done in going around a path in a gravitational field is zero. This implies that we cannot make perpetual motion in a gravitational field.

4.3. Principles

Superposition: The total field due to all the sources is the sum of the fields due to each source. Suppose that we have a force F_1 and have solved for the forced motion. Then we find out that there is another force F_2 and solve for the other forced motion. Using the superposition of solutions we can now predict what would happen if we had F_1 and F_2 acting together. The solution is $x_1 + x_2$ if x_i's are the individual solutions for forces F_i, respectively. In general, a complicated force can be divided into a set of separate forces each of which is simple (in the sense that we can solve for the forced motion they cause).

Equivalence of Simple Harmonic Motion and Uniform Circular Motion: Uniform motion of a body in a circle is closely related to oscillatory up-and-down motion. Although the distance y means nothing in the oscillator case, it can still be artificially given in order to model oscillation in terms of circular motion.

Virtual Work: We imagine a structure moves a little — even though it is not really moving or

even movable. We use this small imagined move to apply the law of energy conservation. This principle is especially useful in problems of the sort depicted in Fig. 5 where we are asked to find the value of weight W such that the system is in equilibrium. Noting that a small move of W toward the bottom should be counteracted by weights W_1 and W_2, we find $hW = \frac{d_1}{d} hW_1 + \frac{d_2}{d} hW_2$, or $W = \frac{W_1 d_1 + W_2 d_2}{d}$.

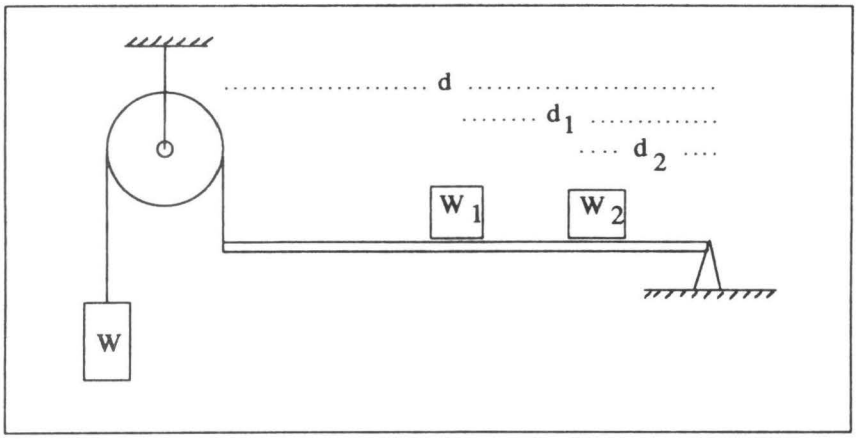


Figure 5. A system of blocks in equilibrium

Analogs: We have already seen examples of analogs when we talked about force vs. torque, linear vs. angular momentum, etc. above. In general, analogy refers to relating two domains which are at first sight dissimilar and using the tools of one domain to solve the problems of the other. The analogy between mechanics and electricity is well-known, as Table 1 partially illustrates [5]. As a result, not only in problem-solving but also in design, analogy has found its deserved place. The following excerpt is especially illuminating in this respect [5]:

> Suppose we have designed an automobile, and want to know how much it is going to shake when it goes over a certain kind of bumpy road. We build an electrical circuit with inductances to represent the inertia of the wheels, spring constants as capacitances to represent the springs of the wheels, and resistors to represent the shock absorbers, and so on for parts of the automobile. Then we need a bumpy road. All right, we apply a *voltage* from a generator, which represents such and such a kind of bump, and then look at how the left wheel jiggles by measuring the charge on some capacitor. Having measured it (it is easy to do), we find that it is bumping too much. Do we need more shock absorber, or less shock absorber? With a complicated thing like an automobile, do we actually change the shock absorber, and solve it all over again? No!, we simply turn a dial; dial number ten is shock absorber number three, so we put in more shock absorber. The bumps are worse — all right, we try less. The bumps are still worse; we change the stiffness of the spring (dial 17), and we adjust all these things *electrically*, with merely a turn of a knob.

Table 1

General Characteristic	Mechanical Property	Electrical Property
independent variable	time (t)	time (t)
dependent variable	position (x)	charge (q)
inertia	mass (m)	inductance (L)
force	force ($F = ma$)	voltage (V)
velocity	velocity (v)	current (I)
resistance	drag coefficient ($c = \gamma m$)	resistance ($R = \gamma L$)
stiffness	stiffness (k)	(capacitance)$^{-1}$ ($1/C$)
period	$t_0 = 2\pi\sqrt{m/k}$	$t_0 = 2\pi\sqrt{LC}$

5. Some Examples

In this section we give some example problems and their solutions. Our aim is to demonstrate the use of physical representations. We regard the following problems as difficult problems for envisioners and expect that they will constitute part of a test set for future programs.

5.1. Rocket Problems

These are taken from [5]. How fast do we have to send a rocket away from the earth so that it leaves the earth? The problem can be stated as a functional requirement: the rocket must leave the earth. We are now asked to find an attribute (the speed) of the rocket. Notice that many other attributes of the rocket has been left out and a pair of objects, the earth and the rocket, have been identified. The essential influence is the earth's gravitational field.

Kinetic energy plus potential energy of the rocket must be a constant. Let us exaggerate and imagine the rocket in two extreme positions. When it is far away from the earth it will have zero potential energy. Besides its kinetic energy will also be zero since we may assume that it barely left the earth. On the other hand, initially it has the total energy $\frac{1}{2}mv^2 - G\frac{mM}{R}$ where m is the rocket's mass, M is the earth's mass, and R is the earth's radius. The conservation of energy gives $v = \sqrt{2gR}$, where $g = \frac{GM}{R^2}$.

Changing the problem a little, at what speed a satellite should travel to keep going around the earth? It turns out that the conservation of energy is not the right way to approach this problem. A force approach written as an equality between centrifugal and gravitational forces, $\frac{mv^2}{R} = \frac{GMm}{R^2}$, gives $v = \sqrt{gR}$. The reason we have thought that a force expression is more convenient here is due to the nature of the problem; there is a "rotating" object and this guides our search among the mathematical expressions applicable to the problem. This should be compared with de Kleer's search strategy [11].

Now let us take a look at the following problem. A rocket of large mass M ejects a small piece of mass m with a velocity V relative to the rocket. Assuming it were standing still, the rocket now gains a velocity v. Using the law of conservation of momentum, this velocity is seen to be $v = \frac{m}{M}V$. Thus, rocket propulsion is essentially the same as the recoil of a gun. It does not need air to push against [5].

Let us suppose that the two objects are exactly the same, and then we have a little explosion between them. After the explosion one will be moving, say toward the right, with a velocity v.

Then it appears reasonable that the other body is moving toward the left with a velocity v, because if the objects are alike there is no reason for right or left to be preferred and so the bodies would do something that is symmetrical.

5.2. The Ball with Strings

Consider Fig. 6 which is taken from den Hartog [8]. A heavy ball of weight W is suspended by a thin thread and has an identical thread hanging down from it. When we start pulling down on the lower string, which of the two strings will break first?

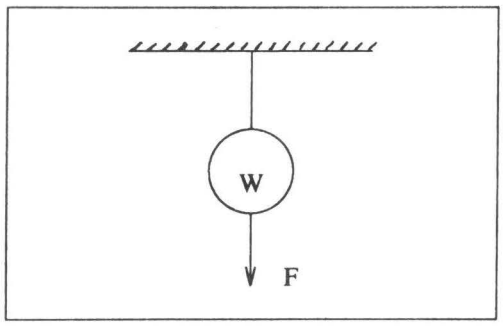


Figure 6. The ball with strings

We assume that we apply force F. Then the bottom string experiences F whereas the upper string experiences $W+F$. Thus the upper string will break first. According to den Hartog, this happens only if we pull down slowly. If we instead give a sudden, sharp pull to the lower string, it will break and the ball remain suspended. This has to do with the fact that the threads are elastic and have a certain elongation associated with the force sustained by them. By giving a quick pull to the lower thread, the force in the lower thread can be made quite large and this force will accelerate the ball downward. But this takes some time and before any appreciable downward displacement is observed the string is broken.

Den Hartog relates a similar experiment. Consider a ball, with a single string attached to it, lying on a table. By a slow pull, one can drag the ball on the table with a uniform speed. In this case the applied force is equal to the friction force between the ball and the table. A quick pull, on the other hand, will break the thread in an instant. The ball is subjected, for a short time, to a large force which subjects it to acceleration. However the ball will hardly move since the time interval is small. Instead its velocity will be destroyed by the retarding action of the friction force.

5.3. The Open Water Jar

A major weakness of the current envisioners is their inability to switch between macro- and micro-worlds. In other words, the individuals that an envisioner knows about are either in the world as we see it or underlying the world (atomic processes). To our best knowledge, there is no current envisioner which can predict that e.g. water would evaporate if it is left in an open jar. Physicists think that our knowledge of atoms, that all things are made of atoms, particles that move around in perpetual motion, attracting or repelling each other depending on the distance between them, gives us very useful information [5]. The atomic hypothesis describes processes.

For example, let us observe the above jar with some water in it. What will happen as time passes? The water molecules are constantly moving around. From time to time, one on the surface is hit strong enough so that it flies out. Thus, molecule by molecule the water evaporates.

If we cover the jar with a lid, we observe no change because the number of molecules leaving the surface are equal to the number of those coming back. Thus, in the long run nothing happens. Note on the other hand that we should find a large number of molecules amongst the air molecules. If we take the lid away and push dry air instead of the moist air, again water will evaporate. The number of molecules leaving the surface is still the same but not as many are coming back.

5.4. The Heated Rubber Band

This is taken from [5] and is another good example as to the convenience of thinking in terms of atomic processes. If we apply a gas flame to a rubber band holding a weight, the band contracts abruptly, as shown in Fig. 7. How can we explain this?

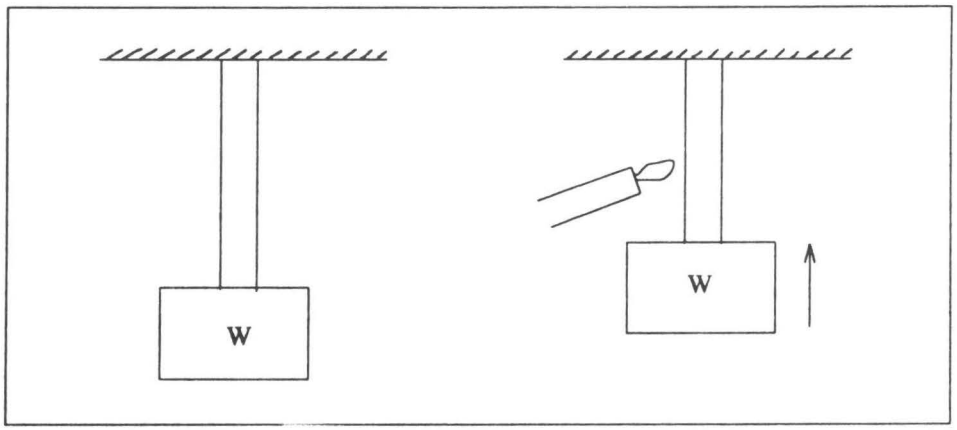


Figure 7. The heated rubber band

A molecular explanation would be as follows. Rubber band consists of a tangle of long chain of molecules, not unlike a molecular spaghetti, with cross links. When such a tangle is pulled out, some of the chains line up along the direction of pull. At the same time chains are hitting each other continually, due to thermal motion. Thus, if a chain is stretched, it will not by itself remain stretched; it would be hit from the sides by other chains which would tend to unstretch it again. When we heat the rubber band, we increase in effect the amount of bombardment on the sides of the chains.

Using thermodynamics, the above explanation can be made more quantitative. When heat ΔQ is delivered, the internal energy is changed by ΔU and some work is done. The work done by the rubber band is $-F\,\Delta L$ where F is the force on the band and L is the length of the band. Note that F is a function of temperature T and L. Now we have $\Delta U = \Delta Q + F\,\Delta L$. Additionally, we have $\Delta Q = -T\frac{\partial F}{\partial T}\,\Delta L$, which tells us that if we keep the length fixed and heat the band, we can calculate how much the force will increase in terms of the heat needed to keep the temperature constant when the band is stretched a bit.

6. Other Related Research

A discussion of naive physics and qualitative physics is given in the dissertation of Schmolze [23]. Here we offer a shorter discussion for completeness and refer the reader to the above reference for an analysis of these areas (e.g. we omit a discussion of Forbus' *Qualitative Process Theory*, nonwithstanding its importance [6]). The articles by de Kleer and Brown [13], Kuipers [14, 15], Forbus [6], Forbus and Gentner [7], and Hayes [9] form the basis of naive and qualitative physics. Incidentally, it is not easy to delineate the areas covered by these terms; we propose that naive physics should be understood as the construction of knowledge representation methods while qualitative physics should cover reasoning techniques.

Patrick Hayes proposed, by the term *naive physics*, the construction of a formal, sizable portion of commonsense knowledge about the physical world [9]. This should include knowledge about objects, shape, space, movement, substances, time, etc. As for the knowledge representation language to be used Hayes is not specific; a collection of assertions in logic, for example, may be sufficient (cf. §2.2). He is not, at this preliminary stage, interested in the efficiency of reasoning with this body of knowledge. What he is really after is "the extent to which it [a naive physics theory] provides a vocabulary of tokens which allows a wide range of intuitive concepts to be expressed, to which it then supports conclusions mirroring those which we find correct or reasonable" [9].

Qualitative physics provides an account of behavior in the physical world. Unlike conventional physics qualitative physics predicts and explains behavior in qualitative terms. Although the behavior of a physical system can be given by the precise values of its variables (temperatures, velocities, forces, etc.) at every moment, such a description fails to provide insights into how the system works. Important concepts causing change in physical systems are concepts like momentum, force, feedback, etc. which can be understood intuitively [13]. They are in conventional physics embedded in a framework of continuous differential equations. In qualitative physics each measurable property such as the speed of a ball is represented in two parts: a quantity plus its rate of change. The representation is qualitative since the quantity values are selected from a discrete quantity space. For example, the quantity space for water may have only two values: the freezing and the boiling temperatures. Rates of change are in general limited to three cases: increasing, decreasing, and constant.

6.1. ENVISION

De Kleer and Brown [13] introduced qualitative differential equations (confluences) and implemented them in a program called ENVISION. To obtain confluences, we let continuous variables take discrete values from a quantity space such as $\{0, +, -\}$. Let $[x]$ denote the qualitative value of an expression x with respect to the quantity space. Then the proposition "x is increasing" is written as $\frac{dx}{dt} = +$ and we can define arithmetic to deal with $[x]+[y]$ or $[x][y]$, although there will be cases where ambiguities will have to be resolved using numeric values (e.g. when the operation is addition and $[x]=+$ and $[y]=-$). Let ∂x denote $[\frac{dx}{dt}]$. Using confluences, we can provide explanations as follows. Consider the pressure regulator in Fig. 8 which is adapted from [13]. A confluence such as $\partial P + \partial A - \partial Q = 0$ where P is the pressure across the valve, A is the area available for flow, and Q is the flow throughout the valve describes this device in qualitative terms. It is seen that an increase in pressure at a causes an increase in pressure at point b. This generates more flow through b. As a result pressure at c increases and this is felt at d. Then the diaphragm e will press downward, causing the valve to close somewhat. As a result,

constant pressure will be maintained at c although the pressure at a is fluctuating. Since a single confluence may not characterize the behavior of a component over its entire range of operation, the range is divided into subregions, each characterized by a different component state where different confluences apply. When the valve is closed the correct confluence should read $\partial Q = 0$; we simply do not say anything about P. Similarly, when the valve is open the confluence becomes $\partial P = 0$.

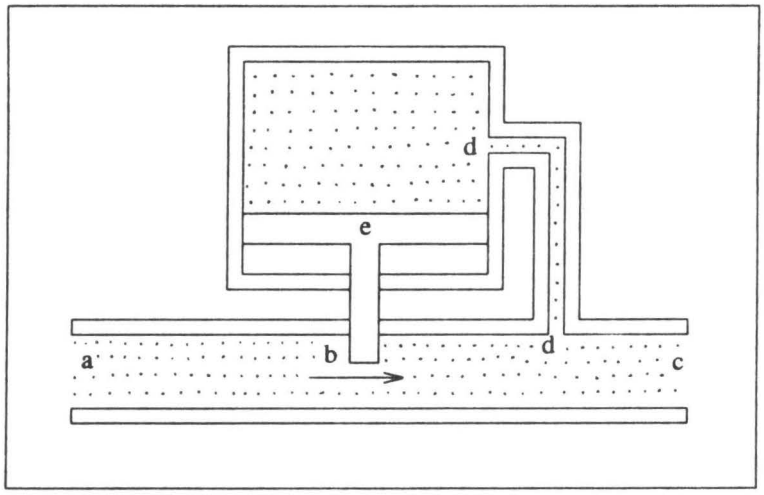


Figure 8. A pressure regulator

ENVISION has a component library where the components relevant to the domain of reasoning are stored. Another module holds the "topology" of the device under consideration. The input to ENVISION are the input signals and boundary conditions. The output is a behavioral prediction along with a so-called causal explanation. De Kleer and Brown's principle of *no-function-in-structure* requires that the laws of the parts of the device may not presume the functioning of the entire device. Various class-wide assumptions are made to avoid tricky situations, e.g. in fluid flow there are always enough particles in a pipe so that macroscopic laws hold; the mean free path of the particles is small compared to the distances over which the pressure appreciably changes; dimensions of an electrical circuit are small compared to the wavelength associated with the highest frequency; etc.

6.2. QSIM and CA

Qualitative simulation can also be defined as the derivation of a description of the behavior of a mechanism from a qualitative description of its structure. Kuipers thinks that causality can be taken as identical to value propagation with constraints [15]. Hence, Kuipers has a constraint-centered ontology [14]. (On the other hand, de Kleer and Brown [13], and Forbus [6] have device- and process-centered ontologies, respectively.) Kuipers' qualitative simulation framework is a symbolic system which solves a set of constraints obtained from differential equations. His QSIM algorithm is guaranteed to produce a qualitative behavior corresponding to any solution to the original equation. He also shows that in some cases a qualitative description of structure is consistent with intractably many behavioral predictions. A couple of techniques representing different trade-offs between generality and power have been proposed for

"taming" this intractable branching [16].

As an example of how QSIM can be used consider the problem of throwing an object vertically into air from some height. Using a problem description language this is first described to QSIM. The description includes, in addition to the initial values of the problem variables, the physical constraints (law-like knowledge) acting on the ball: $\frac{ds}{dt} = v$, $\frac{dv}{dt} = a$, and $a = g < 0$ where s, v, and a stand for displacement, velocity, and acceleration, respectively. These are entered to the system by the user. (In the future, it may be possible to extract the relevant portion of a "physics knowledge base" to use it in a problem.[5]) From this initial information, QSIM reasons in an intuitive way and finds that the ball will rise to a particular height, stop, and then fall to the earth again. The *landmark* point where the velocity becomes zero is discovered by QSIM. Landmark values are important in identifying the different regions of behavior for a mechanism.

In our view, the main problem with QSIM lies in its essence. Obviously QSIM has no knowledge of physics whatsoever. In the above example, QSIM cannot possibly know that the object may leave the earth to orbit around it or even go to the infinity if it is given enough speed initially (cf. §5.1). This is because QSIM assumes that for all cases a is constant — it does not know that the value g for a is applicable only near the surface of the earth and that a decreases inversely with the square of distance. Thus, by submitting a problem to QSIM it is assumed that all relevant knowledge is given; any physics law left unspecified cannot be used.

An interesting extension of qualitative simulation is Weld's "comparative analysis" [25] which was implemented in a program called CA. CA deals with the problem of predicting how a system will react to perturbations in its parameters and why. For example, comparative analysis explains why a ball would go up higher if it is thrown with a greater speed. Weld also studies *exaggeration*. Consider a question like "What happens to the period of oscillation of a block attached to a spring on a frictionless table as the mass of the block is increased?" Exaggeration suggests that if the mass were infinite, then the block would hardly move and thus the period would be infinite. Therefore, had the mass increased a little the period would increase as well. (For another example use of exaggeration, cf. §5.1.)

7. Summary

Contrary to their public image, it is admitted that expert systems are not true experts in their fields [1]. This contributed to the emergence of dichotomies such as "deep vs. shallow knowledge" in AI. While there are problems with using these adjectives, it is understandable that an expert system is not a "deep" model of its domain of expertise. An expert system's *if-then* rules cannot capture but the superficial characteristics of a domain. On the other hand, a real expert has profound thoughts in his domain of expertise. Several criteria have been suggested for true expertise:

- Experts can explain their way of reasoning in a logical way. They do this in a way

[5] An important issue is then the composition of the mechanism under consideration: a system's behavior should be deducible from its structure (e.g. components and their connections), as mentioned in the case of ENVISION. Analogs of this principle are used in various domains. In linguistics, it is assumed that the semantic value of any expression is a function of the semantic values of its syntactic constituents. Thus, the semantic rules will compute the semantic values of increasingly longer parts of a statements. Admittedly, this is sometimes a dangerously mechanistic view.

markedly different from the current expert systems. When asked about how a certain result was obtained, an expert gives information related both to the real world and its abstract models — not just a list of the rules used.

- Experts use multiple models of a domain to classify and solve problems in an efficient manner. They solve difficult problems using difficult methods. However, for easy problems they either reduce the complicated methods to simpler versions or use already available simple methods. Furthermore, experts can predict. Before they delve into a problem they have some general idea about how the solution should look like.
- Experts can discover the inconsistencies in ill-defined problems. They can judge and eliminate irrelevant or contradictory information. Presented with incomplete problem statements, they make reasonable (default) assumptions. As a result, they can work in a ''non-monotonic'' mode, occasionally revising their beliefs during the problem-solving process.

In this paper we argued that an expert theory of envisioning should mainly be based on physics knowledge. This theory satisfies the above criteria. The fictitious entities of physics such as energy, work, force, etc. make up the things that the envisioner reasons about. Abstract principles such as the principle of superposition and laws such as Newton's laws are used as the essential tools for reasoning. Our work is in the precise spirit of Larkin's study [17] in expert-novice difference in problem-solving performance and is based on her distinction between naive vs. physical representations.

Many problems remain to be solved. Central among them is the problem of choosing a concrete knowledge representation method. We think that frames are appropriate for this purpose [22]. Another problem is to define a core subset of physics which can be used effectively for envisioning in *different* areas. Until now, we have mainly studied the domain of mechanics, following the AI tradition. Yet another project is to study the limits of reasoning *without* recourse to law-like knowledge. To invert a remark of Larkin [17], Why are people good at predicting the outcome of physical interactions in the world around them, while they are so bad at physics, even the branch of physics (mechanics) that deals with interaction of everyday objects? Are there other, fundamentally different ways of looking at the physical world? For example, John McCarthy writes that a commonsense knowledge ''database would contain what a robot would need to know about the effects of moving objects around, what a person can be expected to know about his family, and the facts about buying and selling'' in addition to other information [20]. Here the problem lies in integrating these diverse (and obviously, not all physical) domains of knowledge using a base language — a problem McCarthy calls *generality in AI*. He then adds: ''This does not depend on whether the knowledge is expressed in a logical language or in some other formalism.'' We agree but cannot help to point out that this brings us to the dangerous waters of sense experience, learning, causality, etc. [19]:

> In the notice that our senses take of the constant vicissitude of things, we cannot but observe that several particular, both qualities and substances, begin to exist, and that they receive their existence from the due application and operation of some other being. From this observation we get our *ideas* of *cause* and *effect*. That which produces any simple or complex idea we denote by the general name, cause, and that which is produced, effect. Thus, finding that, in that substance which we call wax, fluidity, which is a simple idea that was not in it before, is constantly produced by the application of a certain degree of heat, we call the simple idea of heat, in relation to fluidity in vax, the cause of it, and fluidity the effect.

ACKNOWLEDGMENTS

This paper owes a great deal to [5] as it is evident from the amount of material cited. The interpretations given here, however, are our own and should not be taken as representing the views of the authors of [5]. The first author wishes to thank Tetsuo Tomiyama (University of Tokyo) for his encouragement and invaluable advice.

References

1. Daniel Bobrow, Sanjay Mittal, and Mark Stefik, "Expert systems: Perils and promise," *Communications of the ACM* **29**(9), pp. 880-894 (1986).
2. Alan Bundy, "Will it reach the top? Prediction in the mechanics world," *Artificial Intelligence* **10**, pp. 129-146 (1978).
3. Ernest Davis, "A logical framework for solid object physics," Technical Report No. 245, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York (1986).
4. Andrea A. diSessa, "Unlearning Aristotelian physics: A study of knowledge-based learning," *Cognitive Science* **6**, pp. 37-75 (1982).
5. Richard P. Feynman, Robert B. Leighton, and Matthew Sands, *The Feynman Lectures on Physics, Volume I (Mainly Mechanics, Radiation, and Heat),* Second Printing, Addison-Wesley, Reading, Mass. (1966).
6. Kenneth D. Forbus, "Qualitative process theory," *Artificial Intelligence* **24**, pp. 85-168 (1984).
7. Kenneth D. Forbus and Dedre Gentner, "Learning physical domains: Toward a theoretical framework," Technical Report No. UIUCDCS-R-86-1247 (also UILU-ENG-86-1774), Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Ill. (1986).
8. J.P. den Hartog, *Mechanics,* Dover, New York (1961).
9. Patrick J. Hayes, "The second naive physics manifesto," pp. 467-485 in *Readings in Knowledge Representation*, ed. R.J. Brachman and H.J. Levesque, Morgan Kaufmann, Los Altos, Calif. (1985).
10. Immanuel Kant, *Critique of Pure Reason,* Everyman's Library, London (1984).
11. Johan de Kleer, "Qualitative and quantitative knowledge in classical mechanics," Technical Report No. AI-TR-352, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass. (1975).
12. Johan de Kleer, "Multiple representations of knowledge in a mechanics problem-solver," pp. 299-304 in *Proceedings of the Fifth International Joint Conference on AI (IJCAI-77)*, ed. R. Reddy, Massachusetts Institute of Technology, Cambridge, Mass. (1977).
13. Johan de Kleer and John Seely Brown, "A qualitative physics based on confluences," *Artificial Intelligence* **24**, pp. 7-83 (1984).
14. Benjamin Kuipers, "Commonsense reasoning about causality: Deriving behavior from structure," *Artificial Intelligence* **24**, pp. 169-203 (1984).
15. Benjamin Kuipers, "Qualitative simulation," *Artificial Intelligence* **29**, pp. 289-338 (1986).

16. Benjamin Kuipers and Charles Chiu, ''Taming intractable branching in qualitative simulation,'' pp. 1079-1085 in *Proceedings of the Tenth International Joint Conference on AI (IJCAI-87)*, ed. J. McDermott, Milan, Italy (1987).

17. Jill H. Larkin, ''The role of problem representation in physics,'' pp. 75-97 in *Mental Models*, ed. D. Gentner and A.L. Stevens, Erlbaum, Hillsdale, N.J. (1983).

18. Gottfried Wilhelm Leibniz, *Philosophical Writings,* Everyman's Library, London (1984).

19. John Locke, *An Essay Concerning Human Understanding,* Abridged Edition, Everyman's Library, London (1985).

20. John McCarthy, ''Generality in artificial intelligence,'' *Communications of the ACM* **30**(12), pp. 1030-1035 (1983).

21. Michael McCloskey, ''Naive theories of motion,'' pp. 299-323 in *Mental Models*, ed. D. Gelentner and A. Stevens, Erlbaum, Hillsdale, N.J. (1983).

22. Marvin Minsky, ''A framework for representing knowledge,'' pp. 211-277 in *The Psychology of Computer Vision*, ed. P.H. Winston, McGraw-Hill, New York (1975).

23. James G. Schmolze, ''Physics for robots,'' Doctoral Dissertation, Technical Report No. 6222, BBN Laboratories Inc., Cambridge, Mass. (1987).

24. Yoav Shoham, ''Reasoning about change: Time and causation from the standpoint of artificial intelligence,'' Doctoral Dissertation, Technical Report No. YALEU/CSD/RR#507, Department of Computer Science, Yale University, New Haven, Conn. (1986).

25. Daniel S. Weld, ''Comparative analysis,'' Technical Report No. AIM-951, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass. (1987).

26. Aleksandr A. Zinoviev, *Logical Physics,* trans. from the Russian by O.A. Germogenova, ed. R.S. Cohen, Reidel, Dordrecht, Holland (1983).

Implementation of Technical Rules in a Feature Based Modeller

F.–L. Krause
F.H. Vosgerau
N. Yaramanoglu

Implementation of Technical Rules in a Feature Based Modeller

Frank-Lothar Krause
Fritz H. Vosgerau
Nezih Yaramanoglu

Institute for Machine Tools
and Manufacturing Technology
Technical University Berlin

Pascalstrasse 8-9, FRG 1000 Berlin 10

Telephone (30) 39006 244
Telex 185284 ipkb-d
Fax (30) 39006 299

ABSTRACT

The many years of utilizing CAD systems in industry have confirmed that the design process occurs on a more intellectual level than is permitted by the current systems with their primitive geometrical elements. The transformation of design rules and solutions into primitive geometrical elements is an activity which in the current state of CAD technology must be performed only by the designer. The transfer of the necessary knowledge to the computer can allow the designer at least to be partially relieved from this task. For this purpose a combined computer-internal handling of functional and technological information and geometrical elements is required. The paper presents a new system concept for handling features. Form features and technological and functional features are differentiated. A new kind of entity is introduced to integrate a rule base into the product model, in order to provide an intelligent support of the design process using the feature and form feature elements. In addition to the representation of a principle data scheme for the design of features, technical problems concerning structure are discussed. The realizations presented are based on an analytical boundary representation. Furthermore, the requirements for the technical realization concerning software and the informational contents of feature and form features are discussed. In the structural conceptualization a high flexibility through the feature oriented variability of the form features is emphasized. The algorithms are realized in the form of methods which are managed in a method and model base. The paper closes with the future-oriented requirements of system architecture and man-machine communication technologies.

1 Introduction

Many years of CAD-application in industry have shown that the design process requires a higher intellectual level than is permitted by the existing systems with their primitive geometric elements. Structuring the thinking of the design engineer into lines, arcs and solid primitives would be equivalent to building a text form more letters instead of words and sentences.

Group oriented modelling functions and their advantage are known today especially from state-of-the-art 2D CAD-systems. An example of this is the generation of parts libraries in order to reduce the input times for repetitive modelling activities. It is one of the aims of research in the field of computer aided design at the Institute for Maschine Tools and Manufacturing Technology to integrate expert knowledge into the process of selection and application of design primitives /1/. Only the completion of expert knowledge with user-specific information will make it possible to provide the design engineer with the kind of support that he needs to carry out his tasks adequately. Depending on the kind of production and the range of products there are differences as to the demands on the performance of modelling and model handling functionality /2/.

If one wishes to achieve functional and manufactural optimization of product components at an early stage by means of integration, it is necessary to consider the post design phases of production in the development of a CAD-system. The ability of CAD-systems to take technological factors in the design phase into account results from the coupling of geometry with the technical data and rules /3/. These regularities of the interplay between technology and geometry are valid for partial structures of a component, which as a rule are represented by several elements of a geometric model. Thus the integration of a modeller into the production process can only be realized in an economic and flexible way by means of a logically connected handling of these geometric elements. For this purpose, a few basic functions for the handling of form features have been realized in the course of the further development of a high-performance modelling environment for designing mechanical systems. Form features accur as groups of geometric elements of varying degrees of complexity. A task of form features is to make the generation of a geometric model possible, which consists of coherent groups of geometric elements. These form features can be created according to different aspects such as functionality, calculation, process planning, NC-machining and assembling. The well-aimed assignment of the geometric parts to technological characteristics makes it possible to analyze the geometry to be worked on in an adequate way /4,5,6,7/. Thus, additions and manipulations of component geometries can be oriented to the technology. Furthermore, the created geometric models obtain a degree of variability which makes a coherent manipulation of the geometry possible.

The following report presents development ideas from the point of view of a geometric modeller in order to raise computer aided design to a higher, more adequate level. Apparently, the solution to this problem does not only consist in an enlargement of the modelling functionality, but also in the increase of the application flexibility of the methods of modelling. This increase is necessary in order to connect the modelling methods into a design system by means of knowledge based software components.

2 Form Features as Flexible Model Components

2.1 Model Handling Procedures

The creation of the association between geometry information and other product definition data such as functionality, design conditions and manufacturing process requires flexible model handling methods. Conventional kinds of computer internal object representations and model handling methods can meet these requirements only to a limited degree or, in some cases, not at all. This leads to the necessity of integrating all product definition data in one informational model called product model. At present, CAD-systems manage product definition data in various forms. However, in most cases they do not permit relations between geometry data and production oriented information. Product models created in such a way merely constitute an attributive addition to the geometric model. Previous implementations have shown that an active computer support is only possible by the topological storing of regularities through the application of form features.

It seems that the creation of form features according to different criteria is a necessary model handling mechanism, if a CAD-system is supposed to support its user in following a superordinate design and manufacturing logic. The application of variable form features in the 2D- as well as the 3D-area and the classification of functional and technological characteristics make it possible to realize a problem-oriented, coherent modelling methodology. Two possible data structures present themselves for the management of the form features within solid modelers. Apart from the analytical Boundary Representation (B-Rep), solid representation data can be procedurally stored in the form of the Constructive Solid Geometry (CSG). This report is based on the CAD-system COMPAC which supports an Analytical Boundary Representation /8/. Thus, the following concept is applicable to model representation types of this kind. Within the COMPAC-system this concept was realized by the introduction of new types of entities such as features and form features.

As a rule, form features are a collection of geometry elements, which constitute parts of objects or define a complete body. The assembly of a complete part or assembly respectively determines the features of the target object, which results from the sum of the properties of such form features. Since features can concern geometries of varying degrees of complexity, the definition of form features requires a logical structuring according to the degree of complexity of the underlying geometries. Variable form features can be described in two ways: on the one hand procedurally, or on the other hand by assigning parameters of geometries generated by means of generative methods. Since both procedures bring their own advantage, a problem-oriented combination of both recommends itself. The handling of entities like form features in an explicit manner can be preformed in a B-rep invironment without any basic changes in the system structure. As shown in figure 1 the COMPAC B-rep had to underlie an additive change in the data structure to be able to handle the new entities: rules, features and form features. Figure 2 shows the varying degrees of complexity of explicit form features. They are logically

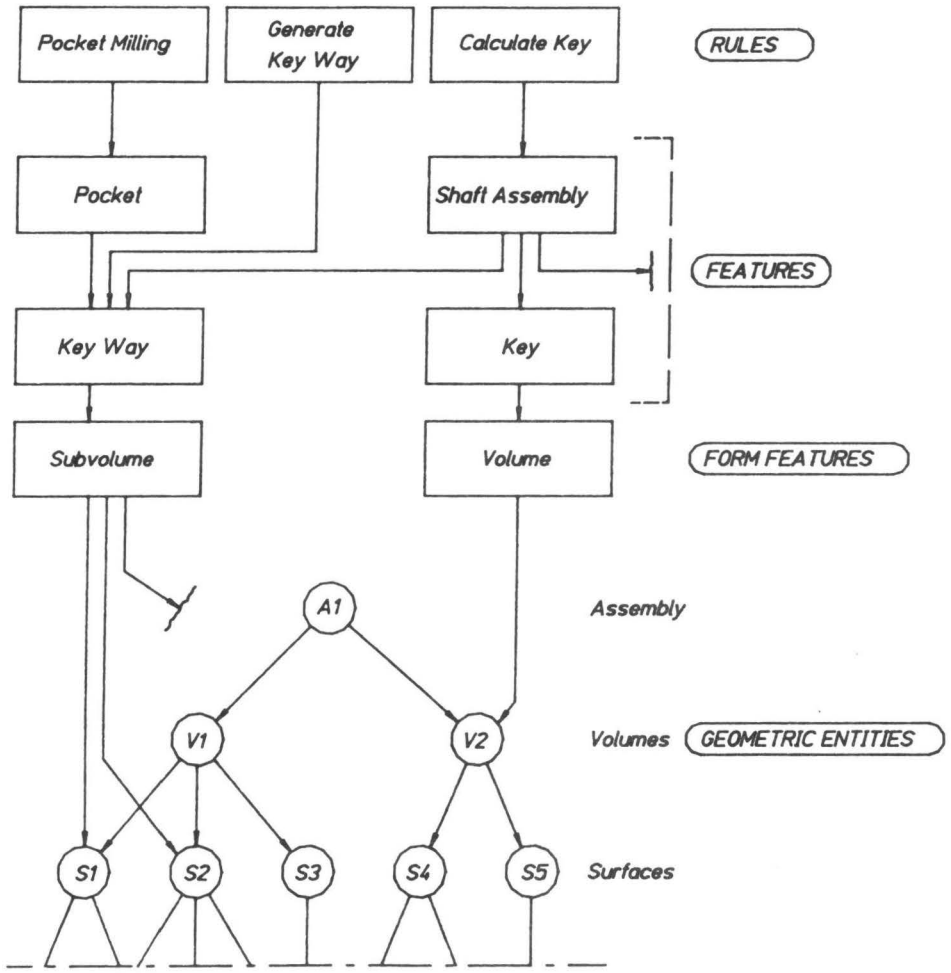


Fig. 1 Expansion of COMPAC data structure for feature handling purposes

classified into three main steps which are named after their basic elements such as surface, subsolid, and solid:

- surface,
- group of surfaces,
- subsolid,
- group of subsolids,
- solid,
- assemblies.

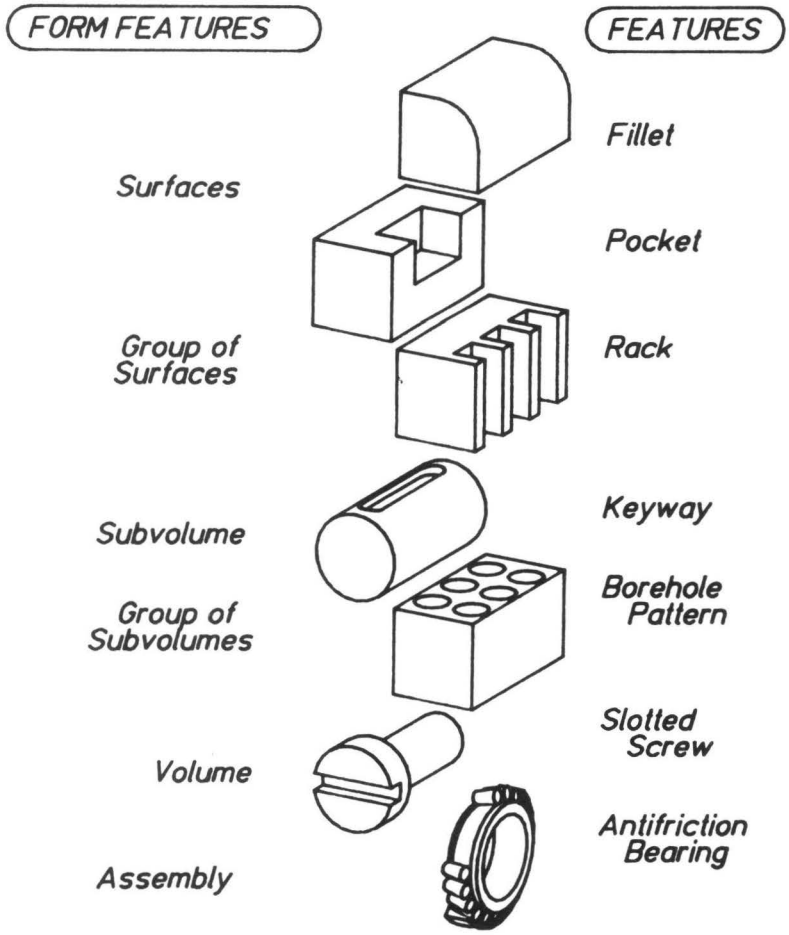


Fig. 2 Classification of form features

The differences between several surfaces and subsolids concern the way of their topological relations towards their neighbouring geometries. From a structural point of view subsolids combine surfaces which completely describe partial solids in a positive or negative way. Neither deleting nor adding such elements afflicts the topological connection of the modified model. However, the manipulation of groups of surfaces requires an additional treatment of the neighbouring geometries. These surface elements are difficult to manage in comparison to higher elements. In principal edges and vertices are also applicable in form features.

The form features of a higher complexity like solids or assemblies can also be handled as implicit entities. In the COMPAC data structure this could be achieved by expanding the form feature entities with additional generative and manipulative methods and defining relations to the rule entities. The rule entities can contain design or constructive rules, which represent the description to generate the form features out of a set of geometric data. Such implicit form features can be manipulated and combined using geometric set operations. An application of this method is shown in figure 3. In the following passage the realization in the existing data structure is presented.

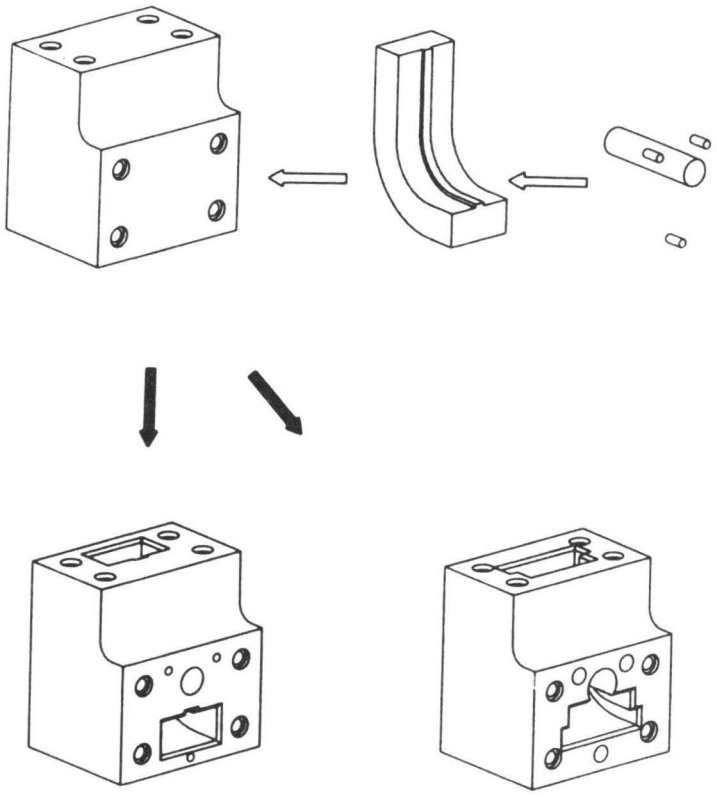


Fig. 3 Implicit modeling using variable form features/9/

2.2 Data Structures

Boundary Geometry describes objects through bordering geometry of lower dimensions such as surfaces, edges and vertices in space. The given combinatory-topological relations between the geometric figures are realized in the representation structure by means of generating relations between corresponding elements. Apart from barely geometric information, attributes consisting of data and text can be mapped in this hierarchically structured representation structure and related to the geometry. These attributes make it possible to manage information concerning the properties of the object to be modelled in addition to the purely geometric information.

For a higher flexibility during the product modelling activity it can become necessary to process the geometry-in-dependent properties of the product. Since it is to be expected that the feature description of a product also possesses a structure, data base functions suggest themselves as a basic tool. Today's data base systems process branching functions which make the definition of relations between data sets possible.

The realization environment with the solid modeller COMPAC meets these demands due to its standardized data handling interface. A new model component outside the geometry was defined in order to use the model handling functions intended for the geometry also for the purpose of

defining relations between data strings. As figure 1 shows, these components consist of feature and form feature entities which constitute the integrative parts of the product model in a network structure in relation with the geometry. Apart from this function features have the task to connect data with the algorithms to be processed. These algorithms can have generative, testing or modifying functions that are both of general validity and specific to form features. Figure 4 shows the application of such a structure to a shaft assembly with a sunk key.

The structure shows an M:N-relation between feature and form feature entities. Thus, one form feature can represent several features. This structure has the advantage that selection algorithms in both directions can be realized. Depending on the activity of the designer an upward or downward search in the structure can be necessary. Depending on the geometry the functional geometry synthesis proceeds in the sequence feature - form feature - geometry; a production-oriented geometry analysis requires an upward search in the opposite direction.

A feature entity consists of a name, a variable set of attributes and the related values. Attributes are represented in textual form, value in the types of "character", "integer" or "real". Thus, not only algorithms with fixed syntax can process this information, but especially the future rules which question, evaluate and manipulate the attributes in a flexible manner. Depending on the case of application the features possess hierarchical or linear structures. The case of problem analysis requires a flexible hierarchical structure between the features which does not necessarily have to be geometry-related on the higher levels. For the case of geometry processing based on features a linear feature structure suffices.

2.3 Concept of a Rule Base

The tasks of design consist of numerous selection processes. Due to certain conditions, each of these selection processes leads to decisions which in turn are expected to lead to correct solutions. A test version of a support mechanism during the design phase was realized in FORTRAN. For this purpose the search strategies for the geometric solutions of functional tasks are implemented based on the evaluation of efficiency factors. The selection algorithms show a structure very similar to that of rule based systems; it consists of condition and action part. Experience with the FORTRAN-environment, however has shown that the expected flexibility in the use of rules cannot be achieved in an economic way by means of traditional algorithmic procedures. The methods of the action part, on the other hand, are of a traditional kind, so that the translation of existing FORTRAN algorithms into another language does not seem to make sence for economic reasons.

For a problem-oriented feature analysis of COMPAC models, the COMPAC data structure was realized in LISP. The analyzing LISP algorithms find borehole patterns (Figure 5) for the generation of manufacturing documents from the topological relations in the model. Fast prototyping and a safe process behaviour proved to be advantageous for this solution. The necessity of handling of two parallel model representations simutaneously

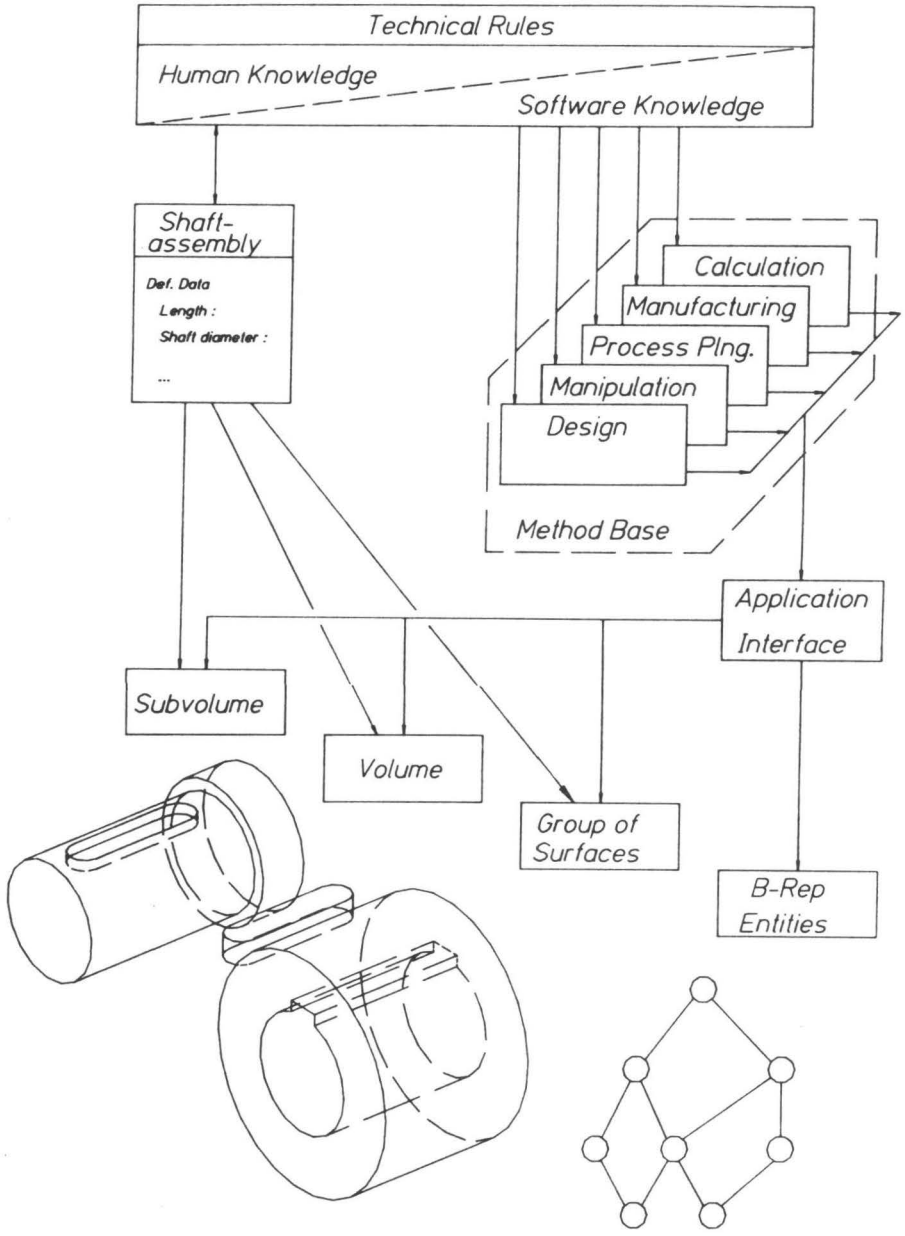


Fig. 4 Shaft assembly mapped into the data stucture

resulted in additional management expenditure. Moreover, the coupling of the languages LISP and FORTRAN proved to be problematic in the existing VAX environment. In order to avoid the development of an inference mechanism and of search strategies, the realization of the selection procedure is

carried out using the tool OPS5. This tool is supposed to serve for the creation of a rule base for the design of mechanical parts and assemblies. Presently, the realization is in the test phase.

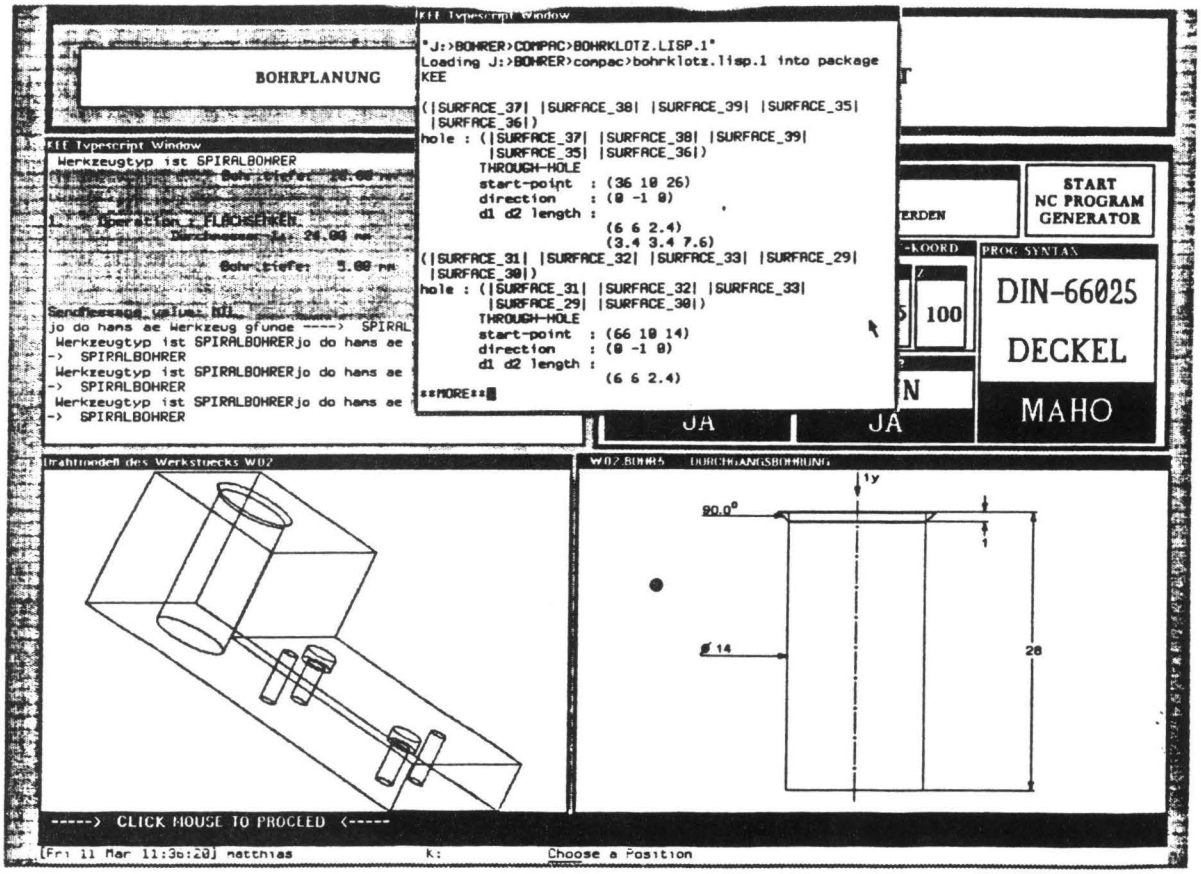


Fig. 5 Rule based feature analysis for manufacturing

The syntax of OPS5 experimental language makes it easy to distinquish clearly between actions and rule base, so that action algorithms can be implemented in the method base environment in any programming language without influencing the algorithms in the rule base. For the left side of the rules algorithms have been realized which take over the processing of feature oriented attributes. Thus it is possible to compare desired values with the values from the attributes in the left side of the rule and to draw conclusions influencing the geometry. Thus the prerequisites for connecting the variable contents of the rule with the modelling functionality have been created. An OPS5 implementation of the decision making during the design of a shaft assembly based on a design-cathalogue /3/ is shown in figure 6.

Gliederungsteil		Hauptteil				Zugriffsteil														Anhang		
Art des Flächenschlusses	Art der Kraftübertragung	Gleichung	Benennung	Anordnungsbeispiel	Nr	Übertragbares Moment	Momentübertragung abhängig von	Aufnahme von Axialkräften	Verwendbarkeit bei	Wirkung bei Überlastung	Verbindung zentrierbar	Unwucht	Nabe axial verschiebbar	Nabe versetzbar	Verbindung nachstellbar	Wellendurchmesser [mm]	Werkstoff	Herstellungsaufwand	Montageaufwand	DIN Quelle (Hersteller)	Anwendungsbeispiele	Anmerkungen
1	2	1	2	3		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Normal	kunmittelbar	$M_t \leq K \frac{d_o}{2} A_T \tau_{zul}$ bzw $M_t \leq K \frac{d_o}{2} A_p p_{zul}$	Keilwelle		1	groß	h, t, l	—	Stoßen Wechsellast	Bruch	ja	nein	bei Spielpassung	ja	nein	10-150	Welle 37Cr4 41Cr4 42CrMo4	hoch	klein	5461/63 5471/72	Zahnräder	Außen-Flanken-Innenzentrierung möglich kurze Nabe möglich
			Evolventenzahnwelle		2			—												5480 5482		
			Kerbzahnwelle		3			—								150-500				5481		
			P3-Polygon		4		e, l, i	—	Stoßen Wechsellast		selbstzentrierend	nein	bei Spielpassung und ohne Last		ja bei Kegel	10-100		klein aber Spezialmasch		—		geeignet für kurze u dünne Naben, Kegeliges Wellenende möglich Profil räumen oder schleifen notwendig
			PC4-Polygon		5			—								10-100				—		
	mittelbar		Querstift		6	klein	d_{st}, D	ja	—				—		ja, bei Kegelstift	0.5-50	Stift 40.S5, 65.86, 95.20K St 50K		mittel	1.7 1470-77 1481, 6324 7346	Hebelbetätigung	Kegel- und Kerbstift möglich
			Tangentstift		7			—	—				—									
			Langsstift		8		d_{st}, l	—	—													
			Paßfeder		9		h, l, b	—	—													
			Scheibenfeder		10			—	—													
			Tangentenkeil		11	mittel		—	Stoßen Wechsellast													

Wellendurchmesser, Form-, Zentrierfaktor, Werkstoffwahl

```
;== Wissensbasis ==========================================================

(P Keilwelle
   ( Durchmesser Wert     (* >= 10.00 <= 150.00 *) )
   ( Moment Lage          << mittel gross >> )
   ( Herstellung Aufwand = hoch )
   ( Montage Aufwand      << mittel klein >> )
 -->
   ( MAKE Loesung Text Keilwelle )
)

(P Passfeder
   ( Durchmesser Wert     (* >= 5.00 <= 500.00 *) )
   ( Moment Lage          = klein )
   ( Herstellung Aufwand << hoch mittel >> )
   ( Montage Aufwand      << mittel klein >> )
 -->
   ( MAKE Loesung Text Passfeder )
)

(P ENDE
   (* <LOS> ( Loesung Text <L> ) *)
 -->
   ( WRITE (CRLF) (TABTO 5)
     |Loesung : | <L> )
   ( CALL PUTPAR ( SUBSTR <LOS> 1 INF ) )
)

; ==========================================================================
```

Fig. 6 Design cathalogue/3/ and its implementation in OPS5

2.4 Basic Software and User Interface

By means of a suitable system concept the designer is to be provided with a software tool that comprises functions of different CAD-systems and makes it possible for him to compile his special user system for his specific tasks. A method and model base system has been designed as a first step towards the realization of such a tool.

As a basis the system contains a method base with a large number of task oriented application methods which can also be called functional components. They can be combined and executed by the user through the system. The system contains management functions which inform the user about the available functional components and which to a large extent automate the search for components satisfying a given functionality. In addition there exists the possibility of testing individual methods and of training the user in running the methods.

The concept provides for the application of numerous types of geometric models and product models. For the model bank models of different structures are planned. This has consequences on the execution of functional components and the possibilities of their combination, because the tolerance and convertibility of model types must be tested.

Besides a high flexibility of the modelling functions the design engineer puts high claims on the user interface. Thus, the user interface is understood to be the main tool for his communication with the system while generating a part modell. Such an interface must provide functionalities to make the definition of the technical form features and the modelling using technical features and functional discriptions possible. As a tool to develop a technical modelling language a dialog specification language DIABES has been realized. This tool enables any user to describe his own individual communication interface with a networklike structure.

Supporting the user with adequate visualizing methods is considered to be one of the basic functions of a geometric modelling system. Such functions can also be used to increase the efficiency while working with form features. Figure 7 shows an application of color coding as a verification aid.

3 Applications in the Design Process

Geometric modelling combines activities of various design phases, which are well known from the conventional design procedure. Design can be characterized as an activity of developing and optimizing forms and functions in mechanical engineering. During this activity the information represention is primarily graphical based on elements. These representations show different degrees of abstraction depending on the development phase of the product. Thus it is a natural need of the designer to communicate with his partner in an adequate graphic language, even if this partner is a computer. In the concept phase he expresses his functional requirements on the product in the form of symbols which constitute the effectual elements of a mechanical system. The actual design restrictions of the form result from such an abstract representational model of the system. Only after a rough shape of the product has been created the design engineer executes a dimensioning and detailling activity on his product. An adequate computer support, which must be able to process varying degrees of geometric details according to the design phase, can only be realized efficiently by means of the application of several model components of differing degrees of abstraction.

The functional analysis of the design task, the finding of the function and the selection of effectual representatives is carried out in an integrated way by means of functional networks and feature-oriented model handling. The data environment of an exemplary realization of a design system for mechanical components has two characteristics: a reference structure in form of a network containing the semantics of functional relations and the extended variable geometric models by functional features. In the case of the design of a tailstock the functional structuring of this component is represented in the reference network.

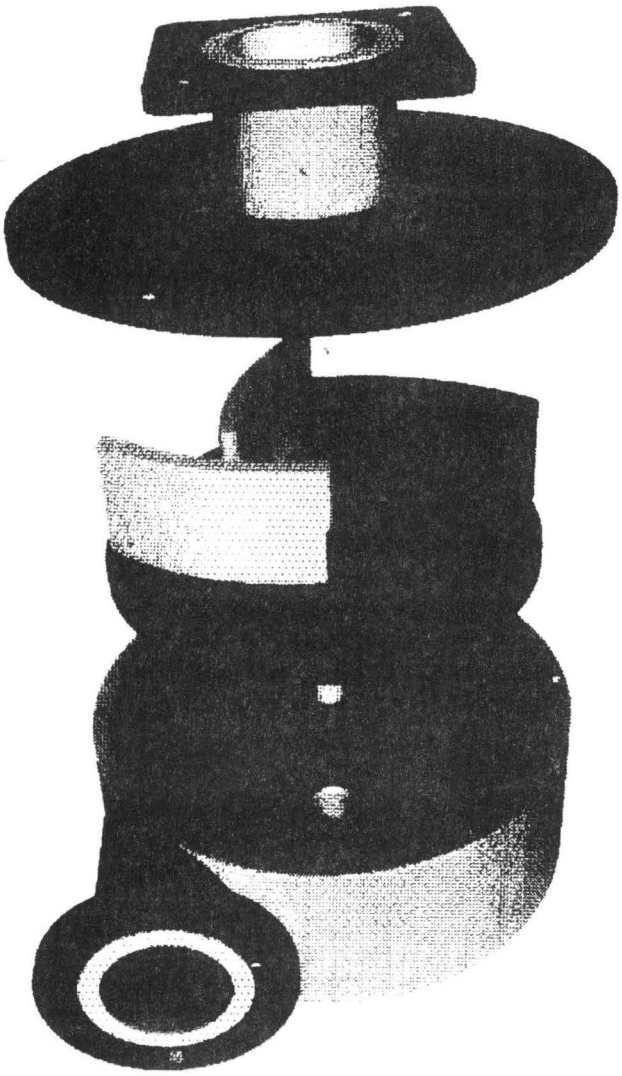


Fig. 7 Color coded visualisation of form features after a feature analysis

Figure 8 shows the hierarchy of a reference structure and the projection of the functional features onto the geometric model for the example of the tailstock barrel. The functional structuring of a mechanical task and the determination of alternative geometric solutions is made possible with the help of special searching procedures. The elimination of the alternatives during searching is realized by the comparison of the evaluation factors and the predefined efficiency profile. The elimination mechanisms applied on the tailstock barrel and the results of two different efficiency profiles are represented in figures 9 and 10.

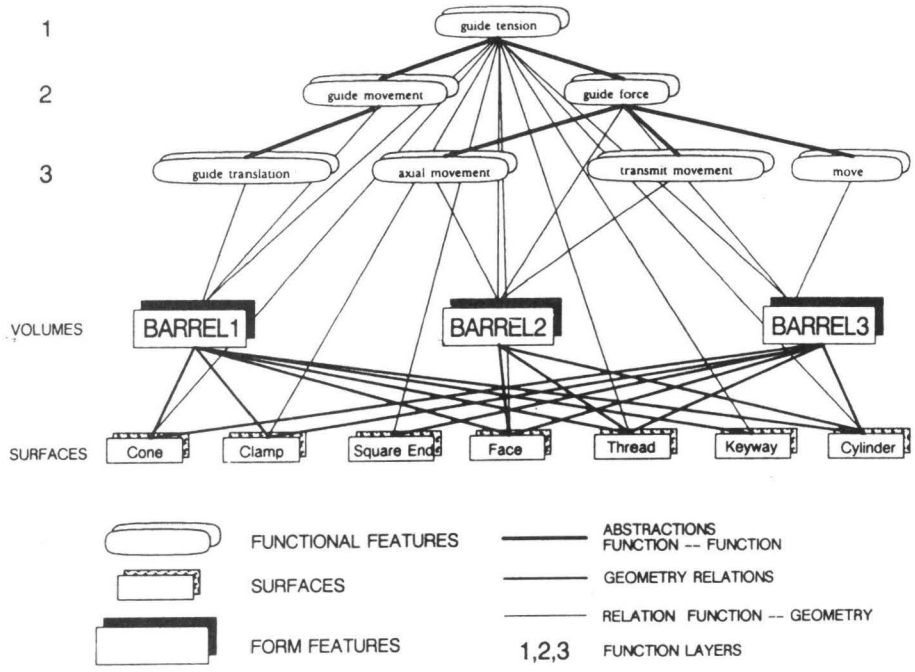


Fig. 8 Design alternatives in a functional network

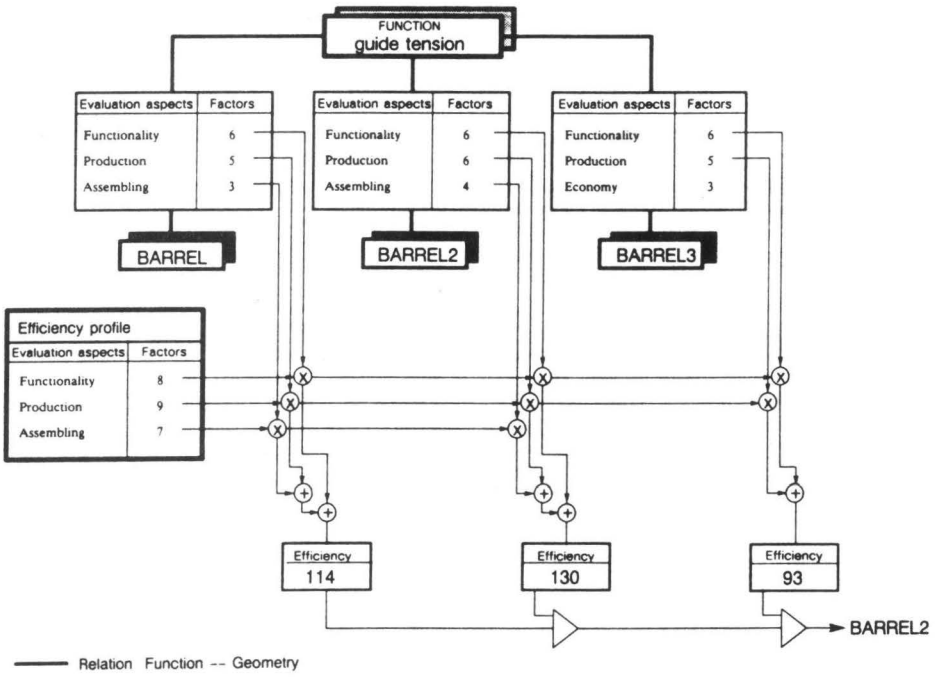


Fig. 9 Elimination of design alternatives

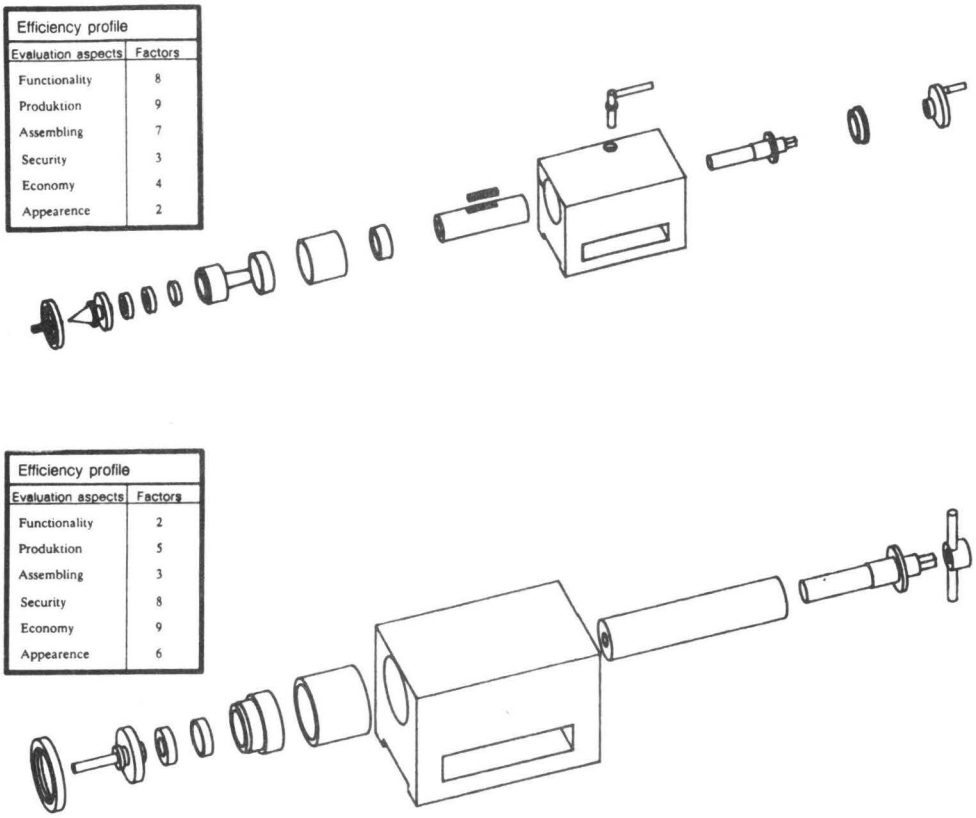


Fig. 10 Two different efficiency profiles resulting in different design solutions

The input possesses possibilities of describing the functional composition of assemblies by means of symbols. An abstract symbolic model is separated into functional components by an interpreter. Before an actual solid model is put together from a parts library, several alternatives are selected and evaluated by rule-based algorithms. The dimensional adaptation of objects is carried out by means of deducing the parameters of the variation rules of the parts applying technical and geometric initial values and conditions.

These variations are carried out by the methods, which are held executable in a methodbase. The methods have access to the feature informations and can manipulate form features and geometric entities using the functions of the modeller. The methods can be activated either by the user interactively or by an integrated rule based algorithm automatically. Figure 4 shows a sample application of the concept on a shaft assembly. The integration of technical rules into the automated decision making based on knowledge based components still is in an experimental phase. For using a rule base to process the design knowledge the OPS5-language seems to be a qualified tool.

4 Summary

There exist various approaches to the development of a CAD-system with intelligent properties. Two main directions of work can be observed:

- research into the design process with regard to developing algorithms for design systematics;

- making the modelling functions more flexible in order to integrate a superordinate logic such as the functionality and technology concerning a product into the modelling activity.

This report is meant to give an impression of the endeavours to make a contribution to the development of CAD-systems from the part of the modeller. Research into the following areas belongs to these endeavours: influences of feature-oriented modelling; problem-oriented geometric analysis; adequate man-maschine-communication; assignment of features of the geometric information to the design process. For this purpose, in the software development an extended informational structure has been realized on the data level which facilitate the application of knowledge-based components. The possibilities for the use of AI-tools have been demonstrated. In the realisation of these development the functionality of the method and model base system developed with the same purpose played a decisive part. Together these developments are expected to constitute an important contribution to the development of CAD-systems which can be classified as "intelligent".

5 Acknowledgement

The development of tools for the design-oriented support of the user was carried during the special research project SFB 203 funded by the German National Science Foundation (DFG) at the Technical University of Berlin. Speaker of the Project: Prof. Dr.-Ing. Drs. h.c. G. Spur.

6 References

/1/ G.Spur and F.L.Krause, CAD-Technik, Carl Hanser Verlag, München, 1984.

/2/ F.L.Krause et.al., Flexibilitätssteigerung und Beschleunigung der Geometrieverarbeitung, in: Tagungsband zum CAD-Kolloquium des Sonderforschungsbereichs 203 der TU-Berlin: Rechnerunterstützte Konstruktionsmodelle im Maschinenwesen, Berlin, 1986, pp.183-214.

/3/ K.Roth, Konstruieren mit Konstruktionskatalogen, Springer, Berlin, 1982.

/4/ M.R.Henderson, Extraction and Organization of Form Features, Proc.Prolamat'85, North Holland Publishing Co.

/5/ S.B.Joshi and T.C.Chang, CAD Interface for Automated Process Planning, in: 19th CIRP International Seminar on Manufacturing Systems "Computer Aided Process Planning", Penn State, 1987.

/6/ B.K.Chai, M.M.Barash and D.C.Anderson, Automatic Recognition of Machined Surfaces from a 3D Solid Model, in: Computer-Aided Design, Vol.16, No.2, 1984, pp.81-86.

/7/ G.Pahl and W.Beitz, Konstruktionslehre - Handbuch für Studium und Praxis, Springer, Berlin, 1986.

/8/ G.Spur, R.Daßler, H.J.Germer and M.Imam, COMPAC - Aspects of Geometry Modelling, Proc. of the 3rd. European Conference on CAD/CAM and Computer Graphics, MICAD'84, Vol.3, Paris, 1984.

/9/ R.Daßler, Variable Geometriemodelle und ausgewählte Anwendungen, Reihe Produktionstechnik Berlin, Carl Hanser Verlag, München, 1985.

/10/ F.Major, W.Grottke, Knowledge Engineering within Integrated Process Planning Systems, International Conference on Intelligent Manufacturing Systems, 16-19 June, Budapest, Hunagy, 1986.

Product and Process Design in Intelligent CAD Workstation

D. Ben–Arieh

Product and Process Design in Intelligent CAD Workstation

by: David Ben-Arieh

Dept. of Industrial Eng.

Ben-Gurion University,

Beer - Sheva, 84105, Israel

Abstract

Today's products are characterized by a short life cycle, more complicated technology, an increasing variety and decreasing costs. The combination of the above factors force industry to accelerate the process of introducing new products. This process from design to manufacturing is termed Product Realization Process (PRP).

This paper describes a new approach towards process representing which is utilized to represent the product realization process. The process representation is based upon Concurrent Prolog as a process description language, and utilizes the FCP dialect (Flat Concurrent Prolog). This representation is the foundation of an intelligent design process of the PRP. It has two major objectives: to enable faster and better design of new products, and to better control the development process of new products.

Product and Process Design in Intelligent CAD Workstation

Today's products are characterized by a short life cycle, more complicated technology, an increasing variety and decreasing costs. In the electronics industry an average life cycle of a semiconductor is estimated to be two years.

The combination of the above factors force industry to accelerate the process of introducing new products. This process from design to manufacturing is termed Product Realization Process (PRP).

The product realization process is iterative: frequent changes in product and manufacturing technology combined with design uncertainty make iteration a must [Ben86].

In order to shorten this iterative process and to have a longer useful product life cycle, the design process needs to encompass more knowledge about the other activities in the PRP, especially in the manufacturing stage. This will not eliminate the need for iterations in the process, but will minimize the magnitude of this phenomenon and its influence on the activities that are driven by the design activity. Especially it will decrease the friction between the production phase of the product life cycle, and its design phase.

This paper describes a new approach towards process representing which is utilized to represent the product realization process. The process representation is based upon Concurrent Prolog as a process description language, and utilizes the FCP dialect (Flat Concurrent Prolog).

This representation is the foundation of an intelligent design process of the PRP. It has two major objectives: to enable faster and better design of new products, and to better control the development process of new products.

Traditionally, Computer Aided Design (CAD) workstations enabled only product design, an activity which is isolated from the other activities in the PRP. The information that the design activity generates is the geometry of the product, parts list and data useful for product analysis. Other users of the design information are hidden from the designers and formally the rest of the PRP is ambiguous at this stage. This new approach extends CAD activities to process design and control: specifically design and control of the design and manufacturing phases of new products.

Representation of Processes

The knowledge about processes is composed of two major perspectives: One is the process flow which describes the various activities and the order of their operation, and the

second is the information flow among the activities. Because of historical reasons each representation scheme was developed individually with defined methodologies and applications.

The information flow representation is based upon structured analysis and data flow diagrams [Dem78] . These methods describe the input and output of the activities but lack the following features:

1. The data items are not detailed enough and are represented only by a name.
2. The relationship among the data items inside the activities is not defined.
3. The order of operation of activities is not clear from the representation.
4. There is no chronological order among the data elements.
5. The functionality of each activity is hidden from the user.

The process flow can be represented using Petri Nets, simulation languages, or parallel processing languages. Petri Nets (as defined for example by [Pet81]) is a clear and well defined methodology but lacks the ability to represent the dynamics of the system. Details such as time of activities, complicated control rules and flow of information are not described adequately.

Simulation languages such as SLAM [Pri86] tries to represent both the process dynamics and the detailed flow among activities. However, the main disadvantage of this approach is the lack of the information perspective in the system, beside flow of physical entities. Additional shortcomings are inability to represent complex synchronization among activities, and to perform static analysis of the process (as is defined later).

The use of concurrent programming languages as tools for process flow representation is a new approach not yet conquered. This type of languages is still under study [WegSmo83] and its main goal is to define concurrent computer processes. Typically these languages are too detailed and burden the user with many external issues such as variable definitions, parameter passing techniques, etc.

The process flow and the information flow perspectives should be tied together since one complement the other in representing a process.

The described hybrid methodology for process representation include the following features:

1. Interactive and user friendly tool for process

2. Description of the information flow in the process, including the transformation function of the input to the output.

3. Hierarchical description of the process.

4. Enabling syntactic analysis of the process in order to define the following properties:
   I. Deadlocks among activities.
   II. Circularity of activities or information flows.
   III. Reachability tree of activities.
   IV. Level of concurrency of the system.
   V. Activities/Data interaction: which activities are influenced by a particular data item, and vice versa.

5. Dynamic analysis and simulation of the process, mainly for performance evaluation of the process.

6. Integration of separate modules into a unified process and confirming its validity.

7. Decomposing a process into independent sub-processes, without violating precedence or other constraints.

Therefore, the design activity will support designing the product and its parts, assigning the activities that are involved in the PRP of that product (manufacturing, as well as purchasing, testing, etc) and designing the information that the activities utilize and its flow.

Flat Concurrent Prolog

This section briefly discusses the properties of FCP (Flat Concurrent Prolog) as to lead to understand its adequacy for process representation. A more complete description can be found in [Sha86] for example.
FCP generally follows the Edinburgh Prolog conventions. A goal, or process is a term of the form p(T1,T2,..Tn), where p is the predicate name, and n >=0 is the predicate arity. FCP uses guarded clauses of the form:

A :- G1, G2,...Gm | B1, B2, ...Bn. n,m >= 0.

In this case A, Gi's and Bi's are goals. A is the clause's head, G's are its guard and the B's its body. The "|" operator is the commit operator, that exists if the guard is not empty.

A flat guarded clause is a clause whose guard predicates are in the set of predefined guard test predicates. An FCP procedure is a list of flat guarded clauses with the same head's name and arity.

FCP has a unique unification mechanism - read only unification. There are two types of variables: write enables

and read-only variables. The read-only operator, denoted as ?, changes a write enabled variable into a read-only one. If X is a write enabled variable then X? is the read-only variable corresponding to X. A read-only variable can only be read from, but not written upon. It receives a value only when its corresponding write enabled variable receives a non-variable value.

The basic active FCP object is a process. A process performs reduction similar to "regular" Prolog. A general clause with multiple goals in the body specifies process forking: all the new goals become active processes concurrently. A unit clause (a clause with empty body and guard) specifies termination. An iterative clause (a clause whose body contains exactly one goal) specifies a state change.

A reduction is suspended (and so is the process) if the unification is suspended, because of a read-only variable waiting for values to be assigned through the write enable corresponding variable. A process is suspended if an instantiated guard is suspended; waiting for the test to become true.

Process Representation using FCP

Terminology

Activity: An actual operation in the modeled system.

Process: An FCP process.

Block: High level operation (of the system) containing one or more activities.

Stream: A communication line for message passing between processes.

Concepts:

The representation of a system follows the next concepts:

- Activities representation: Each activity is represented as an FCP process which has two kinds of parameters; input parameters and output parameters. A process begins when all its input parameters are instantiated, and it lasts as long as there may be more inputs to arrive, or until the process is terminated.

- Communication: The communication among processes is implemented by the use of streams. Streams are incomplete messages, which are actually lists containing a grounded head (car) and an uninstantiated (variable) tail (cdr). This mechanism enables a list of messages to pass between

processes as data processing messages, as well as synchronization signals.

- Hierarchy. The description of the system is done hierarchically by joining activities into blocks, and joining lower level blocks into higher level ones. The reasons for the hierarchical description are twosome: Structuring the modeled process, and localizing internal messages as explained below.

- Data Synchronization. An input parameter is implemented by a read only occurrence of a variable. This utilizes the FCP mechanism in order to model a data driven system.

Description of a Single Activity

Each activity is coded as an FCP procedure with its input and output streams.

For example the activity act1 in Figure 1 is represented as:

```
act1([In|Ins],[act_on(In)|Outs]) :-
                        act1(Ins?, Outs).

act1([],[]).
```

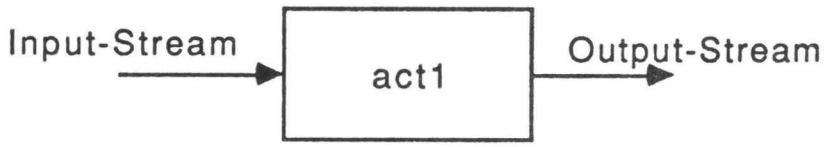


Figure 1: A Single Activity

Relations Among Activities

The relations between activities can be defined by their information flow. Because of the inherent parallelism of the language the control flow of the activities is defined by the synchronization messages or information flow type. The relations can be:

Pipe line. This is the producer-consumer relationship. In this case the output of the first activity serves as the input to the second one. An example is depicted in Figure 2, and its FCP Process is shown below.

```
block(In1,Out2) :- act1(In1?,Out1),act2(Out1?,Out2).
```

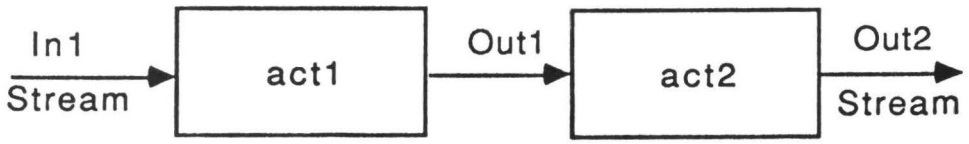


Figure 2: A Pipe Lined Process

It is important to realize that act1 and act2 are concurrent processes synchronized by the message stream Out1.

Sequence. Sequential activities are pipe lined ones, with an additional synchronization variable. For example the same system above is represented as:

```
sequence(In1,Out2) :- act1(In1?,Out1,Sync),
                      act2(Out1?,Out2,Sync?).
```

Note that Out1 is act1 goal.

Iteration. A simple iteration between two activities is represented as a pipe lined process having an additional feedback communication stream. This system has an initial state for creating the first communication stream, and then the system iterates upon the feedback value. Such a system is shown in Figure 3, and is represented as:

```
block(In1,Out2) :- act1(In1?,[begin|Feedback],Out1),
                   act2(Out1?,Feedback,Out2).
```

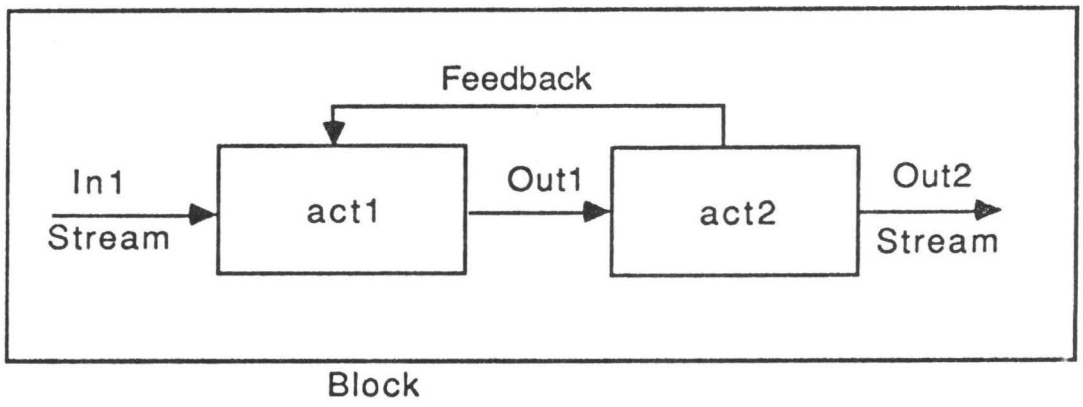


Figure 3: A Simple Iteration

The detailed description of the iteration is available in the Appendix.

It is important to note that iteration is characterized by a number of output messages for each input message. This irregular behavior is difficult to represent, because an activity which gets its input from several iterative activities, may have a different number of messages in each input. The solution to this problem is to structure the system accordingly by creating dummy blocks for each level of iteration. This enables to localize the iterative messages to that block. The output message of the block is the output after the iterative process is complete.

Complicated interactions. More complex structure of activities needs to represent activity which iterates with more than one other activity. In this case the system is modeled using nested blocks, each for an iteration relationship. In this case each block has a fixed input for several values of the output streams inside the block, and a single output of the block.

Blocks. A process in any level of hierarchy is structured into blocks in two cases: When there are more than one output values for each input. In this case the activity iterates within itself. The second case is when the process interacts with another process in an iterative fashion. In the second case both processes form a block.

An Illustrative Example: Design Process in Electronics

This example demonstrates the representation of the design phase of an electronics product, as shown in Figure 4. This phase includes the following activities: Schematic design, simulation of that design, component selection, placement of the components on the board, routing of a board and another simulation and verification of the board. Because of the iterative nature of the process we have three blocks. The first contains the first two activities. The second is a nested block which contains the placement activity connected to the third block which consists of the routing and verifications activities.

The detailed representation is given in the Appendix.

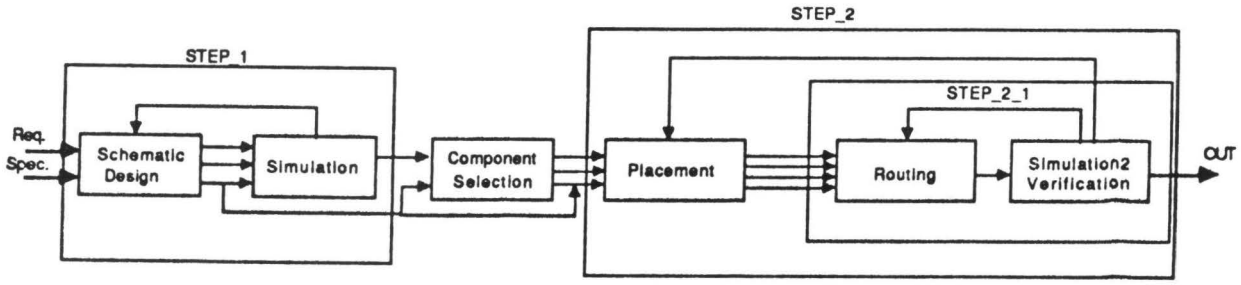


Figure 4: The Design Activities in the PRP

APPENDIX

Description of a Simple Iteration

```
block(In1,Out2) :- act1(In1?,[begin¦Feedback],Out1),
                   act2(Out1?,Feedback,Out2).

act1(In,[begin¦Feedback],[act_on(In¦Out]) :-
                   It_num := 1¦
                   act1(In?,Feedback?,Out,It_num).

act1(In,[F¦Fs],[iteration(In,I_num)¦Out],I-num) :-
                   N_num := I_num + 1,F? = iterate ¦
                act1(In?,Fs?,Out,N_num?).

act1(In,[F],[],_) :- F? = stop ¦ true.

act2([In¦Ins],[F¦Fs],[In¦Out]) :- compute_feedback(In?,F),
                                  act2(Ins?,Fs,Out).

act2([],[],[]).
```

The Detailed Representation of the Design Activities

```
system(in(Req,Spec),Performance):-

step_1(Req?,Spec?,A,B,C,Sync),comp_select(A?,B?,D,E,F,Sync?)
, step_2(B?,C?,D?,E?,F?,Spec?,Performance).

step_1(R,S,A,B,C,Sync):-
    schematic(R?,S?,[begin¦Con],A,B,C),
    sim_1(in(A?,B?,C?),Con,Sync).

schematic(R,S,[begin¦Con],[logic_e¦Chenges],
                [conectivity¦Chenges],[ar_cons¦Chenges]):-
    schematic(R?,S?,Con?,Chenges,0).

schematic(R,S,[Y¦Con],[Cheng¦Chenges],N):-
          Y? = y, NN := N? + 1 ,Cheng = delta(N?) ¦
         schematic(R?,S?,Con?,Chenges,NN?).

schematic(R,S,[C],[],N):- C? = n ¦ true.


sim_1(in([A¦As],[B¦Bs],[C¦Cs]),[Con¦Cons],Sync):-
   screen#ask(sim_1_continue_iterate_on(A?,B?,C?),Con),
   sim_1(in(As?,Bs?,Cs?),Cons,Sync).

sim_1(in([],[],[]),[],finish):- screen#display(sim1).
```

```
comp_select([A¦As],[B¦Bs],[list_of_comp(A,B)¦Ds],

[sugestiv_list_of(A,B)¦Es],[conectivity_of(A,B)¦Fs],Sync):-
Sync? = finish ¦ screen#display(Sync),
comp_select(As?,Bs?,Ds,Es,Fs).

comp_select([A¦As],[B¦Bs],[list_of_comp(A,B)¦Ds],
   [sugestiv_list_of(A,B)¦Es],[conectivity_of(A,B)¦Fs]):-
comp_select(As?,Bs?,Ds,Es,Fs).

comp_select([],[],[],[],[]).

step_2([B¦Bs],[D¦Ds],[E¦Es],[F¦Fs],[C¦Cs],
                    S,[Performance¦Ps]):-
       placement(C?,D?,E?,[begin¦Con],C1,C2,C3,C4),
step_2_1(B?,C1?,C2?,C3?,C4?,E?,D?,F?,S?,Con,Performance),
       step_2(Bs?,Ds?,Es?,Fs?,Cs?,S?,Ps).


step_2([],[],[],[],[],_,[]).


placement(C,D,E,[begin¦Con],[location¦C1],
      [orientation¦C2],[coordination¦C3],
                          [holes_list¦C4]):-
                 placement(C?,D?,E?,Con?,C1,C2,C3,C4,0).

placement(C,D,E,[Y¦Con],[d_place(N)¦C1],
           [d_place(N)¦C2],[d_place(N)¦C3],
                          [d_place(N)¦C4],N):-
     NN := N? + 1,Y? = y ¦
                 placement(C?,D?,E?,Con?,C1,C2,C3,C4,NN?).

placement(_,_,_,[N],[],[],[],[],_):-
                                    N? = n ¦ true.

step_2_1(B,[C1¦C1s],[C2¦C2s],[C3¦C3s],[C4¦C4s],E,D,F,S,
          [Con¦Cons],[Performance¦Prs]):-
      routing(B?,C1?,C2?,C3?,C4?,[begin¦RCon],Performance),
      sim_2(E?,D?,F?,S?,Rout?,Sim_out),
      sim_2_control(Sim_out?,RCon,Con),
      step_2_1(B?,C1s?,C2s?,C3s?,C4s?,E?,D?,F?,S?,Cons,Prs).

step_2_1(_,[],[],[],[],_,_,_,_,[],[]).


routing(B,C1,C2,C3,C4,[begin¦Con],[routing_of(B,C1,C2,C3,C4)
¦Out]):-
     routing(B?,C1?,C2?,C3?,C4?,Con?,Out,0).

routing(B,C1,C2,C3,C4,[Y¦Con],[d_rout(N)¦Out],N):- NN := N?
+ 1 , Y? = y ¦
     routing(B?,C1?,C2?,C3?,C4?,Con?,Out,NN?).
```

```
routing(_,_,_,_,_,[N],[],_):- N? = n ¦ true.


sim_2(E,D,F,S,[Ro¦Ros],[Sim_res¦Rs]):-

screen#ask(sim_2(results_of(E?,D?,F?),are(Ro?),iterate_on
           (placement(n),routing(r),none(n))),
           Sim_res),
      sim_2(E?,D?,F?,S?,Ros?,Rs).

sim_2(_,_,_,_,[],[]).


sim_2_control([R¦Ins],[y¦RCon],Con):- R? = r ¦
   sim_2_control(Ins?,RCon,Con).

sim_2_control([P],[n],y):- P? = p ¦ true.

sim_2_control([N],[n],n):- N? = n ¦ true.
```

References

[Ben86]
Ben-Arieh D., Fritsch C.A. and Mandel K., "Competitive Product Realization in Today's Electronics Industries", Industrial Eng., vol.18, no. 2, 1986.

[Dem78]
Demarco D., "Structured Analysis and System Specification", Yourdon Press, 1978.

[Pet81]
Peterson J.L., "Petri Net Theory and the Modeling of Systems", Prentice-Hall, 1981.

[Pri86]
Pritskr A.A.B., "Introduction to Simulation and SLAM II", Third edition, Halstd Press, 1986.

[Sha86]
Shapiro E., "Concurrent Prolog: A Progress Report", IEEE. Computer, August 1986.

[WegSmo83]
Wegner P., Smolka S.A., "Processes, Tasks and Monitors: A Comparative Study on Concurrent Programming Primitives", IEEE. Trans. on Software Eng., vol. 9, 1983.

Implementing Constraint Propagation in Mechanical CAD Systems

K. ElDahshan
J.P. Barthes

IMPLEMENTING CONSTRAINT PROPAGATION IN MECHANICAL CAD SYSTEMS (draft)

Kamal ELDAHSHAN(1) & Jean Paul BARTHES(1,2)
(1) Université de Technologie de Compiègne
Département de Génie Informatique
BP.649
60206 Compiègne
(2) Société Générale pour les Techniques Nouvelles
1 rue des Hérons
78184 SAINT-QUENTIN-YVELINES
FRANCE

ABSTRACT

This paper presents OPAL, (Object oriented Programming Assembly) a new system for the design and modification of the assembly of mechanical parts based on the constraint propagation mechanism in an object oriented programming environment. Such a system frees the user from the ancillary tasks of tracking consequences of modifications and of verifications of constraints on the computed results. In so doing, graphical aspects are nothing but a secondary task that can be automatized, letting the designer concentrate on the conceptual decisions. Furthermore, the solution is totaly open and can be developed along several axes. We give an idea of the representation we use and of the programming mechanism by developing a small example involving a bolted assembly.

KEY WORDS AND PHRASES

AI, CAD, Intelligent CAD systems, Knowledge representation, Object Oriented Programming, Mechanical Engineering, Assembly, Constraint propagation, Robotics.

0. INTRODUCTION

Current CAD systems are often nothing more than drafting devices. Furthermore, their complexity prevents from integrating easily the needed procedures that could help the user efficiently.

To talk about intelligent CAD systems, we should mention the concept of intelligence as we see it. To automate is to accomplish, with a machine, a task that is usually done by humans. So, in order to automate the designing process, we should replace gradually the designer by a

partially intelligent CAD system. The more our system is capable of standing on for the user, the more it is considered intelligent.

In practice, classical CAD systems fail to offer the designer the intelligent aid that he needs. TOMIYAMA and TEN HAGEN [TOM 87b] presented, a study of the deficiencies of such systems. In OPAL the modification process has been automated. Note that designers waste a lot of time to design assemblies that differ only marginally from already existing ones, or even to modify their preliminary versions to arrive at the final version of the design.

PROPOSED DEFINITIONS FOR FURTHER DISCUSSION:

Intelligence: To automate is to accomplish, by a machine, a task that is usually done by humans. So, to automate the designing process, we shall try to replace the designer partially by a computer system i.e. by a partially intelligent CAD system. The more our system is capable of standing in for the user, the more it is considered intelligent.

Design = top-down refinement + constraint propagation [STE 87]

Designing an assembled object is the process of defining its "hierarchical" structure (components and subassemblies represented by objects) that allows -preferably- robots to assemble it automatically, and of defining the procedural attachments to each object. Such procedural attachements will specify the routing of messages. It should be mentioned here that in CONSTRAINTS [SUS 1980] the structure is "almost-hierarchical".

Constraint propagation: is the process of determining values for all variables in the problem, when the values of one or more variables are given.

AUTOMATING THE DESIGNING PROCESS:

YOSHIKAWA [YOS 82] & [YOS 83] presented a general design theory of CAD systems to be used by a superman. Then he showed how deduction rules may be used by a designer other than the superman to deduce, from his knowledge base, answers to what the superman would judge immediately. Later, with ANDO, they showed how an intelligent CAD system can be integrated with a process planning system [AND 86]. TOMIYAMA and TEN HAGEN presented a study of the deficiencies of classical CAD systems , the concepts for future systems and a new concept of CAD systems which they apply to construct an intelligent integrated interactive CAD system [TOM 87A] and [TOM 87B]. A theoretical framework based on design transactions, for organizing the interactive CAD process was presented by TAKALA and WOODWARD, [TAK 88]. The notion of a transaction will be mentioned later in this paper.

We believe that intelligent CAD systems are the kind of support that should be offered to the designer and that OPAL is a step towards such systems. Note that designers waste a lot of time to design assemblies that differ a little from already existing ones. OPAL saves most of that time`tracking consequences of changes by replacing the designer in different tasks of the design and modification processes, i.e. by automating them.

WHY OBJECT ORIENTED PROGRAMMING FOR CAD PROBLEMS?

In procedure oriented programming, widely used in almost all kinds of applications, one can distinguish procedures and data. Procedures are activated (when running a program) and data iareused. On the other hand, in object oriented programming, programs are organized around entities, called 'objects'. Objects have local procedures (called methods or functions) and local data (called instance variables or attributes).

We propose using object oriented programming techniques for assembly problems. Indeed, the independent nature of objects permits them to represent the independent parts of an assembly, and the relationships among the parts. Assembly constraints are represented within methods as mathematical formulae to be verified. Based on artificial intelligence techniques, object oriented programming is expected to be one of the most useful tools in the design and applications of robotics, upon which automatic assembly will be mainly depending in the near future.

Object oriented programming proves to be suitable not only for CAD/CAM and robotics applications [KIT 86, GLO 86, SAU 86, AKM 88], but also for database systems [BAI 87], computer vision systems [BOL 86] and [PHA 86], pattern recognition [AYR 86], designing user computer interfaces [FOL 88] and even for applications in chemical industry [LOU 88].

The paper introduces a simple example in section 1, giving the object structure; then a discussion of constraint propagation is done in section 2; section 3 presents the actual implementation environment, object formats and message passing mechanism; section 4 gives some considerations for further work.

1. REPRESENTATION

We must summarize briefly how we represent our objects in the simple case of a mechanical assembly consisting of two plates bolted by means of a screw, a nut, and a washer (Fig 1).

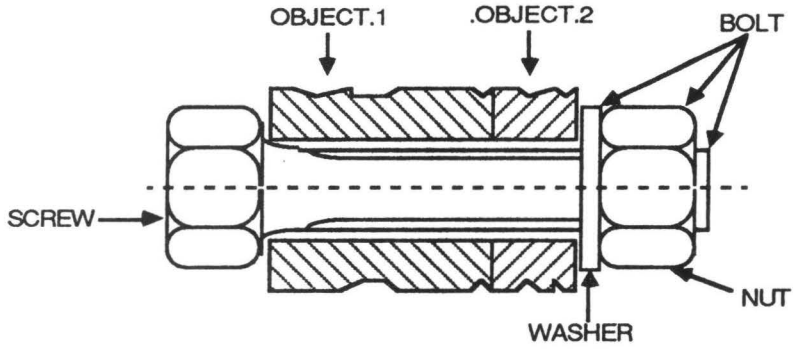


Fig.1: Assembly using a BOLT

If we have to redesign such a system, due to changes in specifications (plate thickness, screw diameter, screw length, etc.), then consequences may either be marginal and result in slight dimensional modifications, or, due to external constraints (e.g. resulting from standard tables),

lead to significant changes in the structure (like needing a second washer, or requiring redesign of the plates).

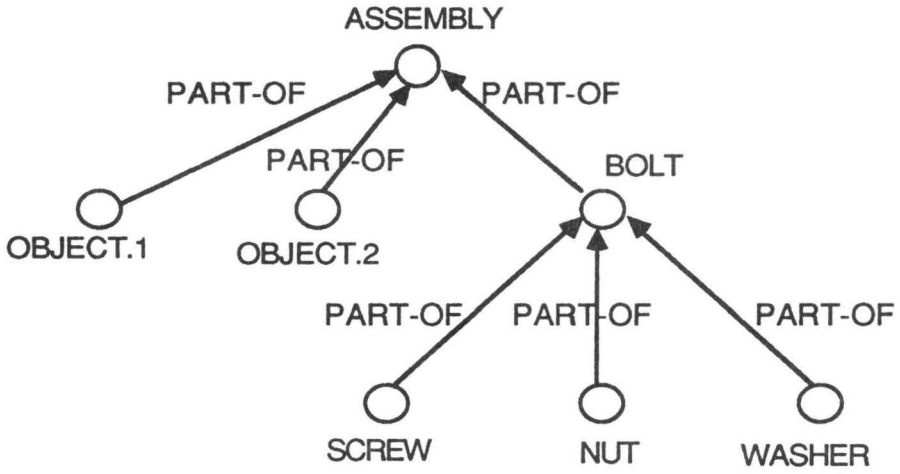


Fig 2:Tree structure of the example

The overall system is modelled as a PART-OF hierarchy of classes, like a composite object [KIM 87] (Fig 2). Note that each class in the hierarchy may be linked to others by various labeled properties inducing a general graph structure containing cycles. An example of cycles would be induced by the property "next-to". The topology of the example in our system can be made more complex as represented on Fig 3, which displays relations like "next-to" or "on" between subparts. Note the local loop involving "next-to" at the nut level, if, for example, we want to put two parts next to each other on the shaft.

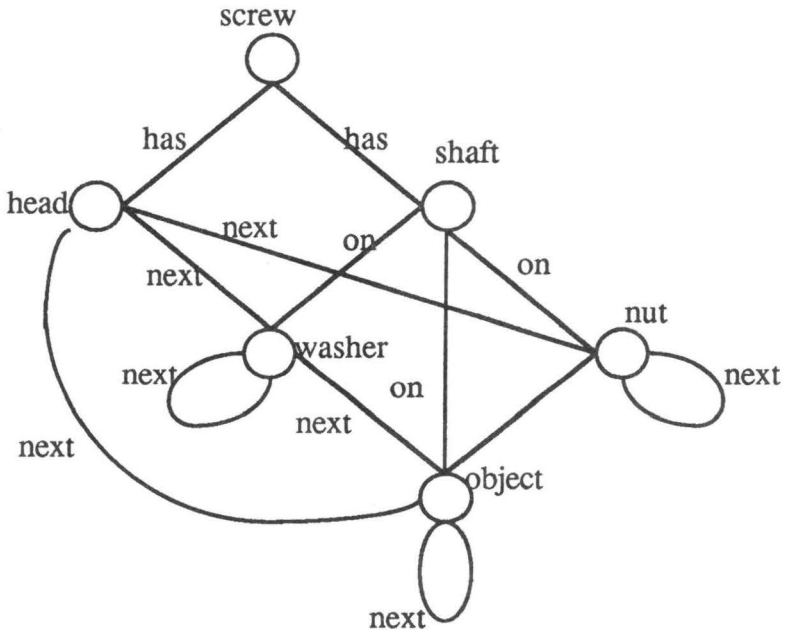


Fig 3 Semantic links between classes

Note also that Fig 3 has the well known drawback of allowing only binary relations between

classes. Now, if we need a concept like "between", we have to create an abstract "between" class such as shown in Fig 4.

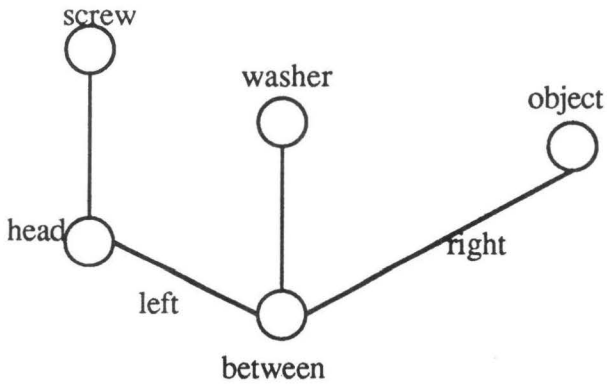


Fig 4 Representing the relation between

Such an approach is discussed at length in [FAH 79] in the context of semantic networks.

It is also clear that caution must be exercised in choosing the level of detail of the actual representation, which depends on the type of problem to be solved.

On the other hand the actual implementation of such a structure is quite straightforward as will be seen in section 3.

When using such a representation two types of constraints appear:

- *topological constraints* reflecting the possibility of the structure or of the functional design (e.g; necessity to end with a nut on the shaft).
- *geometric or dimensional constraints* resulting from computing various parameters (length of the shaft, diameter, size of the holes in the plates, etc.)

We shall not address here the topological constraints, which can be taken care of by a TMS approach, but can also bring a whole set of new problems, if one introduces desirability degrees, thus relaxing the true-false value of some topological constraints.

2. CONSTRAINT PROPAGATION

This section addresses the problem of constraints deriving from computational results on a given topology. We however do not exclude changes in the topology brought by dimensioning considerations (like putting an additional washer), but this is considered here to be marginal.

The bibliography of numeric constraint propagation research work shows that most pioneering work in the subject have been applied to electric circuits [STE 80b]. Indeed, they may be too

complex, but still always constructed of adders, multipliers, ... etc. On the other hand, components of mechanical assemblies may have not only simple forms (sphere cylinder,...), but also complex surfaces (free form surfaces). That freedom of representation discouraged many people from applying constraint propagation to the design of mechanical parts. Also in mechanical assembly, we may have a variety of constraints, while in electrical engineering there is some kind of harmony in that field.

The question is how to control constraint propagation in a consistent way so as to avoid infinite loops or excessive computation times.

Let us consider our previous example. Since everything in our system is an object, we can thus decide to change the diameter value of the screw shaft by sending a message to the screw instance. When the concerned screw receives the message, it does a table look up (by sending a message to standard tables). Thus, propagation can occur in two ways:

(i) along the IS-PART-OF link to a higher level of assembly,

(ii) along "local" links such as inverse ON, to nut, washers, and bolted objects (plates).

In case (i) the higher level must redistribute computations to lower levels, before being able to recombine results, like, for example, changes in the thickness of plates resulting in changes in the shaft length. Thus there is a possibility for iteration, hence for infinite loops.

In case (ii) one verifies first locally and compute all values before sending all values to the higher level. The main problem is to decide which links to follow to propagate "local" changes.

Although it would appear simpler to follow the first strategy, the problem becomes more complex when there are more than two levels in the PART-OF hierarchy, since it is not obvious that climbing to the root (!) is the best strategy.

The early mechanism we had chosen for OPAL was of type (i)[ELD 87]:

-modify an object saving old values,

-send everything up one level to upper subassembly,

-send down to other objects of the first level, to modify them,

-the upper subassembly, verifies the constraints, then propagates up one more level,

-if global constraints are satisfied, accept changes, otherwise declare failure resetting all parameters to their initial values.

Like in most other systems, there was no provision against cycling, since a modified parameter could be modified again, and no backtrack other than failure.

Clearly such issues must be addressed, since failure handling is one of the important problems

in constraint propagation. Different approaches have been used and several expert systems have been written to address the problem. To be mentioned here as examples and not as a taxonomy: STALLMAN and SUSSMAN[SUS 77] proposed using dependency-directed backtracking. MALIN and LANCE [MAL 87] analyse "the tasks and problem solving methods used by an engineer in constructing a failure management expert system from design information about the device to be diagnosed". CONSTRAINTS [SUS 80] introduced the notion of "slices", redundant relations that we add to augment the original constraint network to solve the loops problem in constraint propagation systems. On the other hand, DECHTER and DECHTER showed how to remove redundancies in constraint networks and pointed out some benifits for that [DEC 87].

3. IMPLEMENTATION

The backbone of our approach is the constraint propagation mechanism that influenced our choice of programming techniques. We implement the components as objects using an object centered representation, compatible with object oriented programming techniques, because we found that the independent nature of objects to be suitable for representing the independent parts of the assembly. All the subassemblies, and even the final assembly are represented as objects, just like the primitives (simple components). Such uniformity of representation facilitates the implementation and hence the designer's task. The object centered representation model offers an efficient way for representing the relationships among the parts. It is used extensively by the message passing mechanism for propagating the constraints. Thus many pointers can be managed automatically that a procedural oriented programming user would be forced to manage explicitely.

Most famous work in constraint propagation consists essentially in writing a special language that is well adapted to facilitate the propagation process [BOR 79, STE 80]. We choose a different approach since OPAL is implemented using the PDM format for describing objects, and the BOSS message passing environment. We describe briefly their characteristics.

3.1 PDM FOR OBJECT REPRESENTATION [BAR 79]

The Property Drivent Model (PDM) represents objects as *recursive frames*. Objects are described in terms of properties and attached values, properties may be *terminal*, i.e. with associated atomic values, or *structural*, i.e. linking several objects. Objects may be instances or classes. A model of classes exist, called meta-class, and more importantly, all attributes or properties are themselves objects, which makes the model meta-circular (see [MAE 87] for a discussion of computational reflexion). Attributes are multivalued and objects can be indexed using atomic values attached to terminal properties. In addition object can be permanent using secondary storage, and shared between users (multiple access is provided) [BAR 86a, 86b].

PDM was used to develop VORAS, a family of data and knowledge base prototypes at UTC, and lead to the commercial product G-BASE™.

The internal format of the objects of our small example is given on Fig 5.

```
$BOLT.4
      (($TYPE $BOLT)
       ($BOLTP.DE $ASS.6)
       ($SCRP $SCR.1)
       ($NUTP $NUT.2)
       ($WASP $WAS.3))
 $SCR.1
      (($TYPE $SCR)
       ($SCRP.DE $BOLT.4)
       ($DIAMT   5.0)
       ($SHLEN  25.0)
       ($THLEN  14.0)
       ($HDIA    8.0)
       ($HTH     3.5))
```

fig.5: Occurences of BOLT and SCREW objects (Internal format)

3.2 BOSS FOR MESSAGE PASSING

BOSS (Basic Object Simulation System) [BAR 88] is an environment originally developed for studying object oriented language features and incorporating most features found in known languages, plus additional ones:

-objects are defined with the PDM format,

-methods are first class objects,

-multiple inheritance,

-some basic methods are not attached to any classes: *universal methods*

-delegation can occur through a special subset of universal methods, called *local methods*,

-inheritance is different for values (defaults), properties (structure), methods (behavior),

-inheritance can be dynamically and locally modified,

-atomic values can be handled as objects,

-methods are compiled incrementally when needed, increasing the overall efficiency.

Actions are started by sending messages such as

```
(send '$SCR.1 '=modify-diameter 6.0)
```

which could be activated from menus.

3.3 OPAL IMPLEMENTATION

OPAL benefits from the characterisics of the PDM and BOSS. Indeed many aspects of the

needed language exist either explicitly in BOSS, or implicitely implemented in the methods. For example, we didn't need to implement the function "get-value" [ABE 85], since we had in BOSS the universal method "get" which gets the value of any desired terminal property integrating default values and inheritance. Also the function "set-value" is replaced by the universal method "set", and so on.

When a request for change or for action is sent to OPAL, a cascade of messages is triggered, computing new values and checking constraints.

In the first version of OPAL values subject to changes were saved in a global variable till the end of the constraint satisfaction process. In case of success, the global variable was emptied, else it was unstacked and the values were used again. (Saving values can also be done inside the components themselves: the value of a parameter can be stored as a list of elements, the first being its value at the current state, the rest being older values. This requires more work, but avoids the use of a global variable, which is generally recommended for complicated systems when the number of such variables increases and the programmer looses track of them.)

In the new version of OPAL we modified the send function of BOSS in order to do local backtracking to be able to check for possible cycling. This is done as follows:

Before executing the action corresponding to a message the old context is saved. Then, if the method returns a failure (special value `*failure*`), then the global context is restored to its saved value.

It is the responsibility of methods to check locally if an attempt is done to set a variable or parameter to a value previously examined then discarded.

Iteration over possible values is done within each method, like e.g.

```
(while (and (failp (send <object-id> '=modify test-value))
            (< test-value max-value))
       (incf test-value delta))
```

where all values incremented by a fixed quantity delta are tried until one succeeds or until the maximum is reached (instead of `incf` one could use a generator, e.g. including messages to a table of standards).

Propagating the constraints is thus done essentially through (inside) the methods where OPAL collects the necessary parameters and verifies the constraints. Control is distributed throughout the methods.

In OPAL different types of constraints can be accomodated, e.g.
1-constraints inside a component: the screw's dimensions are dependent upon its nominal

diameter (constraints imposed by standard tables),
2-constraints between two components:
nut's diameter = screw's diameter,
3-constraints among different components:
non threaded length ≤ total thickness of objects ≤ shaft length.

In every case, except perhaps for standard tables, the constraint is an algebraic formula to be checked (or satisfied). We can imagine many other types of constraints especially those concerning mating features for assemblies of complex shaped components.

Note that although algebraic equations can be represented, using adders and multipliers, as electric circuits, we do not feel that it is that easy to represent inequalities or worse, standard tables. In such a situation, the designer needs a flexible representation of constraints that facilitates his task. In object oriented programming has has that flexibility.

In OPAL, there is no node representing the constraint. These nodes had the role of the "maestro" in most constraint propagation languages. Their work can be donne within a function attached to a component or a subassembly. Propagating the modifications, collecting the necessary parameters, testing the constraints and finally diffusing the consequences. All that is usually donne by message passing. Thus we note here that the cocept of applying the assembly operation to two or more objects and verifying a constraint after that we explained in previous paper [ELD 87] is exactly what TAKALA and WOODWARD called latter a "transaction" [TAK 88]. The difference is that their operation is a constructive solid geometry operation and ours is the assembly operation.

In the actual state of the system, we don't attache a weight or a hierarchy to costraints. All constraints should be satisfied otherwise the modification process terminates with failure and all the modifications are cancelled, that is why there is no hierarchy on the constraints like in [BOR 87]. But such a hierarchy can be implemented by creating "constraint" objects which may be attached, by a structural property, to the object representing a component or a subassembly which will then pass some of its methods (namely those concerning the constraint verification and perhaps, propagation) to the new object.

4. CONCLUSION AND FURTHER WORK

We believe that a good CAD system should be able to learn of previous attempts of modifications that were rejected (for violation of constraints). The designer should not be replaced completely by the computer, butpartially because we need the creative part of human beings. A good CAD system should be able to learn of previous attempts of modifications that didn't work.

The separation between the laguages written for numeic constraint propagation and those for symbolic ones should not exist anymore. They must be coupled together. [KIT 87]. That will a step further to apply the same constraint propagation language in CAD design and modification, in database systems [SHE 86] and in job-shop scheduling [FOX 83]. That will be very usefull if applied to the IIICAD system [TOM 87b].

One of the most important applications of constraint propagation is tolerance accumulation which is essential in automatic assembly by robots especially for those surfaces that are supposed to be gripped by the robots hand. That should be done during the design process and not after.

5. REFERENCES

ABE 85 ABELSON, H., SUSSMAN, G.J. and SUSSMAN, J.,
"Structure and interpretation of computer programs", MIT Press, 1985.

AKM 88 AKMAN, V.,
" geometry and graphics applied to robotics", in [EAR 88]

AND 86 ANDO, K and YOSHIKAWA, H.,
"CAD/CAM integration for production management ", IFIP WG 5.7 working conference on new technologies for production management systems, october 86, Tokyo, Japan.

AYR 86 AYRAL, B., "Un générateur de plans opératoires en reconnaissance de formes", GI Memo, U. of COMPIEGNE, FRANCE 1986,(in french).

BAI 87 BAILLY, C. and LENAIN, C., "Bases de données et programmation par objets", GI Memo, U. of COMPIEGNE, FRANCE 1987, (in french).

BAR 79 BARTHES, J.P., VAYSSADE, M. and MIACZYNSKA- ZNAMEROWSKA
"property driven databases", 6th IJCAI, Tokyo (1979).

BAR 86a BARTHES, J.P.,
"VORAS, blending frames with DBMS", GI Memo, U. of COMPIEGNE, FRANCE, 1986.

BAR 86b BARTHES, J.P.,
"Share a frame: the next step", GI Memo, U. of COMPIEGNE, FRANCE.

BART 88 BARTHES, J.P.,
"BOSS version 2.2" GI Memo, U. of COMPIEGNE, FRANCE, 1988, (in french).

BOL 86 BOLLON, H., "Contrôle hétérarchique en vision par ordinateur", thèse de docteur ingénieur, Institut Polytechnique de GRENOBLE, janvier, 1986, in french.

BOR 87 BORNING, a., DUISBERG, R., FREEMAN-BENSON, B., KRAMER, A., and WOOLF, M., "Costraint hierarchies", in OPS87, pp 48-60.

DEC 87 DECHTER, A. and DECHTER, R.,

"Removing redundancies in constraint networks", AAAI 87, pp 105-109.

EAR 88 EARNSHAW et al. "foundations of computer graphics and computer aided design" NATO ASI series, springer verlage.

ELD 87 ELDAHSHAN K., and BARTHES, J.P.,
"Use of OOL mechanisms for computer aided design of mechanical assemblies", proc. Int. conf. "Modelling and simulation" Cairo (egypt), March 1987, vol. 3a, pp 13-24.

FAH 79 FAHLMAN, S.E.
"NETL: A system for representing and using real-world knowledge, MIT press, 1979.

FOL 88 FOLEY, J.,
"Models and tools for the designers of user-computer interfaces", in [EAR 88].

FOR 88 FORREST, A.R.,
"geometric computing environment: computational geometry meets software engineering", in [EAR 88].

FOX 83 FOX, M. S.,
"Constraint-directed search: A Case Study of Job Shop Scheduling", PhD CMU 1983

GLO 86 GLOESS, P. and MARCOVICH, J.,
"OBLOGIS, a flexible implementation of PROLOG logic and its application to the design of a broaching expert system",
First International Conference on Applications of A.I. in Engineering Problems, SOUTHAMPTON, ENGLAND, April 1986.

KIM 87 KIM, W., BANERJEE, J., CHOU, H., GARZA, J. and WOELK, D.
"Composite object support in an object-oriented database system", in OPS 87, PP 118-125.

KIT 86 KITAOKA, S., "Experimental CSG environment for modelling solid", Visual Computer, vol. 2, no. 1, pp 9-14.

KIT 87 KITZMILLER, C.T. and KOWALIK, J.S.;
"coupling symbolic and numeric computing in knowledge-based systems", AI

LOU 88 LOUBEYRE, R.
"Etude de la faisabilité d'un système expert temps réel en contrôle des procédés. Application au démarrage automatique des installations chimiques continues" thèse de doctorat, university of compiègne, march 1988, (in french).

MAE 87 MAES, P.
"Concepts and experiments in computational reflection", in OPS87, pp 156-167.

MAL 87 MALIN, J. and LANCEE, N.,
"Processes in construction of failure management expert systems from device design information", in IEEE Transactios on SMC, vol.17, no.6, nov/dec 1987, pp 956 967

OPS 87 OOPSLA 87.

PHA 86 PHAM, H., "Contribution à la définition d'un système de vision bidimensionelle orienté objet. Implantation de base", Thèse de docteur ingenieur, Université de technologie de Compiègne, decembre 1986. (In french).

SAU 86 SAUL, Y., "A new algorithm for object oriented ray tracing", computer vision, graphics and image processing, vol. 34, pp 125-137.

SHE 86 SHEPHERD, A. and KERSCHBERG, L.,
"constraint management in expert database systems", in proc. 1st int. workshop on expert database systems pp 310-331.

STA 77 STALLMAN, R. M., and SUSSMAN, G. J.,
"Forward reasoning and dependency-directed backtracking in a system for computer aided circuit analysis", AI, vol.9, no.2, pp 135-196.

STE 80a STEFIK, M.J.
"Planning with constraints", PhD, Stanford University, Computer Science department, January 1980.

STE 80b STEELE, G.,
"The definition and implementation of computer programming language based on constraints", PhD, MIT, august 1980.

STE 86 STEFIK, M. and BOBROW, D.G.,
"Object oriented programming: themes and variations", Artificial Intelligence magazine, vol. 6, no. 1, (1986), pp 40-62.

STE 87 STEINBERG,
"Design as refinement plus constraint propagation: The VEXED Experience", 6th national conference on artificial intelligence, Seatle, washington, AAAI july 87.

SUS 80 SUSSMAN, G.J. and STEELE, G.L.
"CONSTRAINTS A language for expressing almost-hoerarchical descriptions", AI, vol. 14, no.1, pp 1-39.

TAK 88 TAKALA, T., and WOODWARD, C.,
"Industrial design based on geometric intentions", in [EAR 88].

TOM 87a TOMIYAMA, T., and TEN HAGEN, P.J.W.,
"Organisation of design knowledge in an intellegent CAD environment", Proceedings of the IFIP working group 5.2 conference on expert systems in CAD, february 87, Sydney AUSTRALIA, GERO editor, North-Holland.

TOM 87b 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, april 87.

WIN 84 WINSTON, P.H.,
"Artificial Intelligence", Addison-Wesley.

YOS 82 YOSHIKAWA, H.,
"CAD framework guided by general design theory", proceedings of the IFIPwg

5.2 conference on CAD systems framework, ROROS, NORWAY, BO and LILLEHAGEN editors, North-Holland 1983.

YOS 83 YOSHIKAWA, H.,
"Automation of thinking in design", proceedings of CAPE 83, Amsterdam, North-Holland, pp 405-417.

THESYS – Implementation of a Knowledge Based Design System with Multiple Viewpoints

K. MacCallum
S. Green

THESYS - Implementation of a Knowledge Based Design System with Multiple Viewpoints

K.J.MacCallum
S. Green

CAD Centre

University of Strathclyde
131 Rottenrow, Glasgow, G4 ONG

Phone: 041-552-4400 x3134
email: janet.ken@uk.ac.strath.cad

ABSTRACT

THESYS is a system which combines the representation of both numerical and spatial views of a spatial arrangement. The user of the system can develop and evaluate the object model, using the facilities given by the system.

After outlining the general strategy for building intelligent CAD systems taken by the CAD Centre at Strathclyde, the development of THESYS from design to implementation is described. This includes the types of knowledge which had to be incorporated and the reasoning mechanisms identified during design. An example of use of the system is given, illustrating the interface and the types of user interaction. Finally, issues that arose as a result of implementation are discussed. These were not fully specified during design, and show the importance of learning by implementation of intelligent CAD systems.

1 INTRODUCTION

Research work at the CAD Centre of the University of Strathclyde has been concerned with building CAD systems which allow the representation in a computer system of knowledge-rich models of designs. A major project within this research is to develop a system for spatial arrangement design which allows modelling of spatial concepts at the early stages of design, in advance of geometric modelling, when information is incomplete. Quantitative information and reasoning co-exists with qualitative information and reasoning; that is numerical co-exists with spatial.

Both types of knowledge, numerical and spatial, are important in the initial stages of certain classes of engineering design problems when concepts of arrangements are being formed and basic constraints resolved. Typically, we would expect such problems to arise in building, offshore and aircraft design. Obvious extensions would be in plant, PCB and factory design. Representation of the two types of knowledge will allow models of arrangements to be created in a computer in advance of a full geometric representation, and allow the models to form the basis for design assessment, for the capture of design expertise, and for further specification. The two different types of information are seen as two viewpoints of a design model in a spectrum of possible viewpoints. Bringing them together raises some important issues in terms of consistent representation and reasoning.

The spatial arrangement project is funded under the Alvey initiative and is in the final year of three. A prototype system called THESYS has been implemented, building on previous work in which some basic formalisms for modelling both viewpoints had been developed (1,2). The system is currently being evaluated by our industrial partners who provided specialist expertise about spatial design throughout the project.

The aim of this paper is to describe the major implementation

issues which have arisen during system development, with particular emphasis being given to the integration of the two main viewpoints, numerical and spatial.

2 THE INTELLIGENT DESIGN ASSISTANT

CAD systems, in one form or another have been available to designers for many years. Of those which are concerned with spatial arrangement, most concentrate on allowing the designer to create and manipulate a geometric model of the object under design. Usually the model has limited information about other aspects of the object. The geometric modelling CAD system provides essentially a mechanism for visualising the object, modifying and redrawing it quickly. Specific links to certain types of post-processing, such as F.E. analysis or NC tape generation may be incorporated.

The strategy adopted in the CAD Centre is to use knowledge-based techniques, extending the model building approach to include other viewpoints of the object, with the objective of providing facilities on the computer which will enable consistent models to be represented simultaneously. This should allow the designer, who already creates and manipulates models of the object in various forms (computer and non-computer based), to express more information to the computer throughout the design process. It should also provide a firm platform on which design and domain expertise can be introduced to support design activities at the initial stages.

Providing a computer system with new types of knowledge allows it to play a different role in the design process. In the CAD Centre, we have defined this role of the computer as that of an Intelligent Design Assistant (IDA) (3). The IDA is regarded as an independent "thinking" process which can communicate with the designer within a specific problem domain of interest. The user initiates any discourse with the IDA, retains authority and control of the progress of the interaction, and has

responsibility for the correctness of the results. The user decides which path to take to improve and build the model of the object using the tools provided by the system.

The IDA, communicating with the user, is an active partner in the process of design modelling and solution searching. It can assess the feasibility of concepts, identify the implications of concept changes, suggest possible solution paths and assume much of the burden of mundane and repetitive analysis tasks.

To begin to realise the IDA concept, specific knowledge types need to be identified and defined (4). We have chosen to integrate numerical and spatial commonsense knowledge as a first step in the development of more intelligent design systems.

Numerical knowledge is important in engineering design. Initially, a numerical model is built which allows the major performance parameters to be estimated, usually using the information given in the specification and past experience from the designer. In many cases, as the design proceeds, this numerical model will be refined - more precise numerical relationships will be used and more information will become available. Commonsense spatial knowledge allows the expression and manipuation of basic concepts about the associations which exist between elements of a spatial arrangement. These are likely to be determined before a full geometry for the arrangement has been defined. If spatial constraints are ignored in the initial stages of design the whole layout may be unsuitable for the function or require extensive redesign.

There is, however, extensive interaction between the numerical and spatial models. Often, for example, spatial concepts and relationships will be given approximate quantifications such as "aspect ratio of a space". Conversely, quantitative information may be used to deduce some geometrical properties of a space. The final outcome of the spatial/numerical interaction will be a more complete model which includes sufficient geometry to link to conventional CAD draughting systems.

3 DEVELOPMENT OF THESYS

3.1 Background To The System

The development of THESYS has been based on experience gained in building two individual systems, DESIGNER (1) and SPACES (2). DESIGNER allows the development and manipulation of a numerical model describing a design object. This model represents numerical characteristics (such as length, mass, centre of gravity) and the engineering relationships which allow the values of these characteristics to be estimated. The SPACES system allows the designer to construct a model of a spatial arrangement expressed in commonsense terms, using concepts such as spaces, boundaries and points, and logical relationships such as next to, colinear, at right angles to, left of and right of. Since these systems allow the development of models containing types of information very different from one another, the operators which allow the models to be constructed are also of different types.

The aim of THESYS is to integrate the numerical and commonsense knowledge modelling, providing the opportunity to develop links between the two types of knowledge, and to investigate the functionality which a designer may require if both types of knowledge are available in a system simultaneously. It is clear that the need for a suitable interface for a designer is also an important factor in the success of the system.

3.2 System Design

The initial phase of system design was to determine the types of behaviour we required THESYS to exhibit in relation to the types of knowledge which were to be incorporated. This behavioural specification was developed from structured interviews with practising designers, and studies of their approach to solving problems by hand.

Five main types of activity which would be needed in a system were identified:

modelling - to allow the designer to describe the problem, to build and modify possible models and to describe solutions.

evaluation - to resolve constraint conflicts, identify inconsistencies and compare predicted and required performance properties of an arrangement.

display - to show any model information or knowledge to the user.

explanation - to show how knowledge has been used in the system, or how conclusions have been reached.

control - to change the autonomous working of the system and the degree of information required in the interface, and to control the session management.

The behavioural specification provided the broad description of the system facilities and an indication of the type of interaction which the user may have with the system.

A more detailed functional specification was then developed to establish how these facilities should be provided and the type of structures necessary to link behaviour with the knowledge required to provide such behaviour.

The first step towards the functional specification was to organise the knowledge which we expected to have in the system. Although we were principally concerned with numerical and commonsense spatial knowledge, our studies of the design problem resulted in a more extensive list.

The knowledge types identified were:

solution knowledge (sk) - the representation of the object being designed. This is created and modified using modelling commands

problem knowledge (pk) - the representation containing the statement of the problem, or the design requirements. This model contains the "desired" values and relationships for the object, some of which will have to be included in the solution, others of which are desirable

general problem knowledge (gpk) - the general representations of types of things that may be expressed in problem knowledge. Current implementation includes the form of numerical goals

spatial commonsense knowledge (sck) - the general representations of spatial objects and spatial relationships

numerical commonsense knowledge (nck) - the general representations of numerical concepts and of the form which relationships may take

domain knowledge (dk) - knowledge about the subject of the problem which may be useful in the problem solving process

design process knowledge (dpk) - knowledge about how to carry out the design process to advance the design situation towards a solution.

The relationship between these knowledge types and the overall system structure is shown in Figure 3.1.

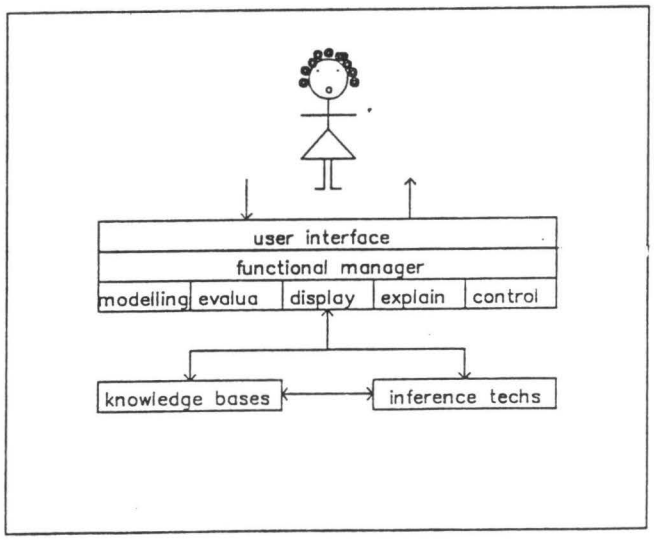


Figure 3.1 Overall System Structure

The second step was to define more clearly the expected use of the various types of knowledge and their interactions.

The types of interaction between the knowledge types were identified as:

class instantiation (cli) - creates new instances using the general definitions for objects represented in the system

goal comparison (gc) - used during evaluation to compare the problem requirements with the current solution

spatial inference (si) - used with the general definitions of spatial relationships to infer new spatial facts in the solution from known spatial facts

numerical inference (ni) - allows the evaluation of the value and precision of a numerical characteristic. The strength of influence between characteristics is also determined.

cross inference (ci) - allows numerical information to be derived from spatial, or spatial information to be derived from numerical information.

logical inference (li) - derivation of the conclusion of a rule statement given a match on the antecedent of the rule

These links are illustrated in Figure 3.2.

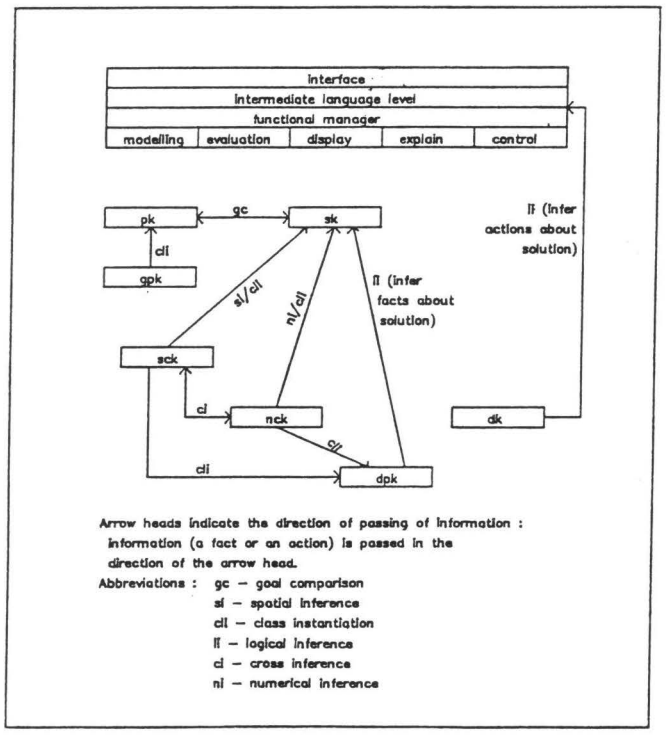


Figure 3.2 - Static Link Diagram

The third step was to define an interface structure which was suited both to a prototyping and a user situation. Figure 3.3 shows the levels of interface from the user interface through to the language level. Ideally, the system should be open at both intermediate and language levels providing varied means of interaction for expert users.

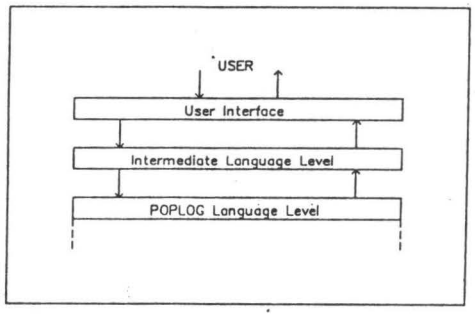


Figure 3.3 - Levels of Interface

3.3 System Implementation

The system was implemented in three separate parts which were then drawn together. These components were:

- spatial knowledge and operators
- numerical knowledge and operators
- the interface

The spatial and numerical components have a similar structure. Each must contain definitions of the types of objects which may be represented and manipulated, and have a number of operators that allow the solution model (instances of these objects and relationships between them), to be created and modified. The spatial and numerical components are linked together by

associating numerical characteristics with spatial objects. Thus, for example, the "length" characteristic represents the length of a boundary. The third system component, the interface, links the spatial and numerical components with the system user.

The implementation language used was POPLOG. This provides a combination of POP11, Prolog and LISP and was selected to allow mixed knowledge representations to be used.

The spatial component of the system was based on the experience gained from the development of the SPACES system (2). The ideas developed there were reimplemented, making use of the data structures available in POP11 which are richer than those of Prolog. This allowed the descriptions of each space to be represented as an object in one POP11 data structure and groups of objects to be associated directly with each other where appropriate. Linking data structures in this way allows local names to be used for objects throughout the system; a particular object may be identified by its position in the overall model structure and its association with other spatial objects. The spatial relationship definitions were implemented in Prolog. This allows full use to be made of the mechanisms available in Prolog to infer new facts using existing facts and general definitions. Interaction between POP11 data structures and the Prolog clauses is necessary however, and required the development of special functions to link the object data structures with the relationship definitions. These have caused us to pay attention to two particular implementation issues - the naming of objects, and the problems of using mixed representations. The spatial operators, implemented as procedures in the POP11 language are the mechanisms by which the system user may create and manipulate the spatial model of the object. Operators are defined to provide the following tasks:

- create and delete objects

- divide and combine spaces

- join and unjoin spaces

- display information about any object

All operators are available to the user through the graphical interface.

The numerical component of the system is a slightly modified version of the DESIGNER system (1). Modifications included translation into POP11 and changes made to the data structures to make them compatible with those in the spatial component of THESYS. The functionality of the system is the same as DESIGNER, as are the numerical concepts which may be represented there. These are characteristics, numerical relationships, rules to control use of relationships and values of characteristics and goals. Unlike the spatial knowledge, there are no pre-defined relationships, only a definition of the form that a relationship may take. The major functions of the numerical operators are to

- create and delete characteristics, relationships, rules and goals
- associate these concepts with each other
- estimate the value of a characteristic
- update the value of a characteristic
- establish the closeness of the model to a goal
- determine how one characteristic affects another
- display information about characteristics, rules and goals,

Like the spatial operators, the numerical operators are made available to the user through the interface. The keyboard interaction that takes place is greatly reduced from the original system through mouse and menu type interaction.

The basic mechanism which is needed to link the numerical and spatial parts together is to associate a characteristic with a spatial object. Having implemented this, the major aspect of numerical and spatial linking is in how the two parts are to be

used together. As an initial investigation into the combined use of the two systems a standard network of geometry-related characteristics and relationships is created whenever a space is created, and this network is modified when any space is divided or combined. In the numerical component of the system all characteristics must have unique names. To achieve this, all characteristic names in this network are generated automatically based on the objects to which they belong. This is not difficult to implement but causes difficulty when the user wishes to identify the area of a particular space or length of a particular boundary.

The user interface to the THESYS must allow the user access to the operators and data structures which make up the model of the design object. It tries to reflect the structure of the underlying systems, and the types of behaviour that the system may exhibit. Figure 3.4 (screendump) shows the basic layout of the screen before any modelling occurs. The window in the centre of the screen, the solution window, is the place where the two viewpoints of the model are displayed. The left hand side allows the spatial arrangement to be created and displayed graphically. The right hand side displays all the numerical characteristics, relationships and rules. The commands which may act on these entities are at either side of the solution window, and are split into two types - those for model building on the left, and those for model evaluation on the right hand side of the screen.

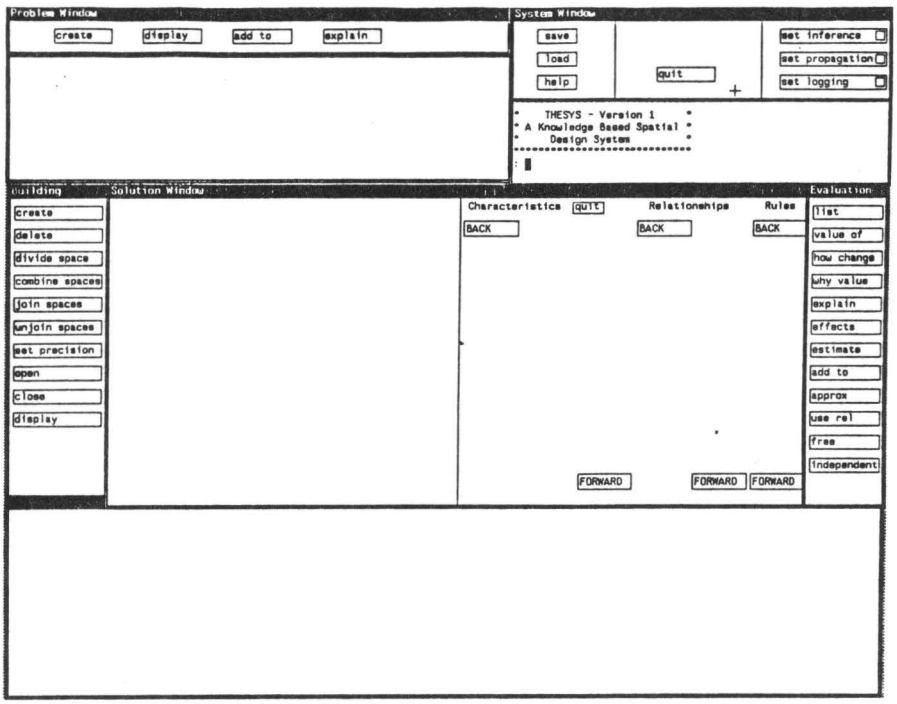


Figure 3.4 - Basic Screen Layout

The top left hand window allows the user to input information about the current problem. At present this is limited to the modelling of numerical goals but it is planned to extend this to spatial problem knowledge. The system window contains commands to do with loading and saving solution models, logging the interaction, and allowing the user to set controls on how much inferencing takes place. The lower window on the screen is for textual interaction - many commands give instructions and provide the user with information about what has occurred to the solution. The text window also accepts information from the user in response to a command.

The implementation of this interface has been done using the PWM tool provided in version 13 of POPLOG. This offers a limited set of window management and graphical facilities and was chosen because it was compatible with POPLOG. Design and implementation of the interface has raised some interesting questions about how to provide the user with an interface which reflects the underlying structure of the model and allows the user full access to the information there.

4 EXAMPLE OF USING THE SYSTEM

This section will illustrate the interaction which can occur with the system and the functions which are available to the user. At the start of any design session, the system contains only a default arrangement object which initially has no spaces associated with it. All new spaces are associated with this arrangement unless the user creates a new arrangement which will then become the new default. For the purposes of this section, all the interaction will be associated with the default arrangement.

The first step the user takes is to create a new space. Figure 4.1 shows the system before any new space is created, with the pop up menu which results from selecting the create command with the mouse. The rectangular space is defined by sketching in the solution window identifying diagonally opposite corners. The space is displayed as an approximate sketch with the "sketching approximation" controlled by the user (Figure 4.2). This approximation is delivered as a value range to the appropriate numerical characteristics created for the space. The system generates all the spatial facts deriving from creating a space together with the standard geometry network. Most characteristics are left without values, waiting for a user initiative.

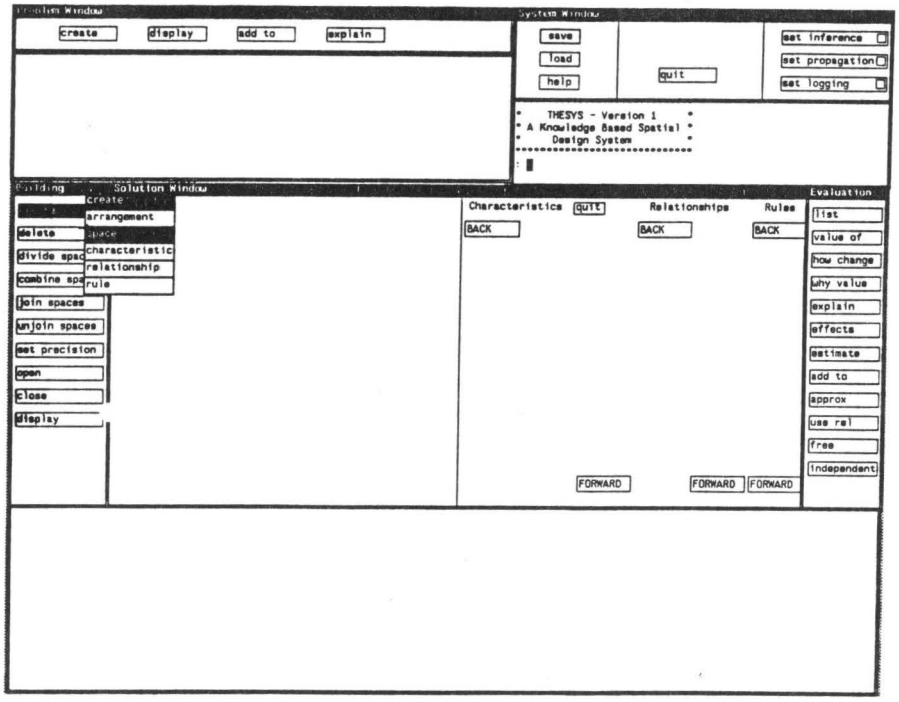


Figure 4.1 - Creating a Space

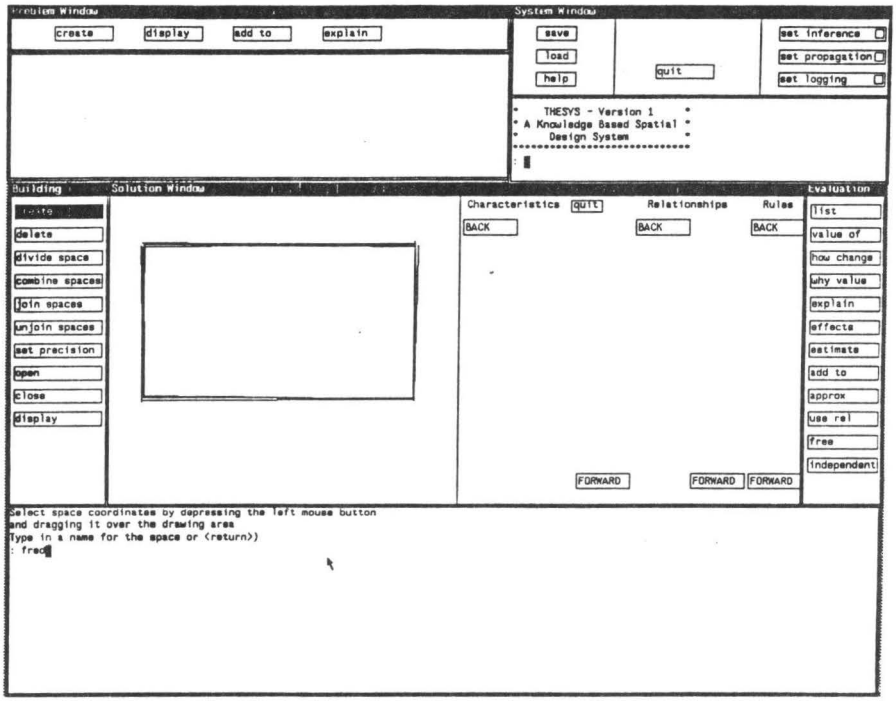


Figure 4.2 - One Space Defined

As a next step space is divided using a horizontal line. Figure 4.3 shows the result of this division, and the process of creating a second vertical division on the upper space. The lower text window gives the user instructions about how to perform the operation. The right hand half of the solution window shows some of the characteristics with automatically generated names which have been created as a result of using the spatial operators.

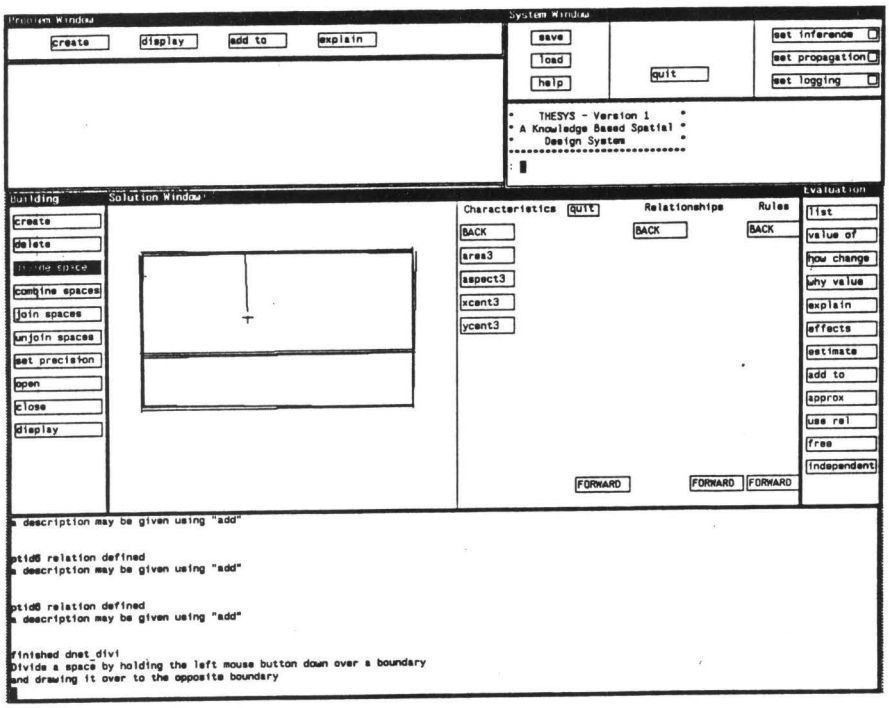


Figure 4.3 - Dividing a Space

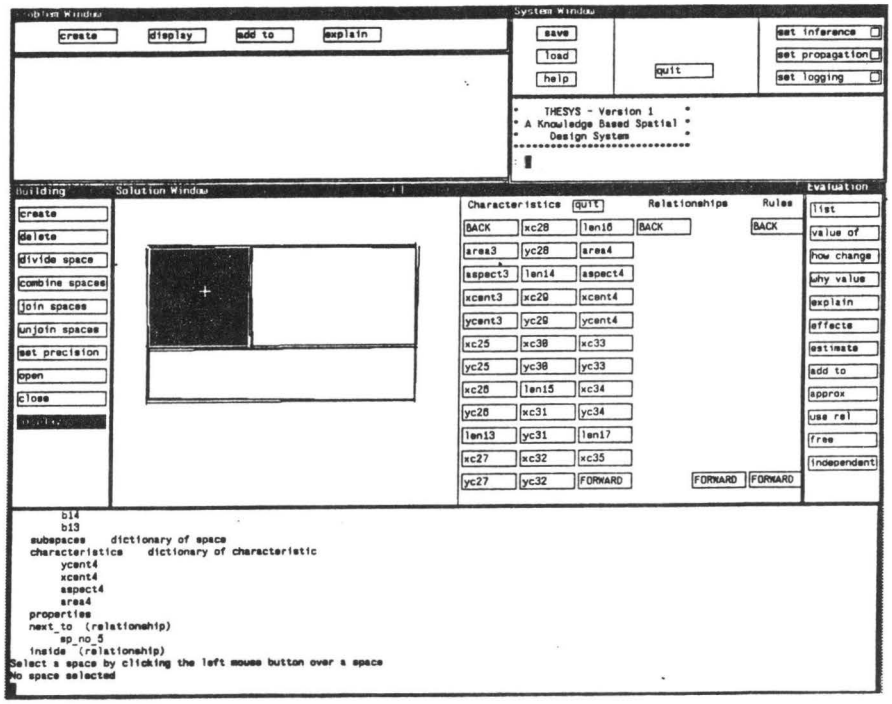


Figure 4.4 - Display Command

Information about individual spaces developed in this initial layout may be requested by the user. Figure 4.4 shows the last part of the information obtained when using the display command, selecting the relevant space, shown in black. It may be seen from the lower test window that the area characteristic for "sp-no-4" is area4. Having developed an initial layout, it is possible to derive further values of some of the numerical characteristics. As an example, the evaluation buttons are used to initiate an estimate of area of space "sp-no-4", identified by picking the appropriate characteristic. Figure 4.5 shows some of the interaction which occurs during this estimate. Once the estimate function has been selected with its characteristic, all interaction occurs in the lower text window. The area was estimated using an aspect ratio aspect4 from the sketch together with a length len16 provided by the user.

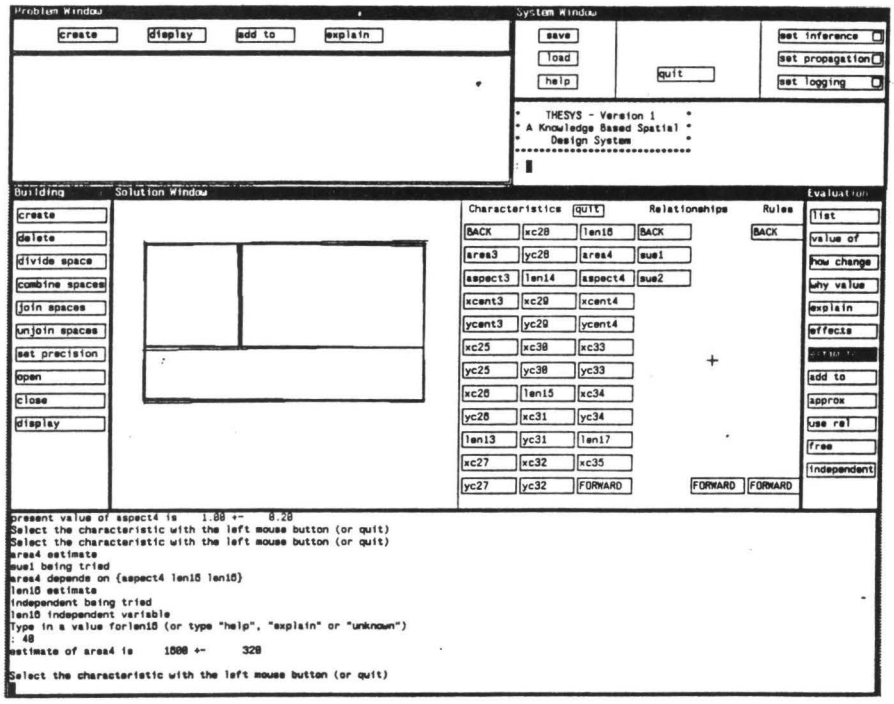


Figure 4.5 - Estimate of "area4"

The final figure shows interaction which can occur in the problem window. Figure 4.6 shows the definition of a numerical goal. Here, it has been stated that area3 should have a value greater than 50. Once defined, these goals can be used to help the designer move towards a suitable value for any numerical characteristic.

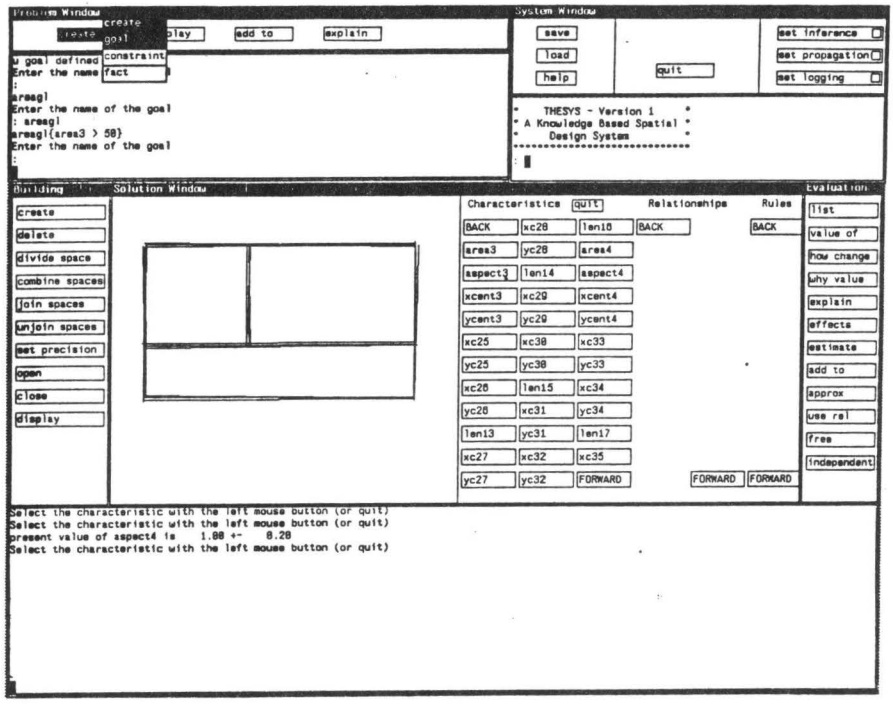


Figure 4.6 - Definition of a Numerical Goal

5 ISSUES ARISING FROM DEVELOPMENT

5.1 Naming

Naming denotes the way in which the user is able to address the objects in the model. To create and manipulate the object model, the user must be able to identify particular items within it - for example individual spaces, numerical characteristics, particular boundaries and points. The interface can go some way to solving the identification of particular items - if the user is able to point at an object, no unique identifier is necessary. However, there are occasions, such as a relationship definition when it is necessary to make text statements about an object; and

(usually meaningless) on all objects or the user has to think of unique names for all objects (time consuming).

The solution in the spatial component of the system is not to require unique names throughout, to allow users to provide names if they wish, but for the system to generate names if none are supplied by the user. Identification of objects is achieved by referencing them through their structure. For example, the boundaries of a space are associated directly with that space and are referred to through the space. To identify a boundary of a space it should be possible to say something like

boundary b5 of space s7

or boundary b2 of space s6

where s6 is identified as s6 subspace of space s11. The implementation of this type of "design" language has not been developed in this system yet, but the structure in the system should allow items to be referenced in this way.

In the implementation, non-unique names cause a problem in the link with the relationships expressed in Prolog. Prolog requires unique identifiers; so the name of an object alone is not sufficient to identify it throughout the model. In the spatial component of the system a two part name is used - the first part identifying a path through the object structure, the second the actual name. This adds to the time taken to move between the two representations.

The naming problem is more apparent in the link between numerical and spatial. The DESIGNER system is built around the use of unique identifiers - consequently all characteristics must have unique names. When associating characteristics with spatial objects, it would be natural to refer to (say) "the area of space a". However, this area characteristic must have a unique name and it is not always easy for the user to establish which characteristic is "the" area of space a. In the implementation of the default network created for each spatial object, characteristic names are generated automatically

- area 1, area 2... aspect 1, aspect 2...
  len1, len2, len3...etc.

This solves the implementation problem, but does not improve the structure and use of the solution model. Development of THESYS has indicated that a major issue is the association of numerical characteristics with objects and the interaction which occurs between characteristics associated with different objects.

5.2 Geometry Network

The geometry issue is concerned with how to represent geometrical information and reasoning in the system. In the system, it has been tackled simply by using a numerical network as generated in DESIGNER. This has allowed flexibility since there is no need to add the numerical values if the user does not know them, and the normal estimating functions of DESIGNER can be used. However the approach is a bit clumsy. Many characteristics and relationships are created, even if not used, and there is no explicit knowledge of geometry or use of geometric reasoning. More integration could be achieved by including spatial logical relationships in the definition of numerical relationships, for example, by using the coincident relationship in identity relationship of point co-ordinates.

The generation of numerical networks during the use of operators is the first step to linking some geometry into the system. Future develoment in this area is important.

5.3 Interface Design

The major issue in interface design is to convey information about the solution model to the user in a manner easily understood by the designer. It must also allow the facilities for manipulating the solution to be easily accessed. The numerical and spatial components of the system consist of many

procedure calls; often a number of procedures have to be used to perform actions which together are seen by the user as only one. In addition, the spatial component does not have a geometrical representation of the arrangement, although graphical interaction is a necessary form of communication. From this, two particular aspects arise:

- how should the approximate nature of the spatial layout be conveyed ?

- which commands should be "packaged" together?

The first, the display of approximate arrangements, is addressed by making the drawing of each space appear like a sketch. This is implemented by drawing a group of lines for each boundary whose endpoints have small but random variations with the given endpoints. The user creates, divides and combines spaces using graphical input, but initially no dimensions are given to these spaces. The appearance on the screen is one of a sketch. It is planned to implement the use of more precise diagrams so that once boundary length or space dimensions becomes more precise, the lines will lose their random appearance. The aim is to convey approximate and precise boundary lengths and position clearly in the same diagram.

The second aspect involving the packaging of commands applies more to the numerical commands than to the spatial ones. For example, in the original implementation it is assumed that all characteristics will have been created before they are used in any relationship definitions. In THESYS, the interface creates a new characteristic if an unknown one is found during relationship definition. Similarly, the definition of a relationship and its association with a characteristic requires the use of 3 different commands in DESIGNER. Mostly, these 3 commands will be used in a fixed sequence, and may be packaged together to reduce the interaction with the user. It is important however that the individual commands are also available separately.

5.4 Mixed Language Use

THESYS has been developed using the POPLOG language which allows a mixture of languages to be used - POP11, Prolog, and LISP. In addition there are facilities to link compiled routines written in other languages such as Fortran, Pascal and C. In theory this allows graphics libraries to be called easily from POPLOG. In practice, THESYS makes use of POP11 and Prolog and simple graphics facilities for the interface available in the POPLOG language. The Prolog/POP11 combination is used within the spatial knowledge component of the system. POP11 is used with a package called ODDS (4) which provides object-oriented facilities. These make the representation of spatial and numerical objects easy to manage. The logical relationships between spatial objects have been defined using Prolog statements, allowing the theorem proving mechanisms to be applied to infer new information about relationships between objects within the model. Prolog implements a different type of representation - a set of rules which define the correct conditions to infer a new relationship.

Implementation problems occur in this mixed language interaction, the main cause being the mismatch between the simple data structures of Prolog and the complex record structures available in POP11. The first problem to overcome was the representation of a complex object in Prolog terms. This has been solved by creating a unique name based on the position of the object in the spatial structure and the object name as described in 5.1. The two part name, providing the position of the spatial object and identifier at that position also provides the means by which more information from the full data structure may be obtained. Using specially written transfer facilities, the value of any slot represented in the POP11 data structure can be found from the Prolog part of the system. The major disadvantage that this has in THESYS is that it slows down execution of the Prolog inference mechanisms.

6 FUTURE DEVELOPMENTS

6.1 System Evaluation

The prototype system, once built must be evaluated to establish the usefulness of the approach and the appropriateness of the facilities built into the system. The evaluation is to be carried out in two phases, the aims of which are

- to produce an understanding of the types of objects and relationships used in spatial design and a simple descriptive model of the psychological processes underlying spatial design

- to assess the facilities for describing the problem, specifying goals and constraints, building, modifying, and displaying possible solutions and explaining reasoning.

The first phase of the evaluation acts as a basis on which to carry out the second. During this first phase designers (and non-designers) will be asked to solve simple design problems with the solution process being recorded on a video. This allows the problems to be tested to establish their suitability for use in system evaluation and a descriptive model of the design process to be developed.

The second phase will monitor designers using the system. Design problems thought to be suitable for evaluation purposes will be solved using the system and the facilities evaluated in relationship to the initial aims stated above.

6.2 Links To Geometric Knowledge

Throughout development of this system we have not tried to develop a full geometric model of a spatial arrangement. However, in two parts of the system, a partial geometric model exists.

- the interface holds a representation of the spatial objects as they are displayed on the screen.

- the small numerical network set up for each space when it is created represents point co-ordinates, boundary lengths and the areas of spaces.

The implementation of a full geometric model involves the translation of the logical spatial model into a geometric form including the resolution of parts of the arrangement when informtion is not complete. In addition, numerical information in the model may contribute to the geometric model and has also to be integrated.

6.3 Addition Of Problem Knowledge, Domain Knowledge and Design Expertise

Problem knowledge consists of all the information gained from the statement of requirements about the object. Some numerical problem knowledge has already been represented and used; numerical goals may be stated, including importances that the designer attaches to these goals. This must be extended to spatial knowledge so that particular requirements and wishes may be stated to the system, which may then be used to compare possible solutions with the requirements.

Domain knowledge and design expertise have not yet been represented in the system. It is proposed that inclusion of this knowledge will increase the assistance that the system can give to the user.

7. DISCUSSION AND CONCLUSIONS

The major part of this paper has been concerned with the stages of development of THESYS. Four main issues arose during development - the naming of objects in design systems, the use of geometry in combination with the spatial and numerical network, interface design and the use of a mixed language system (and representation) for implementation.

The most important aspect of implementation is the way in which implementation details focus on more major considerations in systems design. All the issues discussed previously were identified as a result of building the system. Other issues which have not as yet been addressed include the intelligent handling of errors and the use to which the designer may put a system which combines these particular views of the system.

The four issues described are all important for the development and use of intelligent CAD systems. While these issues had been recognised as being significant at an early design stage, a proper appreciation of the nature of the problems only emerged with a prototyping approach. The experience gained will allow a better specification of system requirements to be made for future projects.

The evaluations, which have yet to be carried out, will test the facilities provided by THESYS and help to evaluate the value of our approach for designers solving real world problems. It is anticipated that the results of these evaluations, will provide excellent opportunities to develop and evaluate improved theories of intelligent CAD.

(1) MacCallum K.J., Duffy A., An Expert System for Preliminary Numerical Design Modelling, Design Studies, Vol 8, No 4, October 1987

(2) Green S., SPACES - A System for the Representation of Commonsense Knowledge about Space for Design, Expert Systems 86, BCS 1986

(3) MacCallum K.J., Duffy A., Green S., The Knowledge Cube - A Research Framework for Intelligent CAD, Presented at Intelligent CAD, MIT, Boston, 5th-8th October 1987

(4) Bennett, M.E., ODDS User Manual
Computer Science Group, Cambridge Consultants Ltd.
Science Park, Milton Road, Cambridge CB4 4DW.

Paper Session:

Design Process

Implementing Intelligent Processors

M. Nadin

M. Novak

Implementing Intelligent Processors

Mihai Nadin and Marcos Novak
The Ohio State University
1501 Neil Avenue
Cranston Center
Columbus, Ohio, 43201
U.S.A.

1. Introduction

In studying the problems involved in implementing Intelligent Processors (IPs) we recognized that the metaphor of the design team advanced in the conceptual phase of our work (Nadin and Novak 1987) represented only a methodological framework. In order to achieve effective implementation, an operational mechanism had to be conceived. Such a mechanism had to allow for operations on design entities as they resulted in the processes of design.

More specifically, the necessary mechanism needed to be appropriate to intelligent processes applied to :

1) all categories of design
2) the wide range of designed artifacts
3) the flow state (design as a fluid process)

as defined within the conceptual model. Our implementation work led to the discovery that a combination of a genetic mechanism and neural network implementation are well suited to the problem. The most complex objects we know, living forms, have evolved through a particular kind of interaction of patterns, i.e. random mutation and cumulative change through the propagation of the coded genetic information of surviving offspring(Dawkins 1986). Design artifacts, as products of complex living entities, can be seen as the results of processes analogous to the genetic.

The need to delegate the task of the parametric variation of a large number of elements to the system, with little or no direct supervision, prompted us to develop a genetically inspired mechanism that allows design alternatives to 'evolve,' as well as to develop through direct designer-machine interaction. The high degree of

interconnection and the multiple interrelations of design parameters suggested a neural network implementation, within the overall structure of a hybrid machine. By no accident, neural networks implementations prove to be best suited for the mechanism adopted.

Our implementation framework, in outline form, consists of a network of IPs that act on diagrams; the diagrams control 'design gene' and 'design chromosome' coding strips within a genetically inspired mechanism; the 'design genes' provide weighted inputs to a neural net; and the neural net determines the parameters that form particular instances of pre-existing archetypes. Each of these aspects of the Design Machine, their interrelation and function, and the modes of operation they allow will be presented in detail below.

2. Overview

2.1 A working definition of intelligence (conceptual model revisited)

Our definition of intelligence (Nadin and Novak, 1987) is recognition, manipulation, and synthesis of patterns (and patterns of patterns). Our hypothesis is that these operations happen at many levels. At very low levels, they are evidenced in the ability of intelligent entities to observe similarity, repetition, and affinity, while at higher levels, they result in the recognition, manipulation, and synthesis of high level types (commonly known as typologies).

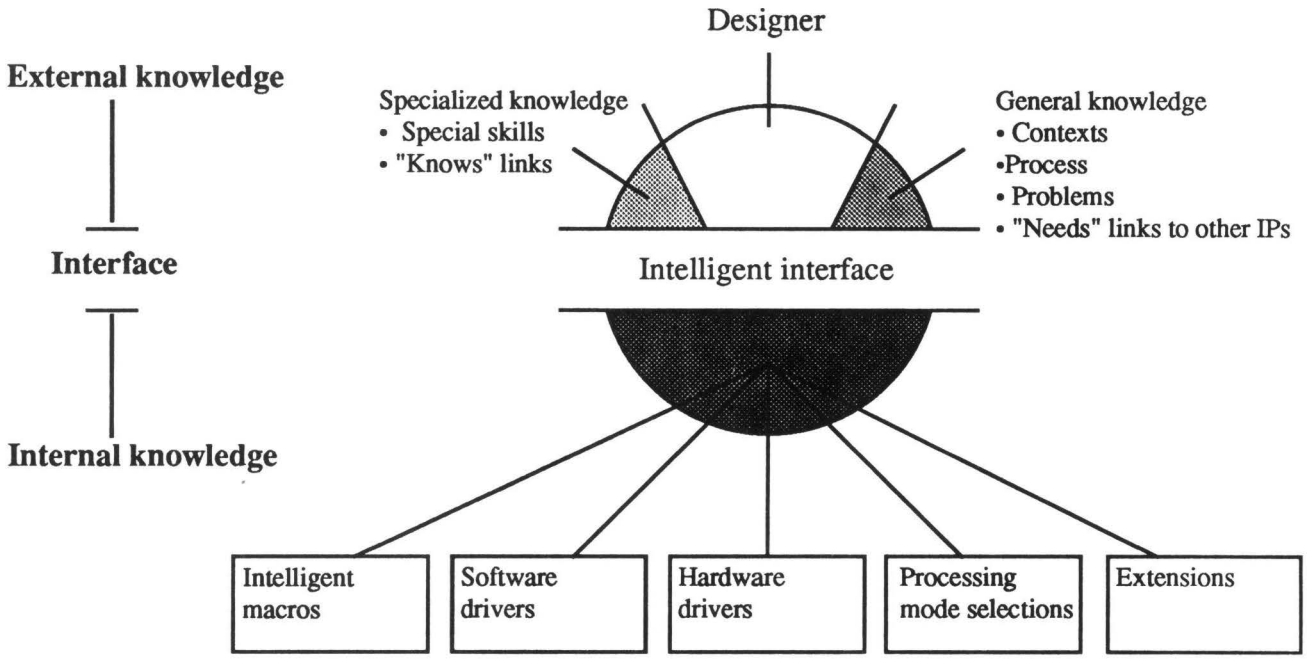
2.2 Search vs. generation

Design has often been described as a search, and various heuristic methods for dealing with what seems to be a very large search space have been proposed . Yet it often seems that designers do not search at all, but arrive at a solution by introducing a set of relationships that the design is to have. This normative aspect of design is analogous to the approach one takes in solving differential equations. While during the design of a complex object there are many instances which require efficient search techniques, the most complex and interesting designs result from the complex interactions of predetermined patterns. In the machine we propose, this is accommodated by offering the designer the capability of setting generative systems to guide the parametric variation of parts, or the capability to set the direction,

vector, or gradient of the mutation of 'design genes'. However, as it will become clear in the next section, we do not limit ourselves to the pattern imposition model.

2.3 The Design Machine as a network

MIND is a Design Machine constituted as a network of intelligent processors:



The context for its functioning is the culture at large. Within culture, the categories of designed objects, events, systems of relations, and systems of reference are acknowledged and constitute a referential framework for future design work. Newly designed items are compared to those available and accepted or refused not only in view of their intrinsic characteristics, but also in respect to the "design language" of any given time and environment.

Within the network, no hierarchy is established among the various processors. This network constitutes and embodies the metaphor of the Design Team. The "knows" and "needs" links of the IPs interact and "fire" multiple processors in order to distribute, collect, and integrate information.

The intelligent design process is one of design evolution, and consists of contributions to the final design by each member of the team. The branching out (hermeneutics) and

the coagulation of the design solution (heuristics) are the result of an implementation of intelligent parallelism as a dynamic interconnection of neural cells. The design team metaphor makes parallelism necessary; parallelism influences implementation decisions.

3. Designata and cognitive characteristics of design

The development of Intelligent Computer Aided Design (ICAD) systems requires a three part definition of the problem , derived from the answers to the following questions: How do people design? What do people design? What are the characteristics of the design process?

3.1 The 'how' of design

There is a limited number of ways in which people can design:

Design as :
- Re-Design
- Selection from existing parts/types and Recombination
- Problem-Solving
- Evolution
- Pattern-Imposition
- Invention

They are computationaly different, but an ICAD system should accommodate all of them. Our previous insistence on the pattern imposition path reflects an interest in an area of design considered until now too hard to be dealt with (Chandrasekaran 1985).

3.2 The 'what' of design

The range of designed artifacts, which we call **designata** can be specified as follows:

- objects
- events
- system of relations
- systems of reference

In this list, the first group specified is made up of design artifacts commonly identifiable as objects (graphic design in 2-D, product design in 3-D, architectural spaces as 3-D voids); the second, entities identifiable as events (sequential ceremonies or sequences of actions pertinent to a task, configurational designs, such as temporal reasoning, plans of action); the third, systems of relations (games, simulations, organizational structures, etc); and the fourth, systems of reference(artificial, notational, textual, etc.). The intention of this method of categorizing is to cover the entire gamut of entities to which the qualifier designed applies within culture, and which a design machine should aim to accomodate.

3.3 The process of design

The support of a fluid state of thinking, for which we have appropriated the term 'flow state,' is the third layer of concern. In order to maintain the appropriate cognitive characteristics during the activity of designing, we propose a range of interactive possibilities from direct manipulation of design elements to automatic design through the evolutionary model. A time-memory window (cf. 4.2.2) will be specifically implemented, given the implications of the non-serial model of design pursued.

We want to stress here that all the layers described are to be supported by the ICAD system. This definitely constrains the implementation of IPs but will result in a system which is not only a collection of tools controlled in an intelligent way, but also an environment for intelligent design processes.

4. Implementational aspects

The implementational framework of the Design Machine is constituted by the architecture through which the IPs act. The layers of the architecture are:

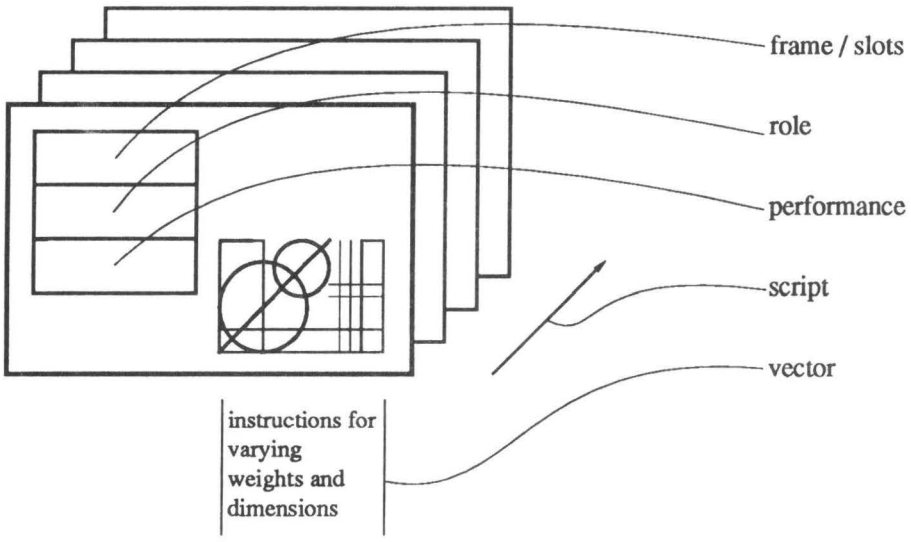
a. diagrams
b. design genetics
c. parametric types

We bring together an algorithmic and a non-algorithmic (neural network) approach. On the algorithmic side, the structure of tasks and algorithms require that we maintain an inference procedure and a representation function. The inference procedure

implemented in the activity of the IPs is abduction. In the process of design, each new potential design is an implicit explanation. These explanations are given in the form of diagrams and are applied as input on IPs, implemented as neural networks. On the non-algorithmic side the need for parallelism, robustness, and plasticity requires that we ensure relaxation and learning (basically through back-propagation). The so–called structured neural networks (Feldman et al 1988) appear to be best suited for IPs implementation. The interaction between the two approaches definitely reflects the complexity of the design tasks considered in our model. An issue of special concern is the interaction between the network and routines necessary to support design work (such as CAD tools).

4.1 Diagrams

The main element of interaction which we propose, and which is central to our data structures, is the diagram. Diagrams are structured templates for the interactive control of coding strips. At the macro level, the diagram functions as an organizing device for different kinds of information. It contains scripts of expected actions, frames of expected values with slots containing default values, and special categories called roles and performances. Roles contain information that modifies the information in the diagram to correspond to different viewpoints (e.g., designer vs. client vs. user). Performances contain information about how things should act, that is, information that can guide the system in simulations of reality.



Diagrams, by definition, explain rather than represent—showing components, arrangements, and relations—and in that sense embody a viewpoint. Different viewpoints, identifying different contexts, can thus be seen as different diagrams (i.e., showing different characteristics by explaining the object according to different "filters"). Diagrams are descriptions structured according to a determined perspective. They act as implicit context identifiers and filters. They are both visual and verbal. Since diagrams are viewpoints, multiple diagrams can be generated from the same object description by filtering out dimensions of the problem that are secondary to the task at hand. Thus, diagrams are also projections.

Diagrams are the main token of exchange between IPs and the designer. They consist of parts, attributes of parts, hierarchies of parts, degrees of similarity between parts (affinity of extensional attribute lists within a particular view) and are nested, layered, and interactive. They present relations among parts, (again attributes, hierarchies, degrees of similarity). Parametric relations (Mitchell 1987) can be described as how a change of parameter allows for the generation of new diagrams showing the influence of the chosen parameter. These relations are implemented in the algorithmic path of the machine as associations between parameters specified through functions(a generative system nmay be thought of as a collection of such functions).

Actions on the components (parts, relations, attributes) are supported by using operations on diagrams (insertions, omissions, substitutions as well as higher level operations). In view of their condition as explanatory visual descriptions, diagrams can be processed as abductions. In this case, inputting diagrams into IPs by acting on components leads to processes of abduction, i.e., design inferences.

Diagrams allow the implementation of three specific aspects of design. First, diagrams act as filters, allowing the designer to impose an abstract structure on the problem by concentrating on particular information while ignoring aspects that are not directly relevant. Second, diagrams embody viewpoints. The problem at hand is different according to the viewpoint we choose to emphasize —designer, client, user— and is represented by different diagrams. Third, diagrams imply contexts. A diagram representing the design of a bridge in the context of the historical development of bridges differs from a diagram representing the design in the context of regional differences in vernacular form, and is different again from one showing the design in the context of technological development.

The main data component in a diagram is visual, as the name implies. Parts and relations of parts are represented in a network. Each node of the diagram can be another diagram. Geometric relations in the diagram, such as size, proximity, overlap, alignment, and containment, carry meaning; modification of the diagram in such ways directly affects the internal data structures and the design itself. Although the diagram is two-dimensional, it can be interpreted as representing more dimensions. In fact, the problem of 'emergent form' (Mitchell et al 1987) pertaining to diagrams is one which we attempt to resolve by establishing 'implied' nodes that the designer has access to. Multiple interpretations of diagrams can co-exist as modified instances of the initial diagram.

For reasons of terminological clarity, we want to specify that the diagram is not an association net. This "negative" definition (what it is not) allows us to define some important characteristics:

First, it is a diagram of diagrams, i.e. it is nested.

Second, it is more than a topological graph; it is also a geometric entity. In diagrams, proximity and size, for example, are recognized.

Third, diagrams can recognize union-intersection-difference relations, etc., i.e., display properties characteristic of a set-theoretic entity.

Fourth, diagrams are used as an intelligent interface, allowing direct action on relations and parts

Fifth, they provide assistance in the 'emergent form' problem by constructing node and relation variables automatically at the intersections of pre-existing connections,

Sixth, like tracing paper, diagrams are layered, so that alternative diagrams can be superimposed and 'traced' to produce new variations.

4.1.1 Stacks of diagrams

As far as the designer is concerned, the final design is represented as a stack of diagrams. This implies that the system has an underlying layer of 'translators' which take the information in the diagrams as input and, through interaction with the design genes, or with the design chromosomes, drives the appropriate parametric modeling software. Alternative stacks of diagrams imply alternative design solutions still under consideration.

Diagrams have one additional role in the Design Machine. A stack of diagrams showing the present state of the design process and the history of the project to date is generated by the system and is accessible to the designer. Thus the 'flow state' is supported in the manner of nonlinear possible associations characteristic of human process of design.

4.1.2 Diagrams and intelligence

In order to embody intelligence, diagrams must accommodate analysis and synthesis, and have the ability to withstand incompleteness and conflict. This is accomplished by linking the diagrams to the neural network. Hence, the DM can indeed learn from the design experience as it results from its functioning. Neural networks have associative memory, adaptive learning from examples, and combinatorial optimization characteristics.

4.1.3 Four categories of design

Our initial premise was that design is essentially a visual activity occurring at a high cognitive level. We looked at what is common to all three types of design (Chandrasekaran 1985) and discovered that diagrams, as defined above, prove to be best suited for the operational mechanism that we needed. This discovery is supported by Peirce's work on thinking (1887) and the relation between diagrams and abduction. In applying diagrams in our initial concept, we discovered that the three commonly acknowledged categories of design could easily be described as particular operations on diagrams. Furthermore, a fourth category could be added:

Category Three: Modify "geometry" of existing diagram;
Category Two: Modify "topology" of existing diagram or combine diagrams;

Category One: Invent diagram;
Category Zero: Invent archetype.

4.1.4 Vector set

A data structure, which we call a **vector-set**, is associated with each diagram or stack of diagrams. This allows the designer to delegate tasks to the IPs by specifying the directions in which various variables are to be modified, either systematically, using a generative system, or randomly, in 'evolution' mode, along the genetic mechanism of design we adopted.

Behind the diagram is a second data structure which is usually hidden but which can be modified directly, either by the designer or by the IPs. This data structure is a 'list of lists', a genetic strip containing the information of the parts that make up an object. Each sub-part has a corresponding strip. Changes to the diagram are directly translated into changes in the strip. Conversely, changes in the strip directly affect the diagram.

4.2 Design genetics

The genetic mechanism employed involves an analogy that is particularly powerful in assisting the design of complex artifacts. In biology, genes control specific characteristics of a life form, such as eye color or overall height, for instance. While genes remain the same, particular 'values' in the gene can differ, producing variations of that characteristic (changing the values of a gene will change blue eyes to brown eyes). An entire life form, on the other hand, is represented by a collection of genes. Variations of this collection of genes or 'chromosome', even by simply changing the order of identical genes, produces different species.

Besides the evident analogies to the "evolution" of design, we have chosen to appropriate this system because several difficult design operations can be expressed directly in genetic terms, which in turn allow for computational descriptions and operations upon them. Genes can be seen as parametric controls of pre-existing 'types,' directly related to the idea of design as re-design. 'Gene splicing' expresses the invention of new design 'species' through the novel combination of pre-existing types into new 'chromosomes.' 'Genetic engineering' allows the creation of entirely

new genes. 'Evolution' and 'breeding' correspond to random and non-random mutation with cumulative selection and change as practiced by designers.

To avoid confusion between the literal terms in their proper biological use and in the computational use which we are implementing, we have adopted the terms 'design gene' for the computational equivalent of a gene and 'design chromosome' for that of a chromosome. Each design gene can be represented by an extensional 'strip' of control and parameter information. Hence, a design chromosome is a strip of strips that specifies a designed artifact uniquely as a particular collection of instances of the pre-existing types. Of course, new types can be created as needed, as a separate operation.

4.2.1 Modes of operation

Disposing of such strips of different length and complexity allows the system to operate in several distinct modes:

1) Direct interaction/ Direct Mode. The designer (Natural Intelligent Processor, NIP) manipulates the diagrams directly, implicitly altering the underlying design gene structure. Changes in position or size modify the particular parameters in the strip only (i.e. the geometry only), while changes in number, kind, or correspondence modify the strip itself (i.e., the topology).

2) Indirect interaction/ Delegation Mode. The designer specifies the direction in which specific parameters are to be modified through the use of the vector-set, and delegates the actual modification to IPs, which carry out the task in the background. The designer is only interrupted when several conditions have been met within a specified 'time-memory window.'

3) Random mutation-Cumulative change/ Evolution Mode. Once the strip/diagram tree for an object has been created by decomposing the object into its underlying types, and while the designer is occupied with other tasks (or away from the DM), the system continues the search in 'evolution' mode. Small random changes are made to the values in the design gene strip, producing alternatives. Alternatives that do not conflict with

requirements survive and carry forward their design genes for further mutation, while the others become extinct. Unlike hill-climbing, this heuristic recognizes that progress can be made occasionally by "climbing" down the hill. It also eliminates backtracking, without eliminating the possibility of moving back to a previous position. Most importantly, it recognizes that often what appears to be a weak alternative can become, given sufficient time for refinement, the most viable alternative.

At any time, several alternatives co-exist and compete for resources (including computational resources). When the number of competing alternatives becomes too large for the available computational resources, the weakest members are eliminated.

4.2.2 Time-memory window

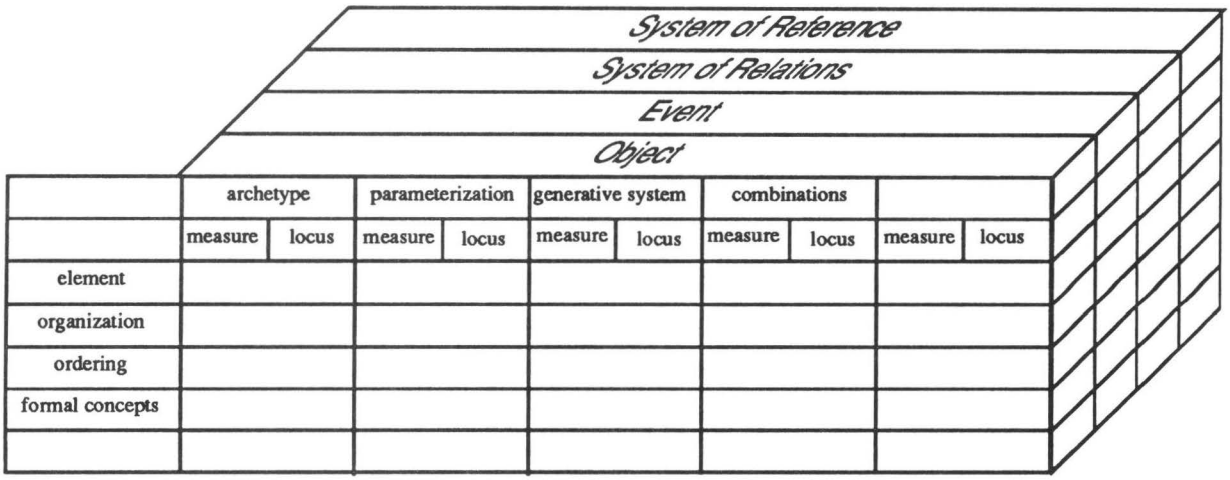
In order to support a non-serial model of the design process, we propose a time-memory 'window'. This concept implies that, at any given time, several alternatives to either the whole design or to sub-parts are 'alive' within the system and are being worked on by the IPs independently. If they reach a point when merging is possible (because certain common conditions are met) within the time-memory window, they are recognized and kept by the system. If not, they are removed and replaced by other alternatives, also modified in parallel and observed within this window.

In addition to determining the parametric definition of objects through concatenation and instantiation, the design genes are directly linked to the diagrams.

4.3 Parametric types

As we have seen, the Design Machine is implemented in a three-tiered architecture , a a layer of diagrams representing reconfigurable neural connections (additional sub-layers within this layer provide task subdivisions), a layer of genetic design, and a layer of types.

The layer of types consists of four fully parametric modules, corresponding to the four categories of possible designed artifacts (specifically, objects, sequences, relational systems, and reference systems), which we have called **designata**. For each designatum, we distinguish the following matrix:



This is to say that there are at least four levels of primitives that are of interest to the designer, as shown in the the rows of the matrix. For instance, an architectural design may be described as consisting of elements -such as columns and walls- organized on a grid, ordered symmetrically, and exploring the formal idea of having analogous plan and section relationships. The columns of the matrix show how the variation of these elements is structured. First, for each row-item we can recognize existing archetypes, such as different column types (doric, ionian, corinthian) in the element row, grid types (radial, central, orthogonal, triangular) in the organization row, ordering principles (symmetry, asymmetry, clustering) in the ordering row; and formal ideas (plan to section relations, overlap, transformation) in the formal idea row.

Each pre-existing type, and any that are added to the system through its use, can be parameterized, as implied by column two. Column three suggests that at this level the system knows how to systematically vary those parameters by employing a variety of generative systems. Finally, column four suggests that the system has provisions for keeping track of combinations of simpler types at each level.

One more distinction is necessary. For each item in the matrix, we recognize that variations can deal with relative or absolute values or, stated differently, with the locus or the measure of a primitive. Depending on the kind of artifact being designed, this distinction may apply to spatial relations (hence to topology vs. geometry or

topometry), to temporal relations (thus to chronology vs. chronometry), and finally to relations of energy (and thus to energy loci, such as equilibrium vs. energy measures, such as particular input values to a circuit).

The four broad categories that constitute the designata help ensure completeness; for each item in the matrix, however, we expect that the initial entries will be domain-dependent, at least initially. It is our intention that the organization of the system encourage cross-disciplinary interactions, of course, and thus no other partitioning of archetypes is attempted.

In terms of implementation this layer is built of fairly standard software modules, such as parametric solid modelers (PADL-2), object-oriented page description languages, etc., to which the DM acts as a front-end.

4.4 IPs learning

An important implementation concept is that of 'overlay/underlay', understood as the extraction/generation of new patterns from old ones through operations of comparison and distinction, addition/deletion, elaboration. These take place through the association of a 'vector-set ' (as defined in 4.1.4) with each diagram. This set of vectors provides control information to a neural net layer. Through initial training and subsequent learning, the network develops the ability to associate particular IPs with specific tasks, while general tasks are propagated throughout the entire network.

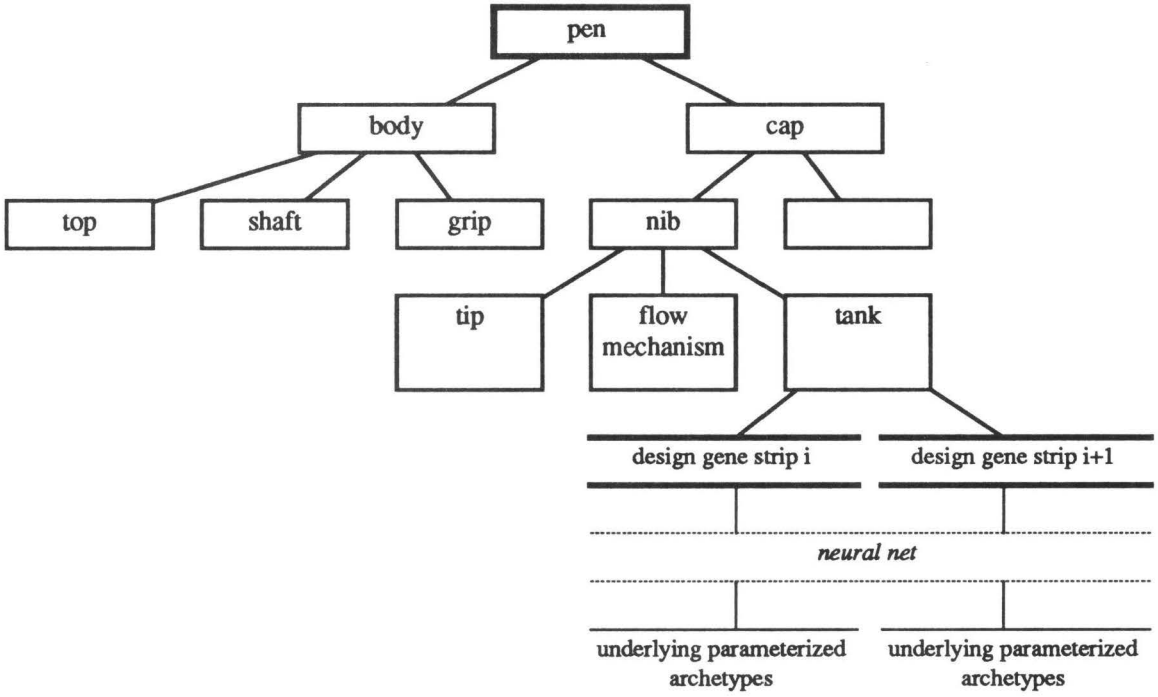
An IP receiving inputs that refer to its specialized knowledge does not distribute the information further along the same level, but rather to a lower level, the parametric variation /resolution level. In this intermediate level, neural nets are used to vary related components of the design while ensuring that corresponding dimensions and attributes do not conflict.

5. Functioning

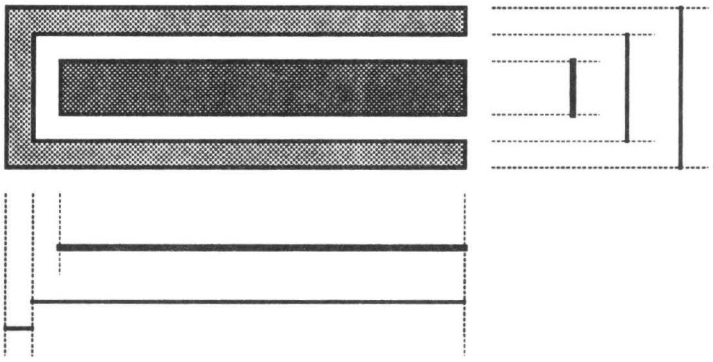
The activity of the Design Machine is based on the parametric interrelation of interactive diagrams and the manipulation of underlying 'design genes'. Diagrams as well as strips function parametrically to control the type layer. Let us explain some details.

5.1. 'Design Gene' coding strip

In order to explain this mechanism, let us examine the following example. The decomposition of an object will usually yield a structure of abstract types (as shown below).



The archetypal constructs that form the branches of this tree are parameterized and placed in the systems initial knowledge base, with provisions for modification, reparameterization, addition, deletion, and so on. Each such abstract type is controlled by a 'gene' strip of data, which we call a 'design gene'. The design gene strip specifies which dimensions can be modified and contains the particular values that determine each instance of this type. New types can be invented and parameterized by the designer, and design gene strips can be spliced to create new objects. The overall object is determined by a long strip of design genes created by merging all the strips from the lower levels.



name	position	parameter list	operation list	function list
tank	x1,y1,z1,R1,P1,Y1	(d h)	2, contained, body	$d_t = f(d_b)$, $h_t = f(h_b)$

name	position	parameter list	operation list	function list
body	x2,y2,z2,R2,P2,Y2	(d_n, d_{out}, h, m)	2, attached, cap	functions of cap

5.2. Specification of modes of operation

The modes of operation discussed in 4.2.1 can be specified as follows:

Direct Mode **D** -> **S** ; **D'** -> **S'**

Delegation Mode **D** + V_i -> $S(V_i)_i$
$S(V_i)_{i+1}$
$S(V_i)_{i+2}$ -> $D'(V_i)_{i+2}$
$S(V_i)_{i+3}$ -> $D'(V_i)_{i+3}$
.
.
$S(V_i)_{i+n}$ -> $D'(V_i)_{i+n}$

Evolution Mode **S** + **rm** $_i$ -> $S'_{rm\ i}$

$$if \quad \mathbf{S}'rm_i \neq \text{nil} \quad then$$

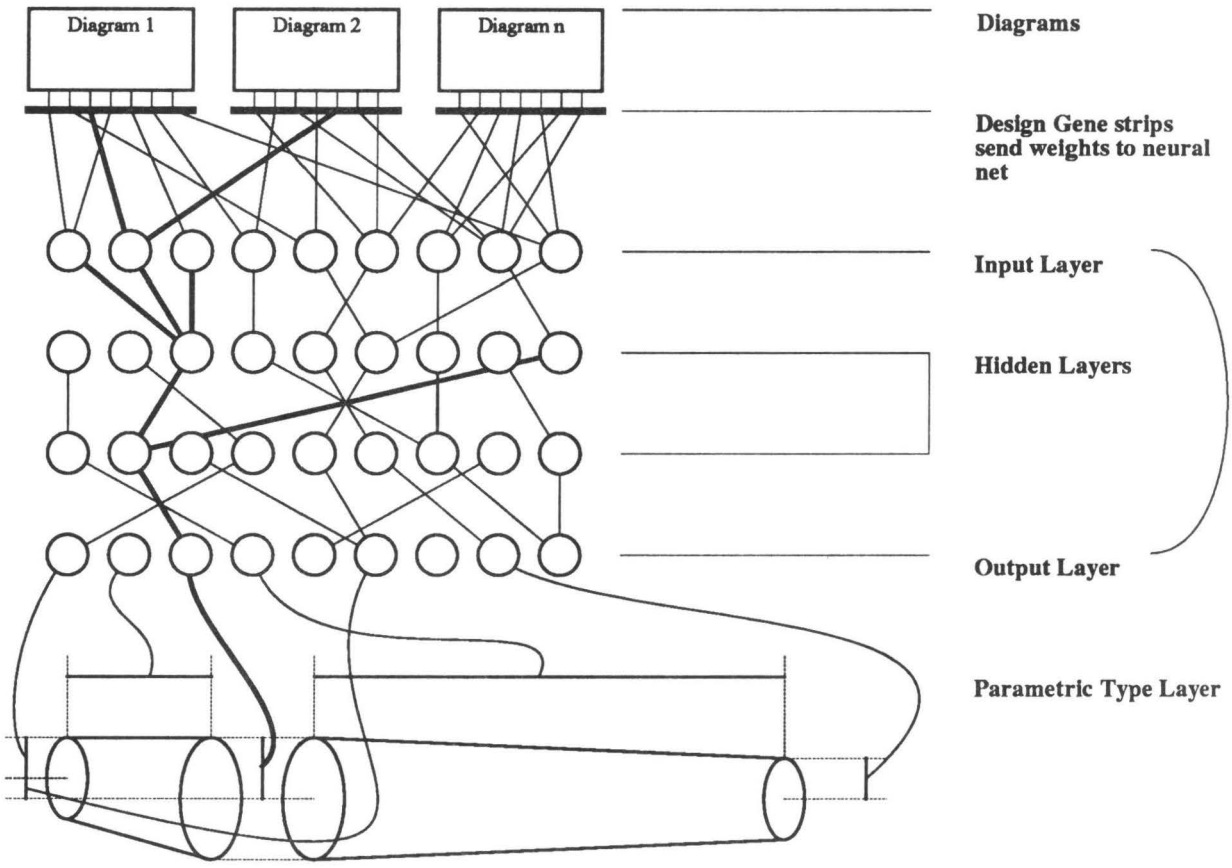
$$\mathbf{S}'rm_i + \mathbf{rm}_{i+1} \rightarrow \mathbf{S}''rm_i + rm_{i+1}$$

where **D** stands for diagram, **S** for design chromosome or 'strip', **V** for vector, and **rm** for random mutation.

This formalism explains how the design evolves from diagrams to strips and how changes in diagrams result in changes in the genetic strips.in each mode What is not immediately evident is the influence of the vector set on the generation of alternative design strips, and, correspondingly, diagrams. Vector sets are used here for implementation of associative memory neural nets. Finally, random mutation follows the genetic model. With the help of diagrams, we would like to offer additional explanations of the modes defined above.

5.3. Simulation

The diagram submitted below exemplifies the functioning of the machine. Important here is not the level of detail, but the structural suggestion.



As a global representation, this figure is meant to suggest how diagrams, strips, IPs implemented as neural nets, and types together generate designs. Details concerning the genetic strips, the way diagrams are applied as inputs into the IPs, and how types are generated can be noticed.

Finally, the manner in which the constitution of IPs as neural networks is supported is explained by the following discussion.

We have concentrated on the implementational platform upon which IPs will act. The actual implementation of IPs follows rather closely from the neural network model, in that activities such as specialization and delegation can be accomplished simply by training separate neural modules to make particular associations. Particular design team interactions (modeled as input vector values) directly require particular

connections ('knows' links and 'needs' links, for instance, modeled as the resulting neural output values). However, many problems remain.

The weighted inputs and outputs and the nonlinearity, as well as the network topology involving feedback and the possibility to employ various rules by which weights are adjusted (i.e., by which learning, self–organization, self–adaptation occur) require further clarification. We are evaluating which specific models of networks are more appropriate for the type of intelligence characteristic of design. We are also evaluating the resulting characteristics of the IPs network during training, which we see as equivalent to a transfer of experience. And we know that the massively parallel model of the IPs network in the Design Machine presupposes algorithms different from those that we were initially prepared to implement. As of this writing, we have not come up with reasonable answers to such questions.

References

1. Brown D.C. and Chandrasekaran, B. (1985): Expert Systems for a class of mechanical design activity, " in *Knowledge Engineering in Computer-Aided Design, Proceedings of the IFIP WG 5.2 Working Conference 1984 (Budapest),* Gero J.S. (ed.), North-Holland, Amsterdam, pp. 259-290.

2. Dawkins, Richard (1987): *The Blind Watchmaker. Why the evidence of evolution reveals a universe without design.* W. W. Norton & Co., New York/London.

3. Feldman, J.A., Fanty, M.A., Goddard, N. H. (1988): "Computing with Structured Neural Networks," in *Computer* **21**(3), pp. 91-103.

3. Mitchell, J.W., Liggett, R.S., Kvan, T. (1987): *The Art of Computer Graphics Programming.* Van Nostrand Rinehold Co., New York.

4. Nadin, M., Novak, M. (1987): "MIND: A Design Machine. Conceptual framework," in
Intelligent CAD Systems I, Theoretical and Methodological Aspects. P.J.W. ten Hagen and T. Tomiyama (eds.), Springer-Verlag, Berlin, pp. 146–170.

Design Transactions and Retrospective Planning

Tools for Conceptual Design

T. Takala

This paper will be available during the Workshop

Additional Contributions

Intelligent Representations of Geometric Knowledge for CAD

S. F. Bridge

INTELLIGENT REPRESENTATION OF GEOMETRIC KNOWLEDGE FOR CAD

Steven F.Bridge
Institut für Rechneranwendung in Planung und Konstruktion
University of Karlsruhe (TH)
Kaiserstraße 12
7500 Karlsruhe 1
West Germany
tel. +49 721 608 2470

ABSTRACT

If Artificial Intelligence is ever to enter the field of assisting Computer Aided Design, it must first overcome a shortcoming. This is the need to represent a certain type of information frequently required in mechanical engineering design, namely any information which is best represented by means of geometric entities in a real number space. This type of information includes the description of the geometry of engineering components as well as the graphs of functions describing aspects of the performance of the design which may require more than three dimensions.

An outline of a system capable of handling this type of information is described. This system is a generalisation of solid modelling and consists of extending the model space to n-Dimensions ($n > 0$), and allowing infinite objects. Centrally, a "Join" operator is defined which maps a set of points into their convex hull. This is augmented by a hierarchical method of representation based on "excursion" boxes which provide a variable level of detail for accessing the geometric models.

The need for flexibility arises from the observation that a collection of geometric entities (a model) may be interpreted as representing the (whole or partial) shape of a physical object or as the graph of a function relating several engineering variables. In the first case the usual solid modelling operations (for example, intersection detection) are required, whereas in the latter case such operations as function application or composition would appear more relevant. The proposed system is capable of both kinds of operations and therefore extends geometric modelling into the geometric representation of knowledge.

Finally, issues of data structure and algorithm choice encountered in the implementation of some of the central elements of the proposed representation system will be discussed.

KEYWORDS : Geometric Reasoning, Knowledge Representation, Geometric Modelling, Computer Aided Design, Artificial Intelligence.

1. INTRODUCTION

This paper investigates the kinds of information represented in mechanical engineering CAD systems and knowledge based deduction systems.

The processes of reasoning about geometry are of interest because, though performed with moderate ease by humans, they remain largely beyond the capabilities of computer systems in spite of solid modelling systems which do maintain informationally complete descriptions of shapes since the importance of geometric representations extends beyond the mere representation of shapes.

Thus Artificial Intelligence deduction systems are based on the systematic exploration of large search spaces of discrete information, whereas engineering design deals with quantities and concepts best modelled by continuously varying parameters which, in turn, can be represented as geometric entities in some space of appropriate dimension. Reconciling these two different representation systems in a productive manner is difficult.

Section 2 explains of why this is so, that is, why it is difficult for knowledge based design systems to take full advantage of the representational power of geometric modelling systems, and proposes a unified representation system for both kinds of information. The concept of a 'continuously varying parameter' is identified as a fundamental concept of which the geometric shape of an object and the graph of a functional relationship are particular cases. This concept of "geometric knowledge" and its particular properties are explored in some detail.

Section 3 then considers how such geometric knowledge could be represented in a generalised geometric modelling system including the corresponding mathematical basis. An outline of the proposed system is then described.

Next Section 4 considers some of the implementation issues surrounding the key operators of the proposed system, and Section 5 presents the conclusions.

2. GEOMETRIC KNOWLEDGE

This Section presents the concept of geometric knowledge. This is introduced gradually in the following Subsections by observing the nature of the representation systems used in Artificial Intelligence and Geometric Modelling Systems.

2.1 THE REPRESENTATION OF INFORMATION

Problems arise when attempting to apply Knowledge Based Systems to Engineering Design. These derive, in part, from a fundamental difference in the representation of information within these two disciplines as illustrated below :

2.1.1 Artificial Intelligence Systems

Knowledge based systems and Artificial Intelligence inference engines in particular are based on the fundamental concept of defining and exploring in a goal directed manner a large search space of distinct *discrete* states of "knowledge" eg the high level programming language PROLOG [1] uses the Horn Clause subset of first order predicate logic to define a search space which is then searched in a systematic manner by a depth first deduction system using clause unification and resolution.

The main advantage of this approach is the declarative formulation of "knowledge" which separates the formal definition of a piece of information from the way it is to be used and resides in the fact that a single definition may be used in many different ways, eg in PROLOG Horn Clauses are used to represent assertions (Horn Clauses with an empty conditional part) and rules (Horn Clauses with both a proper head and conditional part). The deduction machinery which forms part of the PROLOG system uses such information in different ways depending on the position and amount of variables contained in the current user query. If this contains variable(s), the PROLOG system will use the current database of Horn Clauses to find instantiations of those variables but if it contains no variables, the PROLOG system checks whether the user query is contained implicitly or explicitly in the current database.

2.1.2 Engineering Design Systems

Geometric modelling systems and engineering analysis packages are concerned with *continuously* varying parameters or quantities represented by floating point arithmetic.

In Geometric modelling, entities such as spline curves, free form surface patches and tri-cubic volume elements (as used in analytic solid modelling systems, see [2]) all involve continuously varying parameters in, one, two or three dimensions.

Furthermore, geometric entities such as points, lines, arcs, surface patches, can be used to represent non geometrical information eg the graph of a kinematic displacement function. However, this assumes that the geometric modelling system allows such an alternative interpretation to be placed on its models.

2.1.3 Summary

An important, clearly identifiable difference in the representation of information is therefore that Artificial Intelligence systems operate on discrete symbols whereas Engineering Computer Aided Design systems have to cope with the continuity of the real numbers.

2.2 RECONCILING THE REPRESENTATIONAL DIFFERENCES

The above differences in the nature of the represented information raise the following question : how can the discrete nature of an A.I. representation system be reconciled with the continuous nature of engineering quantities ?

Applying A.I. techniques to engineering problems necessarily involves partitioning continuous engineering quantities into finite sets of alternatives each of which is then represented symbolically within the A.I. system. In theory, this operation can either occur when the knowledge based A.I. system is built (and remain static during the use of the system) or, alternatively, can be postponed and only occur "dynamically" at run-time when the system is actually processing a particular problem.

2.2.1 Static Symbolic Representation of Engineering Quantities

Dividing the continuous engineering quantities into symbolic pieces when the knowledge based system is being built effectively hides their underlying continuity by providing the A.I. system with a discrete set of concepts or *specialists*. The latter may in fact use external modelling or analysis programs to perform their deductions before these are re-converted into symbolic form. The idea is similar to that of the "built in" predicates in, say, PROLOG for certain tasks (as, for example, floating point arithmetic) which would otherwise be exceedingly difficult to achieve directly within the confines of the A.I. system.

This approach has been used in an assembly planning system [3] which contains a deep knowledge kinematics "specialist" capable of deducing the instantaneous kinematics of a mechanism from its solid model description provided that the kinematic links have surface to surface contact (as opposed to line or point

contacts) and that the surfaces are either planar, cylindrical or spherical. Other kinds of contacts or surfaces would require providing the system with more complex "specialists".

The drawback of this method is that the engineering analysis or CAD system is only accessible if a suitable set of "specialists" was provided when the system was built. Furthermore, because the engineering system is **only** accessible through the set of provided specialists, it cannot be used to obtain information other than that which is "visible" through the "specialists". This raises the question whether providing such a predefined interface between the engineering system and the knowledge based system does not prevent the engineering system from being of use in circumstances which *were not anticipated* during the construction of the combined system; a weakness which would sensibly reduce the advantage of having an underlying engineering system in the first place.

Many of the knowledge based systems (for engineering design or otherwise) resolve this problem through selecting a sufficiently narrow field of expertise.

The great advantage of this approach is that the deduction machinery of the knowledge based system remains essentially unmodified and so its construction is simplified as an already available deduction system can be used.

2.2.2 Dynamic Symbolic Representation of Engineering Quantities

Building a set of appropriate "specialists" into a combined knowledge based system assumes that the process of dividing the continuous engineering quantities into a finite set of alternative symbolic concepts can be done in advance. The alternative approach is to delay this process until run-time where it could be tailored to suit the current needs, but, this would mean changing the A.I. deduction system in order to accommodate this additional task. In particular, it would have to take into account the additional properties of continuously varying parameters including the kinds of information that can be expressed through them and how these may be used, eg a simple description of the kinematics of a mechanism may consist of a series of graphs describing how the position (typically expressed in terms of translations along coordinate axes, and rotations about them) of each link changes as the mechanism moves through its operating range. Although these graphs are not "solid models" themselves, they are totally determined by the geometry of the links and their connections.

The process of deriving the kinematic description of the motion from the solid model (or, more often, from a simpler kinematic joint model) is one of numeric computation.

The designer will, however, not use all this very detailed numeric information, but only verify some critical values. If these are not satisfactory the mechanism is adjusted and the kinematic motion description recalculated. This iterative loop may then be repeated as often as necessary.

A geometry analysis system would calculate the kinematic displacement of each point on the mechanism up to an unnecessary degree of precision, but a purely symbolic system might not provide sufficient detail where the designer needs it.

An *ideal* system would be capable of quickly computing a representative (symbolic - discrete) approximation to the kinematic displacement curves and of refining these in the areas of interest to the designer to the extent that he requires (or his model permits), using more and more numeric computations.

It therefore appears that the continuously varying engineering quantities support a method of access in which they are partitioned by the designer into two categories, those that, in the current context, are of interest and those that are not. In particular a continuously varying engineering quantity can, in general, assume any of a infinite number of possible values; that is the set of possible values has an infinite number of elements and can therefore be partitioned in infinitely many ways. It is therefore not immediately evident that a fixed symbolic approximation whose the set of possible values has a finite number of elements (and therefore can only be partitioned in a finite number of different ways) will always be sufficient to capture the distinctions that are of importance to the designer.

Therefore an important question which arises in this context is how many different ways can a piece of information expressed in terms of continuously varying parameters be used ? Or, in other words, what is the ultimate declarative form of information based on continuously varying quantities ?

2.3 THE DECLARATIVE PRINCIPLE

The possibility of placing more than one interpretation on a geometric model has already been encountered. In particular, some engineering data, although not directly geometric, can nevertheless be represented by means of geometric entities. The graphs of continuous functions in a real number space (of appropriate

dimensions) all lie within this category. Such graphs are used throughout engineering to represent a large variety of information from kinematics, stress analysis, vibration frequencies and a host of other "performance" data. The process of design involves using such data and making deductions from them.

How many distinct interpretations can in fact be placed on a geometric representation of information and how these may be used will now be clarified.

First consider the interpretation of a geometric model in n-dimensional space (see example in Figure 1a) as the graph of a relationship or function. This perspective requires the n-dimensional space to be partitioned into two parts, namely a d-dimensional subspace equal to, or containing the domain of the function or relationship and the $(n-d)$-dimensional orthogonal space containing its range (Figure 1b). The process of applying the function or relationship to a point in the domain space can be constructed using such primitive modelling operations as orthogonal projection, set intersection and Cartesian product.

The starting position is a point in the d-dimensional domain space (Figure 1c). The Cartesian product of this point with the $(n-d)$-dimensional range space is added to the n-dimensional space containing the geometric representation of the graph of the function or relationship (Figure 1d). The intersection of these two geometric representations can then be calculated (Figure 1e). If the initial point was outside the domain of the function or relationship, then the outcome will be the empty set and there is no image of the point under that particular function. Otherwise, the intersection will yield some geometric entity that can be projected (orthogonally) onto the $(n-d)$-dimensional range space to yield the geometric representation of the image of the initial point (Figure 1f). If the graph is indeed the graph of a function, then the image is also a point, otherwise it may be a more complex entity such as, for example, an interval (or more precisely its geometric equivalent, a line segment).

In particular, if the graph of a function or relationship is known, its inverse can be computed just as easily simply by inverting the roles of the range and domain spaces. It follows that for a geometric model representing the graph of a function or relationship in an n-dimensional space, the **full range of possibilities** consists of **all** ways of partitioning the n-dimensions into two sets respectively forming the domain and range spaces. The image (in the range space) of a point in the domain space can then be constructed using the method outlined above.

In enumerating all possibilities, this approach also introduces two extreme cases, namely when either the range space or the domain space is empty while the other consists of the whole n-dimensional space. In the first case the function or relationship has no range (ie the range is empty) and in the second case it has no arguments (ie the domain is empty).

Even though neither case has any obvious interpretations, the following possibility nevertheless exists :

Consider the process of plotting the graph of a proper function (Figure 2a–d). This consists in the following three steps : enumerating some (in theory all) points in the domain (Figure 2a), computing the image of each of these points under the function (Figure 2b) and finally adding the domain-point, image-point pair (Figure 2c) to the graph of the function (Figure 2d). The extreme cases can now be interpreted as particular instances of this general schema.

A 'function' with no range is not a usual function in that applying it, in turn, to each point of its domain yields nothing. In fact, according to the above schema, doing so, amounts to **enumerating** the points in the domain. As in this case the domain is in fact equal to the 'graph' of the 'function' or 'relationship' the geometric model is effectively treated as the set of its constituent points, with the main emphasis resting on the points and not the set as a whole.

On the other hand, a 'function' with no domain yields its whole 'range' from nothingness as the empty (domain) set does not contain any points for which the 'image' might have to be computed. As in this case the range is in fact equal to the 'graph' of the 'function' or 'relationship' the geometric model is effectively treated as a **whole indivisible set**, and may thus be used as a single 'entity'. This is in fact the traditional interpretation that solid modelling systems place on their geometric models of solid objects.

If this interpretation of the extreme cases is accepted, then all possible interpretations of a geometric model representing some piece of information involving continuously varying parameters have been enumerated in a single theoretical paradigm.

An n-dimensional space containing a geometric model is partitioned into a sub-space of d dimensions, the 'Domain', and the 'Range' space of $(n-d)$ dimensions orthogonal to it. Then all possible interpretations of that **same** geometric model lie within one of the following alternative cases (see example in Figure 3) :

if
- $d = n$ — the geometric model is considered as an enumeration of its points, which may be used for some subsequent construction process. This could be termed an "inward looking" point of view.
- $0 < d < n$ — the geometric model is considered as a relationship or function linking some domain set in the Domain-space with a range set in the Range-space. This corresponds to viewing the geometric model as either relating a whole domain set and a whole range set or as relating an enumeration of points in the domain set to an enumeration of corresponding points in the range set.
- $d = 0$ — the geometric model is considered as a 'whole' in its own right, and may be subjected to some set theoretic operation or transformation mapping which involved all of its member points. This could thus be termed an "outward looking" point of view.

As this enumeration covers all possible interpretations, it effectively defines the ultimate limit of the declarative formulation of any piece of information. It can therefore be used as the reference point against which to compare existing systems.

Although solid modelling systems exhibit powerful geometric modelling facilities, these are always interpreted as representing the shape of some physical object and can therefore only be manipulated in a very limited way - namely as whole 'solid objects'.

On the other hand deduction systems based on Horn Clause logic permit an n-ary predicate to be queried with any combination of its n parameters instantiated in order to find the remaining unknown ones. The strict approach to first order logic however prevents the set of all parameter tuples from being treated as a single indivisible whole even though this is theoretically possible (see Chapter 1 Section 3 in Bell & Machover [4]).

Furthermore, a certain reduction in the "declarativeness" of the formulation of knowledge often accompanies the definition of "specialists" eg the PROLOG "built in" predicate for floating point multiplication. When this predicate is used with more than one uninstantiated variable an error is generated because the answer set is uncountably infinite : its elements cannot therefore be enumerated in a universally acceptable manner.

It should now be apparent that higher-dimensional continuously varying quantities in n-dimensional ($n > 0$) spaces form a central concept as the *generalised form of information* of which (three-dimensional) geometric shapes and the (n-dimensional) graphs of functional relationships are particular instances.

Therefore any piece of information expressed in terms of n-dimensional continuously varying quantities, together with the full spectrum of interpretations described above, must be considered as a piece of geometrically represented *knowledge*. Any engineering information which is expressed as a particular instance of such continuously varying quantities will hereafter be referred to as **geometric knowledge**.

Using the above kinematic example, it follows that the simple integration of an existing solid modelling CAD system and a knowledge based deduction system will prove inadequate for the representation of non-geometric, though geometry related, continuously varying data such as the description of the kinematics of a mechanism.

To sum up, the idea of a (higher-dimensional) continuously varying parameter is the fundamental concept which pervades much of engineering information and singling out the particular case of three-dimensional geometry does not make sense in a design context. Instead the concept of geometric knowledge, as defined above, should be used.

2.4 HIERARCHY OF REPRESENTATION

The previous Section clarified the principle of the declarative formulation of knowledge in the context of continuously varying quantities but it did not resolve the question of how the continuous nature of engineering quantities could be reconciled with the discrete nature of Artificial Intelligence deduction systems. An approach for doing so will now be described. This is based on two sets of ideas : excursion boxes [5] and qualitative representation systems [6].

2.4.1 Definition of Excursion Boxes

The concept of excursion boxes was first suggested by Medland [5] in an engineering context, where an excursion box is defined as the least orthogonal box, whose sides are parallel to the coordinate planes, enclosing a given CAD model of an engineering component. The dimensions of the excursion box of a particular CAD model of an engineering component correspond to the excursions of that component along the respective coordinate system axes.

In the context of a system for the representation of geometric knowledge the concept of an excursion box must be generalised to higher dimensions. In fact the excursion box of any n-dimensional continuously varying parameter can be constructed by taking the Cartesian product of (the convex hull of) its projections onto each of the coordinate axes of space. If the continuously varying parameter ranges over a connected region then the projections will always consist of a (convex) interval (or its degenerate form, a point) and the convex hull operation is not necessary.

To understand the role of excursion boxes in the representation of geometric knowledge an insight into their properties is needed.

2.4.2 Properties of Excursion Boxes

From the definition given above, excursion boxes can clearly be both represented and constructed within a geometric modeller. They are also convex geometric figures with a finite number of vertices (ie convex polytopes) as they are formed through the Cartesian product of intervals (Figure 4) which are instances of convex sets.

From a practical point of view, an excursion box has only limited storage requirements as it can be represented by just two points (which are such as to define a main diagonal of the excursion box), that is $2n$ numbers for an n-dimensional space. Thus storing an excursion box along side each continuously varying parameter is a feasible proposition.

The resulting intuitive appeal of 'simplicity' makes excursion boxes well suited to their dual application as a representation of uncertainty and as a building block for a hierarchical system of description.

Because an excursion box provides a simple, orthogonally decomposable, bounding envelope of a geometric model it can be interpreted as an approximate description of the model. In particular, the volume of space enclosed by the excursion box represents a region of spatial uncertainty as, in general, only some of it is actually occupied by the model.

The advantage of excursion boxes becomes apparent during intersection computations. If the excursion boxes are disjoint, so are their contents. The computational complexity of comparing two excursion boxes is very low as this only involves a sequence of comparisons on the $2n$ numbers defining each box. In particular, the comparisons can be made independently along each coordinate direction (where they reduce to comparing two intervals) and then combined to yield the relationship between the excursion boxes themselves. Thus when many items potentially intersect the excursion boxes can first be tested so as quickly to eliminate unnecessary intersection computations. The computational requirements of the problem of finding all intersecting pairs amongst a set of excursion boxes in n-dimensions has already been the subject of some research [7, 8] which has been summarised by Lee [9].

However, this advantage of excursion boxes is even greater when they form part of a hierarchy [5]. At the lowest level of which they are associated with individual geometric modelling primitives. Geometric models consisting of the aggregation of many primitives can themselves be represented by a larger excursion box which is obtained by finding the least enclosing box of the excursion boxes of the members of the geometric model. This process of representing sets of excursion boxes by a larger excursion box can be repeated until, at the top of the hierarchy, a single n-dimensional excursion box encloses all entities contained in the n-dimensional space.

Such a hierarchical organisation is well suited to engineering systems which usually consist of multiple levels of assemblies of sub-assemblies of piece parts, eg a motor vehicle is an assembly which contains the engine, itself an assembly containing, inter alia, the valve block, which in turn contains valve assemblies which finally consist of individual parts like the valve itself.

As this example shows, the hierarchical organisation converts the problem of identifying a particular part from the problem of finding the part directly, in one selection operation, from a very large set, to one where the set of possibilities is gradually narrowed down through many selection operations each from a very much smaller set. Of the two methods, the latter tends to be quicker in yielding the answer.

Indeed the process of starting with the large excursion boxes near the top of the hierarchy and gradually investigating the smaller ones lower down the hierarchy may be continued beyond the "base" of the hierarchy. At the "base" of the hierarchy each excursion box just contains a single geometric modelling primitive. Such an n-dimensional excursion box can be systematically decomposed in a manner similar to Octrees, into a collection of 2^n smaller excursion boxes. These may in turn be used to divide up the contents of the original box by intersecting each with the original content to yield a partition of the original content into smaller pieces. Finally the excursion box of each of these pieces is computed. The resulting set of excursion boxes effectively extends the hierarchy beyond its original base. Through repeated application of this algorithm, it can, in theory, be extended indefinitely.

However, this will eventually result in two kinds of excursion boxes those which straddle the boundary, and those which only contain interior points of the geometric model and thus need not be considered any further. This leads to an increase in the complexity of the required algorithms.

Nevertheless, this approach would lead to a system which operates exclusively on excursion boxes which it can decompose as often as required.

In this context an excursion box should be considered as a region of spatial uncertainty which has associated with it the means of decreasing this uncertainty should this be required : in the case of an excursion box which contains many entities, the reduction of uncertainty is achieved by switching attention directly to the excursion boxes of the contained entities, and in the case of an excursion box containing only a single geometric primitive simply apply the method suggested above.

Note that as a representation of spatial uncertainty, excursion boxes provide a single formalism for both the issue of uncertainty of dimension and uncertainty about the complexity or nature of the shape [10]. In both cases, replacing the single large excursion box by its content of smaller excursion boxes will provide a more accurate description of both its dimensions and emerging shape.

The principles underlying the idea of excursion boxes as outlined above are not entirely new as the idea of a hierarchical representation system has antecedents in both CAD and AI.

For example consider the area of graphical representation of two-dimensional curves in the form of Strip Trees [11] or the Artificial Intelligence system "Mercator" for the representation of (two-dimensional) spatial knowledge [10].

More directly, the concept of excursion boxes is related to the representation of objects by means of Octrees. However, the key difference is that Octrees consists of an enumeration of spatial occupancy based on the decomposition of *space* into regularly shaped subspaces, whereas the excursion boxes are positioned relative to the *object*. Furthermore plain Octrees only involve decomposition cells which, at the leaves of the Octree, only store one piece of information, namely whether they are full or empty. Polytrees [12], a generalisation of Octrees, allow in addition slightly more complex leaf cells, such cells containing either a vertex, an edge or a surface. Finally, excursion boxes at the lowest level of their hierarchy contain still more complex information in the form of a whole geometric model.

2.4.3 Qualitative Systems

Qualitative reasoning is a branch of Artificial Intelligence whose aim is to develop models of human common sense reasoning applied to the physical world [13, 14, 15], in particular, for use in future knowledge based systems.

A qualitative reasoning method should be able to handle imprecise descriptions of physical systems such as weakly specified functional relationships or non numeric specification of parameter values. It should further be able to cope with systems which do not have closed form solutions and identify unexpected points of qualitative change while not requiring extensive computational resources [15].

These are all properties which also make qualitative reasoning interesting for design. In particular the ability to both represent and obtain some kind of deductions from descriptions which are but outlines of a system should be valuable for those stages of the design process where complete descriptions are not yet available and decisions must nevertheless be made.

The proposed qualitative structural and behavioural descriptions only require simple deduction processes based on the ability to create and match simple expressions, rather than on arithmetic operations or symbolic integration. Partial knowledge can be represented in the sense that both parameter values and functional relationships are only constrained to lie in qualitatively defined classes rather than being described explicitly.

As qualitative reasoning appears to be a relatively new field of AI slightly different approaches are still being explored in parallel [13, 14, 15, 6]. For the current discussion the QSIM system [15, 6] will be used in preference to the others because it concentrates directly on sets of qualitative relationships and does not represent either devices [13] or processes [14] directly. As a result the QSIM system retains a greater level of generality by avoiding built in assumptions concerning the domain of application.

In the QSIM system a continuously variable parameter is considered as a continuous function (for example of time) and is represented in terms of a list of qualitative states. Each qualitative state is the combination of two intervals and a symbolic quantity. Of the two intervals, one represents the domain of the independent variable (typically an interval of time) and the other the corresponding range traversed by the continuously variable parameter. The additional symbolic quantity indicates the direction of change (monotonically increasing, constant or monotonically decreasing) of the value of the continuously varying parameter with respect to the independent variable over these intervals.

The objective of the qualitative reasoning system is to derive this description of the behaviour of the dependent continuously variable parameters solely from a qualitative model. Such a qualitative model consists of a set of qualitative relationships of which there are three principal types : the arithmetic ($X = Y + Z$), derivative ($Y = dX/dt$) and proportional ($Y = M^+(X)$ meaning that Y is a strictly increasing function of X). Note that the proportionality relationship, in particular, provides a very much weaker level of description than would be possible with a numerical or analytic approach to the solution of (differential) equations.

Because of the qualitative nature of the modelling, a given qualitative description will characterise different physical realisations of the same device [13]. This feature contributes to making qualitative representation methods attractive for design problems as it means that alternative avenues of design development can be evaluated at an early stage without first having to pursue each to a (potentially wasted) level of detail.

The converse is also true; that is, a given system can have different qualitative representations [15] which reflect different levels of detail, and thus may not all be appropriate to the current problem.

In the case of the QSIM system, it has been proved that its qualitative deduction algorithm will produce every actual behaviour of the device being modelled. However, because of its local point of view, it will also predict spurious behaviours which will not be exhibited by any mechanism satisfying the structural description [6].

Returning to the qualitative representation scheme it is important to observe that the Cartesian product of the domain and range intervals of a qualitative state forms the excursion box of the graph of that continuously variable parameter. Thus, when a piece of geometric knowledge is interpreted as a function or relationship, its excursion box on its own is not sufficient as it does not contain any indication of what the function inside the excursion box might look like. Therefore the excursion box based representation must be augmented by some additional pieces of information.

Although the representation scheme of the QSIM [15] system appears to be a good starting point it nevertheless needs to be extended to take into account the following additional phenomena : the effects of higher dimensions, non-functional relationships, and a hierarchical structure.

Functions of more than one variable (whose graphs would be thus have to be represented in three or higher dimensional spaces) could be represented qualitatively by the Cartesian product of the qualitative descriptions as perceived along each coordinate axis. That is, for a function of n variables, its domain would be decomposed into excursion boxes. Each excursion box would be such that the function were monotonically increasing, decreasing or constant (over the corresponding range) with respect to one of its variables independently of how the others were chosen within the part of its domain enclosed by the excursion box. That this system of description is sufficient for representing any continuously varying function (in higher dimensions) clearly follows from the fact that the vocabulary of qualitative reasoning is sufficient for functions of one variable.

Note that one higher dimensional phenomenon which has to be taken into account is that in a qualitative description of a function of n variables the excursion box of the domain will also, in general, include points which are either not in the domain of the function, or are in the domain of the function, but not in the part to which the current qualitative description applies. The excursion box of the domain will thus have to be supplemented by the geometric model of the actual part of the domain under consideration.

The approach described so far clearly only applies to single valued functions of several variables. However, there are many pieces of geometric knowledge which may be considered as mappings but which are not single valued and therefore are not 'functions' according the mathematical definition. These can be further

distinguished into those mappings which, for a point in the domain, yield a discrete set of corresponding image points and those that yield a line segment, ie an interval of values. The former can be considered as a finite collection of 'proper' functions (and the qualitative descriptions obtained accordingly) while the latter must be represented by a pair of 'proper' functions, one describing the upper bound and the other the lower bound of the solution interval(s).

Again this approach only applies to functions which yield a one-dimensional image for each antecedent in the domain. However, mappings which yield an r-dimensional image may be treated as a set of r functions which each yield a one-dimensional image, and thus be represented by r qualitative descriptions obtained accordingly, whereas the r-dimensional image as a whole would be represented by its excursion box.

The vocabulary of the qualitative descriptions must be extended in order to support hierarchical descriptions in parallel with the excursion box hierarchy. The additional concepts required to describe qualitatively larger sections of functional behaviour as enclosed in larger excursion boxes towards the top of the excursion box hierarchy are *non-monotonic but generally increasing*, *non-monotonic but generally decreasing* and *constant/indeterminate*. Obtaining these descriptions amounts to performing a kind of filtering operation in which small scale variation is 'ignored' in favour of the global trend, similar in nature to the 'human' description of a line, drawn free hand, as 'straight' rather than irregular and undulating. It is further possible to qualify the amount of non-monotonicity by augmenting the description by an excursion box whose size describes the minimum amount of change required in order to be certain of observing an increase (or decrease) in the function value. The size of this additional excursion box would also help decide when a given level of qualitative description is too crude and needs to be refined.

2.4.4 Geometric Knowledge Representation

Section 2.3 described how a same piece of geometric knowledge can be considered as a set of points or as a entity as a whole, or as some kind of function or relationship. When a piece of geometric knowledge is considered as a whole or as an enumerated set of points, excursion boxes can be used to qualitatively represent its spatial distribution. When a piece of geometric knowledge is considered as some kind of functional relationship a qualitative description can be used in addition to the excursion box.

Therefore excursion boxes and the proposed qualitative representation system are complementary as, together, they cover the full spectrum of the declarative representation of a piece of geometric knowledge.

In particular the excursion boxes act as a finite number of "containers" of pieces of geometric knowledge which may be refined as needed. Furthermore, the excursion boxes may be set up so as to reflect some particular property. As at any time there are only a finite number of excursion boxes, these form a crude but finite approximation to a piece of geometric knowledge while reflecting the distribution of some property.

This is in fact the key to the proposed mechanism which would enable geometric knowledge to be used in conjunction with AI deduction systems. The AI deduction system can simply use the current finite set of excursion boxes in the same way that it previously used purely symbolic quantities, but with the following important difference. When a given representation in terms of excursion boxes is not sufficiently accurate for the current purpose, this can both be detected (for example when the region of overlap between two excursion boxes is larger than the currently acceptable level of uncertainty) and the situation remedied by decomposing the offending excursion boxes into more detailed ones. Therefore this is a possible mechanism for a more "intelligent" integration of AI systems and geometric knowledge.

Thus the combination of excursion boxes and qualitative representations seems to yield a system of representation(s) which can be tailored dynamically through demand driven evaluation to the level of detail required for resolving the current problem while supporting the full spectrum of declarative representation of a piece of geometric knowledge.

2.5 DIRECT COMPARISON OF REPRESENTATION SYSTEMS

In order to obtain an insight into the configuration of a system for the representation of geometric knowledge, it is instructive to evaluate the knowledge representation approach taken in Artificial Intelligence from to the point of view of geometric solid modelling and vice versa. As it is the *approaches* to knowledge representation that are being compared it is not really necessary to select a pair of specific systems. Nevertheless, the comparison will be loosely based on the Horn Clause subset of first order predicate calculus (as found in a knowledge programming language like PROLOG [1]) and a conventional (CSG or analytic) Solid Modelling system.

2.5.1 Horn Clauses as Geometric Modeller

When considering the knowledge representation scheme of Horn Clauses for the purposes of solid modelling the following correspondences would intuitively appear to be the "most natural". An n-ary predicate would correspond to an n-dimensional space of say Euclidean geometry. The assertion of a fact using that predicate (that is a Horn Clause with an empty conditional part) would be interpreted as a specifying a point. This point would have as coordinates the predicate's n-parameters and exist in the space named after and defined by the predicate (Figure 5a). A rule (that is a Horn Clause whose conditional part consists of the conjunction of further predicates) would specify a geometric figure which can best be described as the analogue of a linear space in Euclidean geometry. Indeed if the predicate contains a variable as its nth argument then the geometric equivalent is a point whose nth coordinate is unknown and may thus range over the whole set of real numbers and therefore lie anywhere on the line (that is, the one-dimensional linear space) which is parallel to the nth coordinate axis (Figure 5b). Similarly, if the predicate at the head of the Horn Clause contains two variables, the point specified by it will only be constrained to lie in a plane (that is a two-dimensional linear space) defined by the two coordinate axes corresponding to the variables arity position in the predicate. Executing a query (that is specifying a Horn Clause only consisting of a conditional part) will look for (all) "points" known to the system which "intersect" the "linear space" specified by the query (Figure 5c).

The most unusual feature from the solid modelling point of view is that the whole system works without a metric (that is the "points" only have a tuple of symbolic names as coordinates), is discrete and works in any finite dimension (whereas solid modelling operates exclusively in three dimensions).

The fact that many of the "points" are not expressed explicitly but are re-deduced by the rules when needed is similar to solid modelling where only a few key points are stored for each volume patch in an analytic solid modelling system [2].

2.5.2 Geometric Modelling as Knowledge Representation

Conversely, when considering the knowledge representation scheme of solid modelling for the purposes of logical deduction the following correspondences would intuitively appear to be the "most natural" — these are essentially the same correspondences as used above, except that now the flow of the analogy will proceed in the opposite direction. In particular, an n-ary predicate was considered to be the analogue of an n-dimensional space. As the solid modeller operates exclusively in three-dimensional space, there would appear to be only a single three-ary predicate. Similarly, the assertion of a fact was considered as the analogue of a point within the predicate–space. Therefore, each point in the solid modeller corresponds to the assertion of a fact under the single three-ary predicate. However, a whole (non-degenerate) solid model consists of an infinite number of points. Therefore a (non-degenerate) solid model would be the analogue of asserting an infinite number of facts in an AI deduction system. Finally, in the above analogy, 'rules' involved the representation of whole linear spaces and correlating points within these across different predicates of possibly different arity. Such 'rules' cannot be represented at all in the solid modelling system. There are two reasons for this : firstly the solid modelling system is restricted to a single three-ary predicate (ie three-dimensional space) and thus cannot express relationships between different predicates, ie spaces (of possibly different dimensionality). Secondly solid modelling is restricted to finite objects and thus cannot represent a whole, non trivial, linear space.

Viewed as a deduction mechanism solid modelling systems reflect these representational features. The central deduction operation consists of computing the intersection of solid models. Interpreting a solid model as the assertion of (an infinite number of) facts this "deduction" process consists of matching sets of "facts" (essentially by computing the intersections of the corresponding solid models). In the absence of (the ability to represent) "rules" this is the only form of deduction that can be performed by a solid modelling system.

However, the ability of asserting an infinite number of "facts" under a predicate (that is, to represent a solid model containing an infinite number of points) has a very profound consequence : the approach to deduction consisting of systematically trying out all explicit combinations will never terminate, unless some additional mechanism (related in some way to the mechanism of intersection computations of solid modelling) is added to deal with them.

2.5.3 Comparison Summary

The criteria for a knowledge representation system capable of handling both kinds of information can now be easily obtained as the union of the properties of the systems dealing with each kind individually. In fact

these as well as the ideas from the preceding Sections can now be combined into an informal statement of the objectives of a system for the geometric representation of knowledge. Such a system should :

1. be able to represent geometric knowledge, that is, continuously varying parameters in n-dimensions. This involves expanding the capabilities of solid modelling to allow multiple, higher-dimensional spaces with individual models which may contain a proper linear space (and thus extend infinitely in some direction).
2. with the help of excursion boxes be capable of performing the appropriate kind of "deductions" on these entities, in a way analogous to the operations performed by AI or solid modelling systems.

3. REPRESENTING GEOMETRIC KNOWLEDGE

3.1 MODELLING GEOMETRIC KNOWLEDGE

The principal ideas presented in the previous Section are that :

1. Engineering design has to deal with quantities and concepts best represented by continuously varying parameters in n-dimensions, that is with geometric knowledge.
2. Geometric Knowledge supports many different interpretations (as a set of points, as the graph of some kind of relationship or as a whole entity) all of which have been enumerated in the framework of a single theoretical paradigm.
3. In order make geometric knowledge compatible with AI deduction systems it must support a hierarchical representation scheme, each level of which consists of a finite description. This hierarchy overcomes the limitations of a 'static' approximation into a finite number of symbolic concepts by being extendable where and when needed so as to provide a more refined picture in a consistent manner. Furthermore the hierarchy promotes efficient access to information as a fairly crude symbolic representation may, in many cases, be already sufficient. However, the hierarchy must consistently provide some information about the more detailed information contained at the lower levels independently of the interpretation that the user may currently wish to use. The proposed method of achieving this goal is to use excursion boxes together with an (augmented form of) qualitative description.

In spite of the range of interpretations that may be placed on a piece of geometric knowledge it can be simply represented as a geometric model. This supports the complete spectrum of interpretations as : firstly, the geometric model can be treated as a whole. Secondly, the interpretation as the graph of a relationship or function can be supported as the application of the relationship or function to a point in the domain can be computed by a sequence of geometric modelling operations as described in Section 2.3. Finally, the interpretation as a set of points can be obtained through the excursion box hierarchy. As the lower levels of the excursion box hierarchy are explored, more and more points will be obtained within the geometric model.

Thus a suitably augmented form of geometric modeller should be capable of representing items of geometric knowledge. The link between a geometric model and its continuously varying parameter is simply that the geometric model represents the set of all possible values that the continuously varying parameter can adopt. The detailed requirements for such a modeller can be now be obtained as the union of the properties of the systems described in Section 2.5. The description of a scheme for such a generalised modeller now follows.

3.2 GENERALISED POLYHEDRAL SOLID MODELLER

The proposed system for the representation of geometric knowledge can be considered as a generalisation of a polyhedral solid modeller. This involves two generalising steps.

The first generalisation is to represent polyhedral models in an n-dimensional space. This is needed in situations where the geometric model does not represent the shape of an engineering component, but should be interpreted as the graph of a function, or some other non-shape related information. In this context consider the following examples.

Only two-dimensions are required to represent the graph of a function of a single variable, for example how the distance along a given path changes with time during a displacement. On the other hand, the representation of the shape of a solid object requires a three-dimensions space. The displaced positions of a moving component, for example those of a link within a mechanism observed throughout its operating cycle, consists of four-dimensional data : three dimensions are needed to represent the component in some position in space and a fourth dimension to measure how far the motion has progressed. In this case the first three dimensions correspond to the physical three-dimensional space (using units of distance) and the fourth to either time (using units of time) or the fraction of the whole motion that has already been completed (in

which case no units of measurement apply). Finally, a solid object has six degrees of freedom under rigid motion transforms : three translational and three rotational. It follows that any given displacement can be described using six values corresponding to the magnitudes of the changes in those six degrees of freedom. Therefore any given displacement can be represented as a point with those six values as coordinates in a six-dimensional space. This space has been called the *configuration space* [16, 17]. In a given context all points in this six-dimensional configuration space do not correspond to practical displacements because they might bring the mobile object into collision with obstacles in its vicinity. Constructing the six-dimensional images (in the configuration space) of three-dimensional physical obstacles and computing a configuration space trajectory which avoids them is being used as a means of planning valid trajectories for robot manipulator arms [16, 17].

All these examples illustrate the use of continuously varying quantities in spaces of different dimensionality to represent information (geometric knowledge) that is clearly relevant to engineering design. Therefore in order to cope with the full spectrum of such engineering information, the modelling system must be capable of representing geometric entities in any finite dimensional space with the same ease that a logical deduction system exhibits with respect to the use of n-ary predicates (see Section 2.5).

The second generalisation is the extension to the modelling of infinite objects rather than being limited to finite ones. This feature is needed for two reasons. The first is that it effectively allows universal quantification over continuously varying parameters, which, in turn, is needed in order to express "rules" similar to those in systems based on Horn Clause logic. The second reason is that the concept of (points at) infinity is needed in qualitative reasoning systems to qualitatively describe a quantity which does not have an upper (or lower) bound by means of an open interval with one, or more, infinite end points.

The two mathematical principles involved in the above generalisations will now be explored. The polyhedral n-dimensional modeller is essentially based on the theory of "Join Geometries" [18], whereas the extension to infinite objects requires some results from "General Topology" [19].

3.3 JOIN BASED GEOMETRY

The main problem which needs to be addressed is to obtain a precise definition of how continuously varying parameters (in n-dimensions) can be represented in a computer. The problem centres on the fact that a (non-trivial) set of these will contain an infinite amount of elements. As the computer storage capacity is finite, these cannot be represented explicitly. This leaves the question of whether a *finite* representation of an *infinite* number of elements is a well founded proposition. That the answer to this important question is 'yes' is implicitly assumed by all existing solid modelling systems but does not seem to have been justified explicitly. (The main emphasis of earlier work was primarily placed on deriving a mathematical definition which formally captures the properties intuitively associated with 'solid' objects [20, 21]). Fortunately the answer to this important question is indeed 'yes' and takes the form of the Krein-Milman Theorem which can be found in the branch of mathematics which is concerned with convex sets.

At this stage it is worth pointing out that a modern, coordinate- and (in this context more importantly) dimension-free approach to the subject of convex sets can be obtained through the systematic exploitation of the "Join" (or convex hull) operator which gives rise to the concept of "Join Geometry" [18]. Further details, and more formal definitions of some of the terminology can be found in Appendix A.

A continuously varying parameter, interpreted as a geometric figure (or set of points) in some n-dimensional space, is said to be convex, when it has the following property. If any two of its points are chosen and joined by a line segment, then all the points on line segment already belong to that geometric figure; ie no part of the line segment will lie "outside" the original figure. The convex hull of a given geometric figure, F, (again regarded as a set of points) is the smallest convex geometric figure containing F.

The Krein-Milman theorem then states that (in a Euclidean Space) a closed and bounded convex set **is** the closure of the 'Join' (or convex hull) of its extreme points [22, 23, 24, 18]. In other words, a (bounded and convex) polyhedron, for example, can be described **completely and unambiguously** by the set of its extreme points (which are more often known as vertices). If the set of extreme points is known then the convex hull operation applied to this set of extreme points will yield the desired description of a continuously varying parameter, and may be interpreted as a (convex) geometric figure in some n-dimensional space. Therefore only the actual set of extreme points need be stored. If this set is finite, then a finite description of a continuously varying parameter ranging over an infinity of points is possible.

Examples of convex geometric figures with a finite number of extreme points (vertices) are a single point (that is a zero-dimensional space), intervals on the one-dimensional real number line (note that intervals are of practical interest as they are used in the representation scheme of the QSIM qualitative reasoning system), convex polygons in the two-dimensional plane and convex polyhedra in three-dimensional space.

The general term for these and their higher-dimensional equivalents is that of "polytope" [18, p133]. (A simple example of a convex geometric figure which has an infinite number of extreme points can be found in the form of a solid sphere where every point on its surface (infinitely many of them) is an extreme point).
The next Section will extend the representation scheme to infinite entities, a process which requires some changes to the modelling space itself.

3.4 COMPACTIFICATION OF SPACE

The problem with the above approach is that the continuously varying parameter may only range over a bounded region of space (that is, a region of finite size). To overcome this restriction and allow continuously varying parameters to range over an unbounded region, a more general formulation of the Krein-Milman Theorem involving the topological concept of compactness is needed as a first step. Note that in a Euclidean n-dimensional space any closed and bounded set is compact [19 p144].
The more general formulation of the Krein-Milman Theorem is that each compact, non-empty convex set is identically equal to the closed join (or convex hull) of its set of extreme points [25 p174].
One of the 'simplest' unbounded sets over which a continuously varying parameter may range is the set $\mathbb{R}$ of all real numbers which can be also be interpreted geometrically as an unending straight line. This set is not compact. However, it can be made compact through the addition of two points, $+\infty$ and $-\infty$, and yield the (compact) extended real number line [25 p71, 19 p149]. Note that the real number line (extended or not) is also a convex set, and in the case of the extended real line can be described finitely as the closed convex hull of its two extreme points $+\infty$ and $-\infty$.
Thus for continuously varying parameters in one dimension the problem has been solved. The extension to higher dimensions is however fairly simple using the Tychonoff Theorem [19 p143, 25 p83].
The Tychonoff Theorem states that the Cartesian product of a collection of compact (topological) spaces is compact (relative to the product topology).
Consider a higher, say n-dimensional, space obtained from the Cartesian product of n (compact) extended real number lines. This space will also be compact (Tychonoff Theorem) and therefore representable as the closed convex hull of its extreme points (Krein-Milman Theorem).
The theoretic foundation for the representation scheme of continuously varying parameters in n-dimensions having been successfully established, the next Section considers the kinds of primitive operations that will be needed.

3.5 OUTLINE OF MODELLING OPERATIONS

This Section presents an initial overview of the primitive operations that will be needed for the practical representation of geometric knowledge.
For the representation scheme proposed above to work the 'Join' (or convex hull) operation must be built into the system. The Join operator must be able to compute the convex hull of a finite list of points, in n-dimensional space where each point may be either finite or at infinity.
For non-convex shapes the representation scheme must be extended. A very simple option is to include the ability to form lists of the primitive convex polytopes. If such a list has the appropriate mathematical properties (for example, non-overlapping, connected, bounded, closed, regular, (semi-analytic) and three-dimensional [20, 21]) it may be interpreted as a non-convex (polyhedral) solid object. The ability to form lists of polytopes (which may or may not have these properties) is not usually found in solid modellers, but is one of the most fundamental representational abilities of Artificial Intelligence deductions systems and should therefore be retained.
Like most solid modellers the system should support the basic set theoretic operations. Of these intersection and set theoretic difference are the primitives as the union operation consists of forming a list of the intersections and differences. The intersection operation on convex polytopes is closed as the intersection of convex polytopes is itself a convex polytope [18 p525]. The set theoretic operations of difference and union do not always yield a convex polytope. However they will always result in a finite list of convex polytopes.
The representation scheme can support unbounded entities which includes the important category of linear spaces. Examples of linear spaces are straight lines (one dimensional linear space), planes (two dimensional), three dimensional space, etc. These kinds of entities are sometimes found in CAD systems in the form of construction lines which, although not part of any CAD model, do provide help with the geometric construction process. The kinds of operations that are most relevant would be a linear hull operator which would create the least linear space containing its operand and a linear complement which would create the linear space orthogonal to (the linear hull) of its operand. In an n-dimensional space the linear complement

to an m-dimensional linear space is the $(n-m)$-dimensional linear space which is orthogonal to it while containing an identified point.

The remaining operations are related to defining spaces within which continuously varying parameters will reside. Apart from the facility for creating new n-dimensional spaces, operators are needed for moving information between them : these are simply the Cartesian product (for combining lower dimensional information into a higher dimensional space) and orthogonal projection (for reducing higher dimensional information into a lower dimensional space) operations.

This completes the list of the more interesting or unusual operations required by the proposed geometric knowledge representation scheme. Many other 'house keeping' commands will be needed in an actual system for such tasks as editing or copying existing data and interfacing with the permanent storage media of the computer.

4. IMPLEMENTATION ISSUES

This Section considers some of the issues surrounding the implementation of a system for the representation of geometric knowledge.

The sequence of implementation of a system of the scale and relative complexity of that proposed for the representation of geometric knowledge must to some extent reflect the interdependence of its constituent ideas. The generalisation to n-dimensions of a polyhedral solid modeller based on the Join (or convex hull) operator is a relatively independent element of the proposed system in the sense of not being heavily dependent on other parts while the centre of the representation system, the excursion box hierarchy, is not really feasible without it. This generalised modeller is, however, also one of the most complex parts of the proposed system.

Therefore, a possible implementation path begins with the generalised modelling system, augments it by the excursion box hierarchy, then introduces the qualitative descriptions and finally considers (future) extensions to non-polyhedral models.

However, owing to the amount of software that would be required, an implementation of a complete system is not yet available. The implementation is currently restricted to demonstrating the feasibility of the principal operators for the generalised, n-dimensional modelling system. The detailed implementation issues will therefore concentrate on the generalised polyhedral modeller and are presented in two parts, the data structures and the algorithms.

4.1 DATA STRUCTURES

The main data structures of the generalised modelling system are associated with the representation of points, of convex polytopes and of lists of these. Each of these data structures must be augmented by its excursion box and qualitative description.

4.1.1 Representation Of Points

The representation of points has to take into account that the dimension of the modelling spaces is user selectable and that some of the points may be at infinity.

Modelling spaces of different dimensionality imply that different numbers of coordinate values will, in general, have to be stored for the points. As a fixed size storage scheme would have to provide sufficient space for the highest dimensional case this approach would be rather wasteful of computer storage capacity. It therefore seems preferable to opt for the dynamic allocation of storage capacity for point coordinate tuples in spite of the additional programming complexity that this entails.

The data structure used for the representation of points must be able to represent both finite points and points at infinity. Two schemes may be considered, the extended Cartesian and homogeneous coordinates.

In the extended Cartesian coordinate system the coordinate tuple of a point $P = (x_1, \ldots, x_n)$ in n-space must be capable of containing either a real number or one of the values $-\infty$ and $+\infty$ at each of the x_i $(1 \leq i \leq n)$.

With homogeneous coordinates a point P in n-dimensional space is represented through $(n+1)$ coordinates $(x_{h1}, \ldots, x_{hn}, \lambda)$ which are related to the original n coordinates $(x_1, \ldots, x_n)$ by $x_i = x_{hi}/\lambda$ for $1 \leq i \leq n$. When λ is non-zero the corresponding point is finite, when λ is zero and at least one of the x_i is non-zero then the the point is at infinity — effectively the n coordinates $(x_{h1}, \ldots, x_{hn})$ represent a vector rooted at the origin and pointing towards it. When all the $(n+1)$ homogeneous coordinates are zero then the point is indeterminate — a condition which should be avoided.

Neither of the two representation systems is ideal for use in the proposed generalised modelling system : the representation of parallel linear spaces is complex with homogeneous coordinates, and the representation of

linear spaces at an arbitrary finite angle to the coordinate system is virtually impossible with the extended Cartesian system. Nevertheless homogenous coordinates are adequate for geometric modelling and extended Cartesian coordinates are well suited to the representation of excursion boxes.

4.1.2 Representation Of Polytopes

A polytope is the closed convex hull (join) of its finite set of extreme points (vertices) each of which may either be finite or at infinity.

Two properties of polytopes are that a polytope is completely determined by its set of extreme points (vertices) and that its boundary is itself composed of polytopes of next lower dimensionality. For example, the boundary of a polyhedron, a three-dimensional polytope, consists of two-dimensional faces, each of which is a polygon, that is a two-dimensional polytope.

The representation system for an n-dimensional polytope P should reflect these properties by providing access to the boundary components of P by dimension, that is to the list of $(n-1)$-dimensional hyper-faces of P, the list of $(n-2)$-dimensional hyper-edges of P, ..., the list of 1-dimensional edges of P, and the list of 0-dimensional vertices of P.

An important aspect to be taken into account in the design of a data structure for the representation of polytopes is the notion of recursion such that the data structure representing a polytope which does form part of the boundary of some larger polytope should be the same as that of a polytope which does not. The advantage of such a recursive representation system is that it enables the use of recursive algorithms and thus reduces the complexity of implementing a system capable of operating in n-dimensions.

The topology of a polytope is essentially similar that of a hyper-graph. Whereas an ordinary graph is defined as a set of vertices and a set of edges (ie pairs of vertices) a hyper-graph presents an altogether more complex structure. This results from the 'edges' being generalised from joining two points to 'hyper-edges' joining many points. In a polytope P these hyper-edges correspond to the boundary components of P which can be further be classified by dimension (a property which is the consequence of the vertices being defined in a space with coordinate geometry).

4.2 ALGORITHMS

This Section considers some of the geometric modelling algorithms for the generalised solid modeller which is needed for the representation of geometric knowledge.

The principal primitive operations are Join (or convex hull), Cartesian product and intersection. Most of the remaining modelling operations such as set theoretic union can in fact be expressed in terms of these primitives, or reduce to simple data structure manipulations such as, for example, building a list of polytopes. The following Sections consider each of these algorithms in turn.

4.2.1 Convex Hull (Join) Algorithm

The Join (or convex hull) operation takes a set S of p points in a n-dimensional space and finds the smallest convex set which contains S (See Appendix A for the definition of terms such as convex).

A large number of convex hull algorithms exist [9]. Most concentrate on the computation of two-dimensional convex hulls and while some consider the three-dimensional problem, those for higher, n-dimensional cases are very much rarer.

Two- and three-dimensional convex hull problems can be computed in $O(p \log p)$ time where p is the number of points [26]. In both cases these times are optimal, though, in the two-dimensional case they can be improved when additional information about the input points is known [27]. The computational complexity of convex hull problems in higher dimensions is more difficult to determine as it is depends on the properties of n-dimensional polytopes. However, these (and in particular the upper bounds on the number of components of a polytope) are but incompletely understood (see [28 Chapter 10]).

For example, the divide and conquer algorithm for two- and three-dimensional convex hulls described in [26] is based on the property that the number of edges of the convex hull with p extreme points (vertices) is at most linear in p. However, in higher dimensions there are polytopes whose number of (hyper) edges is $O(p^2)$ in the number p of vertices [28 Section 10.1] so that a generalisation of that particular method to higher dimensions is not possible within $O(p \log p)$ time.

In fact only one algorithm for computing convex hulls in n-dimensions was found [29]. No bounds on its computational complexity were given other than an empirical (favourable) comparison with the 'brute force' method. Nevertheless, an optimal algorithm for computing convex hulls in n-dimensional space where n must be an even number has been reported [9].

Therefore, for optimal performance, a mixture of algorithms should be used : for two- and three-dimensional problems use the corresponding optimal algorithms, for n-dimensional problems (where $n > 3$), if n is even

use the algorithm referred to in [9] and if n is odd use the algorithm reported in [29]. This approach would ensure best known performance at the cost of programming complexity.

Alternatively, because of its generality, the algorithm described in [29] could be used throughout obtaining simplicity of implementation possibly at the expense of computational efficiency.

In their paper [29] Chand and Kapur both supply a proof of correctness and state their convex hull (Join) algorithm as implemented in a FORTRAN program. This algorithm computes the convex hull of a finite set of finite points in n-dimensional space through a process of 'gift wrapping' in which the $(n-1)$-dimensional faces of the polytope are generated one at a time.

The algorithm relies on the property that exactly two ($(n-1)$-dimensional) *faces* of an n-dimensional polytope P intersect along each ($(n-2)$-dimensional) *edge* of P. If a face F and edge E belonging to F are known, the second face F' can be found by conceptually rotating the plane of the known face F about edge E through an appropriate angle. This process is similar to that of wrapping up a gift in a sheet of paper. The new face F' will be bounded by new edges (at least $n-1$ of them). This process of gift wrapping can therefore be repeated until all edges are contained in exactly two faces at which stage the polytope has been fully determined. A mechanism for determining a starting face is also provided.

This 'gift wrapping' algorithm must be extended to include the case where some (or all) of the points are at infinity. For this purpose, points, both finite or at infinity, are represented by their homogeneous coordinates. The extension of the convex hull (Join) algorithm to points at infinity is not too complex but does introduce a number of additional cases which must all be taken into account (for details on this or any of the following algorithms see [30]).

4.2.2 Cartesian Product Algorithm

The Cartesian product operation takes a list of geometric models and constructs a new geometric model, corresponding to their Cartesian product, within a specified higher dimensional space whose dimension must equal the sum of the dimensions of the operand models' spaces.

The Cartesian product command also requires further information which specifies how the dimensions of each operand space map onto the dimensions of the product space. This offers sufficient flexibility for any possible permutation of coordinates in the resulting product model.

The algorithm operates through building up the boundary components of the n-dimensional result of the Cartesian product starting with the zero-dimensional vertices and finishing with the n-dimensional result polytope itself. The main complexity of the Cartesian product operation resides in the number of concepts and entities involved and in particular the sometimes complex manner in which these have to be accessed.

4.2.3 Set Theoretic Algorithms

The set theoretic operations used in geometric modelling are intersection, set-difference and union. These are defined in the usual manner as follows. Considering two geometric models M_1 and M_2, their intersection $M_{M_1 \cap M_2}$ is the set of points common to both M_1 and M_2, their union $M_{M_1 \cup M_2}$ is the set of points which occur in at least one of M_1, M_2 and the set theoretic difference $M_{M_1 - M_2}$ is the set of points which belong to M_1 but do not belong to M_2.

In the case of an intersection or set theoretic difference operation an inexpensive excursion box based comparison can be used for early identification of operands which do not intersect (by virtue of their excursion boxes being disjoint) and thus avoid computationally more expensive algorithms.

The intersection $J_{J_1 \cap J_2}$ and set difference $J_{J_1 - J_2}$ of two polytopes J_1 and J_2 of same dimension d can be computed through repeated intersections with appropriate half spaces. Though not necessarily very efficient such an algorithm has the advantage of both simplicity and independence of dimension.

The set theoretic union of polytopes follows by observing that the union of two polytopes J_1 and J_2 is equal to the following list of $J_1 - J_2$, $J_2 - J_1$ and $J_1 \cap J_2$.

4.2.4 Orthogonal Projection

The orthogonal projection operation takes a list of geometric models and constructs a new geometric model, corresponding to their orthogonally projected image, in a given lower-dimensional space.

The orthogonal projection operation also requires further information to specify how the dimensions of the space containing the operands map onto to those of the projection space. This offers sufficient flexibility for any possible permutation of coordinates in the resulting projected model.

The property that the Join (convex hull) of the projected extreme points (vertices) of a polytope is equal to the projection of the initial polytope itself can be used to obtain a simple implementation of the orthogonal projection algorithm : to project a polytope, only project its extreme points and then compute their convex hull. Note that although this process is simple to implement (given the availability of the Join operation)

it is not necessarily as efficient as a general algorithm for projecting j-dimensional polytopes contained in d-dimensional spaces into some p-dimensional space ($j \leq d$ and $p < d$).

This essentially completes the description of the algorithms, which were mainly devised to enable the early development of a prototype system for the representation of geometric knowledge. The implementation of such a prototype system is currently in progress.

5. CONCLUSIONS

The application of Artificial Intelligence techniques to CAD proposed in this paper does not involve applying a knowledge based system to a particular design problem. Instead the fundamental concepts used in a simple inference engine (PROLOG) are examined and applied directly to CAD so as to obtain an augmented geometric modelling system.

The main result of this research is to propose a powerful and flexible representation scheme for geometric information or knowledge in a Computer Aided Design context.

The paper began by identifying a representational need in Artificial Intelligence with respect to a type of information frequently required in mechanical engineering design. The information in question is the representation of, and interaction with, geometric shapes of solid objects. The fundamental nature of the difficulty was traced to the use of *continuously varying* parameters to represent such information within the context of deduction systems operating on *discrete* representations. The rest of the paper then systematically developed and refined a method of overcoming this problem.

The statement of the problem was generalised from the restricted context of the representation of the shape of solid objects to any information which is best represented by means of continuously varying parameters (such as geometric entities) in a real number space. In particular, this includes any continuous relationship linking engineering quantities.

At this stage the central idea was presented for elevating a geometric model to the (geometric) representation of a piece of knowledge which involves a continuously varying parameter. This simply consists of a theoretical framework for enumerating all possible modes of interpretation or use of such a geometric model and coincidentally also defines the ultimate limit to the principle of declarative knowledge representation.

In order to link AI deduction systems and geometric knowledge, an additional hierarchical representational element was introduced by borrowing the concept of excursion boxes and a representation scheme from qualitative reasoning. The excursion boxes provide a finite systematic way of representing geometric uncertainty or approximate (geometric) information and were shown be closely related to a one-dimensional system of representation of Qualitative Reasoning. The required generalisations of the original qualitative representation scheme were also considered and found to originate from three sources, namely the generalisation to higher dimensions, the generalisation to mappings and relationships which are not necessarily functions according to the strict mathematical definition and the introduction of a hierarchy.

A direct comparison between the representation systems of the Horn Clause subset of Predicate Logic and solid modelling demonstrated a representational requirement for multiple spaces of any finite dimension (as analogues of multiple predicates of any finite arity) and the ability to represent unbounded quantities as well as finite ones.

This led to the postulation of a generalised polyhedral modeller as the means of representing information involving continuously varying parameters. A theoretic justification for the correctness of the representational requirements that this involves was sought and found in the form of a pair of Theorems in Join Geometry and point set Topology.

The more practical issues surrounding the implementation of the proposed system were then considered through the discussion of the kinds of data structures that would be capable of dealing such new representational requirements as points at infinity and convex polyhedra (polytopes) in higher dimensions as well as of the principal (geometric) modelling algorithms.

ACKNOWLEDGEMENTS

This research was sponsored up to 1987 by Dowty Rotol ltd, Gloucester, England under the Dowty Rotol Research Assistantship at Imperial College, London, England. It is currently being continued under an 1851 Research Fellowship at the Institut für Rechneranwendung in Planung und Konstruktion at the University of Karlsruhe (T.H.), Karlsruhe, West Germany.

REFERENCES

[1] W.F.Clocksin, C.S.Mellish, "Programming in Prolog," Springer Verlag, 1981.

[2] M.Casale, E.L.Stanton, "An Overview of Analytic Solid Modelling," *IEEE Computer Graphics and Applications*, 45–56, February 1985.

[3] D.E.Jakopac, "A Multi-Level, Model Based Planner for Mechanical Assembly," PhD Thesis, Northwestern University, August 1985.

[4] J.Bell, M.Machover, "A Course in Mathematical Logic," North Holland, 1986.

[5] A.J.Medland, "The Computer Based Design Process," Kogan Page, 1986.

[6] B.Kuipers, "Qualitative Simulation of Mechanisms," Technical Memo MIT/LCS/TM–274, Massachusetts Institute of Technology, USA, April 1985.

[7] H.Edelsbrunner, H.A.Maurer, "On the Intersection of Orthogonal Objects," *Information Processing Letters*, **v13**, **n4,5**, 177–181, End 1981.

[8] H.Six, D.Wood, "Counting and Reporting Intersections of d-Ranges," *IEEE Transactions on Computers*, **vC-31**, **n3**, 181–187, March 1982.

[9] D.T.Lee, F.Preparata, "Computational Geometry : a Survey," *IEEE Trans. Computers*, **vC-33**, **n12**, 1072–1101, December 1984.

[10] E.Davis, "The Mercator Representation of Spatial Knowledge," in *Proceedings International Joint Conference on Artificial Intelligence*, 295–301 (Vol.1), 1983.

[11] D.H.Ballard, "Strip Trees: A Hierarchical Representation for Curves," *Communications of the ACM*, **v24**, **n5**, 310–321, May 1981.

[12] I.B.Carlbom, I.Chakravarty, D.Vanderschel, "A Hierarchical Data Structure for Representing the Spatial Decomposition of 3D Objects," *IEEE Computer Graphics and Applications*, 24–31, April 1985.

[13] J.DeKleer, J.S.Brown, "A Qualitative Physics Based on Confluences," *Artificial Intelligence*, **v24**, 7–83, 1984.

[14] K.Forbus, "Qualitative Process Theory," *Artificial Intelligence*, **v24**, 85–168, 1984.

[15] B.Kuipers, "Commonsense Reasoning about Causality : Deriving Behaviour from Structure," *Artificial Intelligence*, **v24**, 169–204, December 1984.

[16] T.Lozano-Perez, "Spatial Planning: a Configuration Space Approach," *IEEE Trans. Computers*, **vC-32**, **n2**, 108–120, February 1983.

[17] T.Lozano-Perez, "Automatic Planning of Manipulator Transfer Movements," *IEEE Trans. Sys. Man Cyb.*, **v11**, **n10**, 681–689, October 1982.

[18] W.Prenowitz, J.Jantosciak, "Join Geometries : a Theory of Convex Sets and Linear Geometry," Springer Verlag, 1979.

[19] J.L.Kelly, "General Topology," Springer Verlag, 1955.

[20] A.A.G.Requicha, "Representations of Rigid Solid Objects," in *Lecture Notes in Computer Science 89 : Computer Aided Design, Modelling, Systems Engineering, C.A.D. Systems.*, J.Encarnacao(Ed), Springer Verlag, 2–78, September 1980.

[21] A.A.G.Requicha, "Representations for Rigid Solids : Theory, Methods and Systems.," *ACM Computing Surveys*, **v12**, **n4**, 437–464, December 1980.

[22] H.G.Eggleston, "Convexity," Cambridge University Press, 1969.

[23] C.Berge, "Topological Spaces," Oliver and Boyd, 1963.

[24] R.T.Rockafellar, "Convex Analysis," Princeton Univ. Press, 1972.

[25] C.Berge, "Espaces Topologiques : Fonctions Multivoques," Dunod, Paris, 1966.

[26] F.P.Preparata, S.J.Hong, "Convex Hulls of Finite Sets of Points in 2 and 3 Dimensions," *Communications of the ACM*, **v20**, **n2**, 87–93, February 1977.

[27] D.T.Lee, "On Finding the Convex Hull of a Simple Polygon," *Int. J. Computer and Information Sciences*, **v12**, **n2**, 87–98, April 1983.

[28] B.Grunbaum, "Convex Polytopes," J.Wiley and Sons — Interscience Publishers, 1967.

[29] D.R.Chand, S.S.Kapur, "An Algorithm for Convex Polytopes," *Journal of the Association for Computing Machinery*, **v17**, **n1**, 78–86, January 1970.

[30] S.F.Bridge, "Aspects of the Geometric Representation of Knowledge for Computer Aided Design," Ph.D. Thesis, Imperial College of Science and Technology, London, 1988.

[31] F.Valentine, "Convex Sets," McGraw Hill, 1964.

[32] T.W.Gamelin, R.E.Greene, "Introduction to Topology," Saunders College Publishing, 1983.

APPENDIX A
CONVEX SET THEORY AND TOPOLOGY

This Appendix introduces some mathematical definitions from Convex Set Theory in Section 1 and from Topology in Section 2.

A.1 CONVEX SET THEORY

This Section introduces some of the definitions of Convex Set theory. The starting point is the introduction of a convex Join operator which allows the definition of a Convex Set. The definition of the Interior, Closure and Boundary of a Convex Set is then followed by the Krein-Milman Theorem.

Definition of Join Operator. *The open convex Join of two distinct points p_1 and p_2 (denoted by $p_1 \cdot p_2$) is defined as the set of points, p, such that $p = \lambda p_1 + (1-\lambda)p_2$ with $0 < \lambda < 1$*

$$p_1 \cdot p_2 \equiv \{q \mid \exists \lambda \in I\!R \;\; q = \lambda p_1 + (1-\lambda)p_2 \;\; \text{where} \;\; 0 < \lambda < 1\}$$

Note that $p_1 \cdot p_2$ does not include either of the points p_1 or p_2; in Euclidean space, the Join of p_1 and p_2 is simply the open line segment between p_1 and p_2. Also the convex Join of a point p with itself is equal to itself by definition : $p \cdot p \equiv p$.
The idea of convex Join can be extended to several points : the 'open convex Join' of n distinct points $p_1, p_2, \ldots, p_n$, is simply defined as the set of points, p, such that $p = \lambda_1 p_1 + \lambda_2 p_2 + \ldots + \lambda_n p_n$ where $\lambda_1 + \lambda_2 + \ldots + \lambda_n = 1$.

Definition of Convex Hull. *The open convex Join of a set S, $S \cdot S$, is defined as the union of all open convex Joins of all subsets of S. For any set S, the set $S \cdot S$ is called the convex hull of S.*

The concept of convex Join can also be formulated in a more abstract manner, without appealing to notions of arithmetic or dimension, in terms of an operator (on points or sets of points) which fulfils certain axioms. Such a treatment is given in [18] and forms the basis of *Join Geometries.*

Definition of Join Geometry. *A Join Geometry consists of a pair $(J, \cdot)$ where J is a set and '$\cdot$' an operation which maps points of J into points of J and satisfies axioms J1–J7.*

Let a, b, c and d be points of J.
- J1 (Existence law). $a \cdot b \neq \emptyset$.
- J2 (Commutative law). $a \cdot b = b \cdot a$.
- J3 (Associative law). $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- J4 (Idempotent law). $a \cdot a = a$.
- J5 (Existence law). $a/b \neq \emptyset$.
- J6 (Four Term Transposition law). If $a/b \cap c/d \neq \emptyset$ then $a \cdot d \cap b \cdot c \neq \emptyset$.
- J7 (Idempotent law). $a/a = a$.

Note that / is the extension operator derived from '$\cdot$' and is defined as follows.

Definition of the Extension Operation. *The extension of a point a from a point b of J, denoted by a/b, is the set of all points x of J which satisfy $a \subset b \cdot x$.*

Definition of Convex Set. *A set, S, of points is convex if and only if (iff) for all points p, q of S, $p \cdot q$ is included in S.*

$$S \text{ Convex} \iff \forall p, q \in S \;\; p \cdot q \subset S$$

Definition of Polytope. *A polytope is a convex set which is the convex hull of a finite, nonempty set.*

Definition of Interior Point. *A point, p, is an interior point of a convex set S iff for all points x of S, there exists at least one point y of S such that p is an element of $x \cdot y$.*

The set of all such points p is called the 'interior' of set S, denoted by $\mathtt{I}(S)$.

Definition of Contact Point. *A point p is a contact point of a convex set S iff there exists a point x of S such that $p \cdot x$ is included in S.*

The set of all such points p is called the 'closure' of set S, denoted by $\mathtt{Cl}(S)$. A set S is closed iff $S = \mathtt{Cl}(S)$.

Definition of Boundary Point. *A point p is a boundary point of a convex set S iff p belongs to the closure of S, but p does not belong to the interior of S : $p \in \mathtt{B}(S)$ iff $p \in \mathtt{Cl}(S)$, $p \notin \mathtt{I}(S)$.*

The set of such points p is called the boundary of S and is denoted by $\mathtt{B}(S)$.
In effect a convex set is said to be closed when it contains all of its boundary points and is said to be open if it does not contain any of its boundary points.

Definition of Boundedness. *A set S is said to be bounded if the least upper bound (supremum) of the distance between any two of its points (ie its diameter) is finite.*

Definition of Extreme Point. *A point, p, of a convex set S is said to be an 'extreme point' of S iff p is never contained in the open convex Join of two or more distinct points of S.*

$$p \text{ extreme} \iff \text{if } p \in x \cdot y \text{ then } x = y$$

The Krein-Milman Theorem. *In a Euclidean space a closed convex set is the closure of the convex hull of its extreme points (Theorem 13 in [22]). More generally, a convex, compact, non-empty set is equal to the convex closure of its extreme points (p174, [25]).*

Proofs can be found on page 24 in [22], page 167 in [23], Section 18 in [24], Sections 13.10 to 13.15 in [18] and part XI in [31]. (Compactness is defined in Section 2 of Appendix A).

A.2 TOPOLOGY

This Section introduces a few definitions from Topology. In particular the definitions of Topological Space and Compactness are followed by the compactification of the Real Number line and the Tychonoff Theorem.

Definition of Topological Space. *A nonempty set S together with a subcollection $\mathtt{T} \subset 2^S$ is a topological space if the following three axioms are satisfied :*

A1 $S \in \mathtt{T}$, $\emptyset \in \mathtt{T}$.
A2 $T_1, T_2 \in \mathtt{T} \Rightarrow T_1 \cap T_2 \in \mathtt{T}$.
A3 $\forall \mu \in M,\ T_\mu \in \mathtt{T} \Rightarrow \bigcup \{T_\mu \mid \mu \in M\} \in \mathtt{T}$.

Note that 2^S denotes the power set of S, that is the set of all subsets of S, and M is an arbitrary set of integer subscripts.
Thus T is a set of subsets of S which satisfies the three conditions : both the empty set and S itself are included in T, the intersection of any two members of T is a member of T and the union of the members of any subset of T is also a member of T. The set T is called a topology of S. The elements of T are called 'open sets'. For further information see [19] Chapter 1, [32] Chapter 2.

Definition of Compactness. *A set S is compact if every open cover of S has a finite subcover. That is, S is compact if, whenever $\{U_\mu\}_{\mu \in M}$ is a family of open sets whose union is S, then there are finitely many of the U_μ's whose union is S.*

Note that a collection of sets $\{U_\mu\}_{\mu \in M}$ is a cover of some set S if S is included in the union of the sets in the collection, that is $S \subset \bigcup_{\mu \in M} \{U_\mu\}$

Theorem. *Any closed set S contained in a compact set R is itself compact [Section IV.6 in both 23 and 25].*

Example. *Compactification of real Number line.*

The space (line) of all real numbers, $\mathbb{R}$, is not bounded as there is no largest (or smallest) real number. Although $\mathbb{R}$ can be covered with open intervals of unit length, no finite subcovering of $\mathbb{R}$ can be found. $\mathbb{R}$ is therefore not compact.
The real number space $\mathbb{R}$ can be extended by adding two elements called $+\infty$ and $-\infty$ and, for any $\lambda \in \mathbb{R}$, stipulating that

$$+\infty = (-\infty)(-\infty) = (+\infty)(+\infty) = \lambda + (+\infty) = \lambda - (-\infty)$$
$$-\infty = (-\infty)(+\infty) = (+\infty)(-\infty) = \lambda + (-\infty) = \lambda - (+\infty)$$
$$+(-\infty) = -(+\infty) = -\infty$$
$$-\infty < \lambda < +\infty$$

And further for any $\lambda > 0$:

$$(+\infty)\lambda = \lambda(+\infty) = (-\lambda)(-\infty) = (-\infty)(-\lambda) = +\infty$$
$$\frac{\lambda}{-\infty} = \frac{\lambda}{+\infty} = 0$$

The extended real number space is denoted by $\hat{\mathbb{R}}$. For further information see Section I.1 in both [23] and [25].
Now $(\hat{\mathbb{R}}, \mathtt{T})$ is a topological space. The collection of open sets T contains the open intervals of $\mathbb{R}$, the union of $\{+\infty\}$ with an open interval of $\mathbb{R}$ of the form $]\lambda, +\infty[$ and the union of $\{-\infty\}$ with an open interval of $\mathbb{R}$ of the form $]-\infty, \lambda[$.

Theorem. *$\hat{I\!R}$ is a compact space.*

For further information and the proof of compactness of $\hat{I\!R}$ see Section IV.6 in both [23] and [25].

Tychonoff Theorem. *Any product of compact spaces is compact.*

Proofs may be found in [32] p102 and [19] p143.

In particular the n-dimensional extended real number space $\hat{I\!R}^n = \overbrace{\hat{I\!R} \times \ldots \times \hat{I\!R}}^{n \text{ times}}$ is therefore compact. It follows that any closed, non-empty convex set S in $\hat{I\!R}^n$ is the convex hull of its extreme points in $\hat{I\!R}^n$. If some or all of the extreme points are at infinity (that is, have infinite coordinates) then S is of infinite size (that is, the least upper bound on its diameter is $+\infty$).

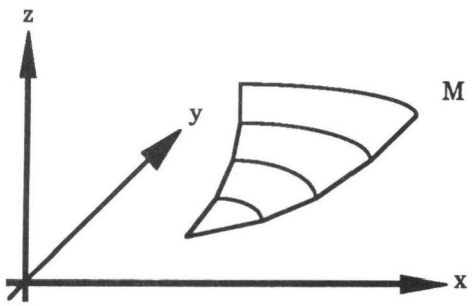


Figure 1(a) : A Geometric Model M in n-D space (n = 3).

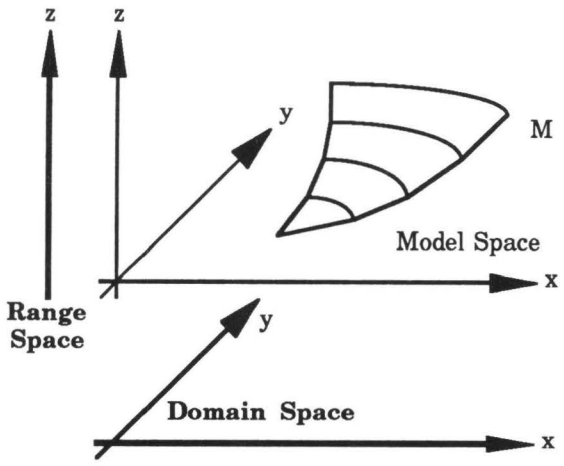


Figure 1(b) : Domain and Range spaces

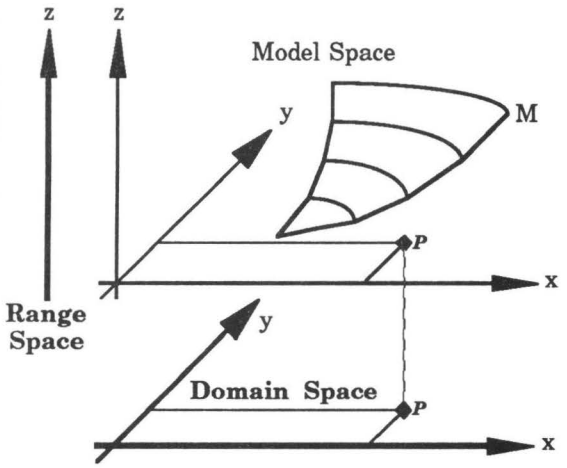


Figure 1(c) : Function application starting point *P* in Domain Space

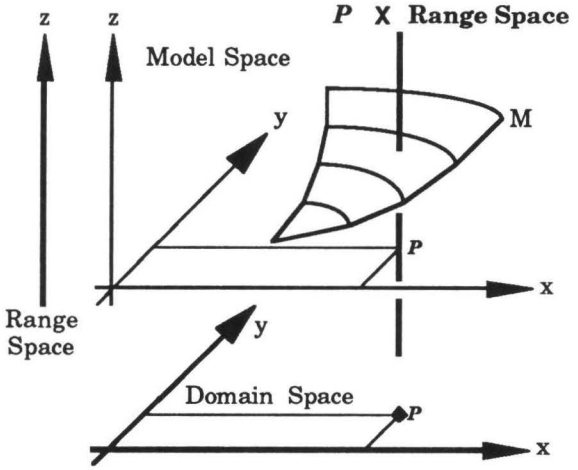


Figure 1(d) : Function application : Cartesian product of *P* and Range Space into Model Space

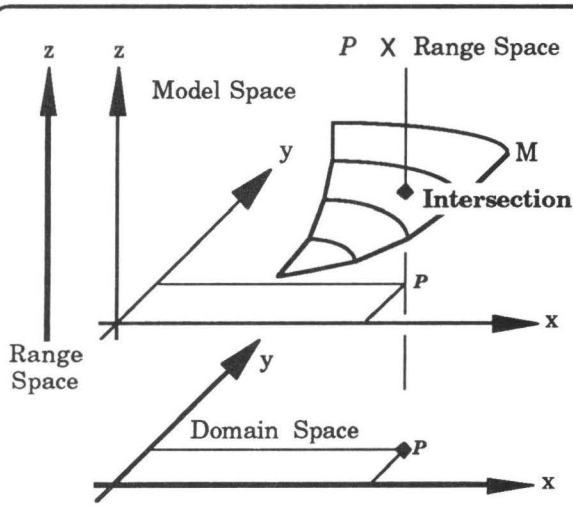


Figure 1(e) : Function application : Intersection of two models in the Model Space.

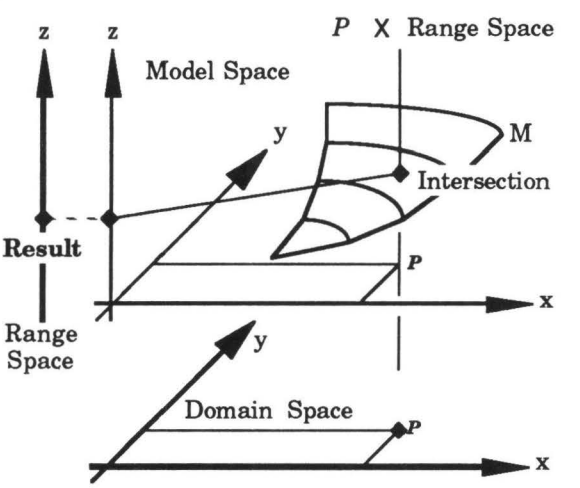


Figure 1(f) : Function application : Projecting the result into the Range Space

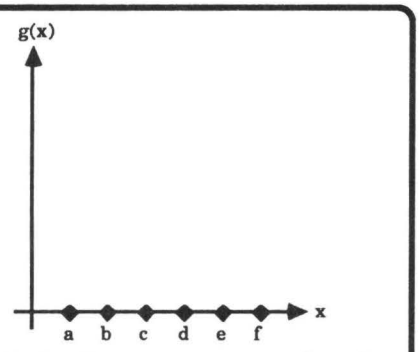


Figure 2 (a) : Enumerate points a, b, c, d, e and f in the Domain space.

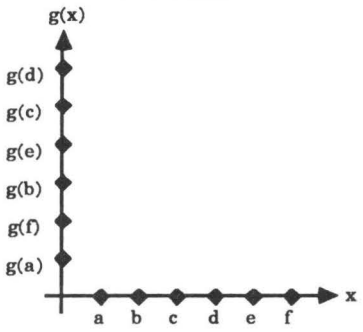


Figure 2 (b) : Compute corresponding image points in Range space

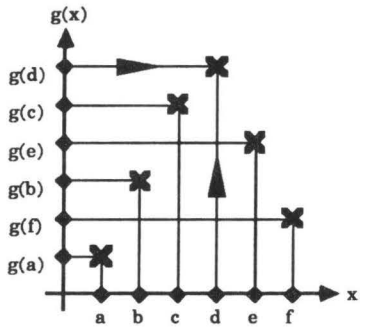


Figure 2 (c) : Construct points on the graph of g

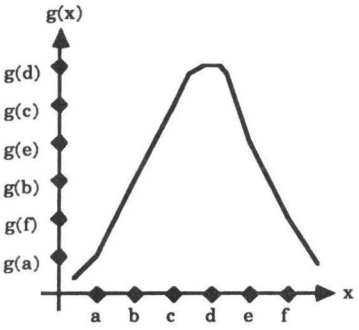


Figure 2 (d) : The graph of the function g

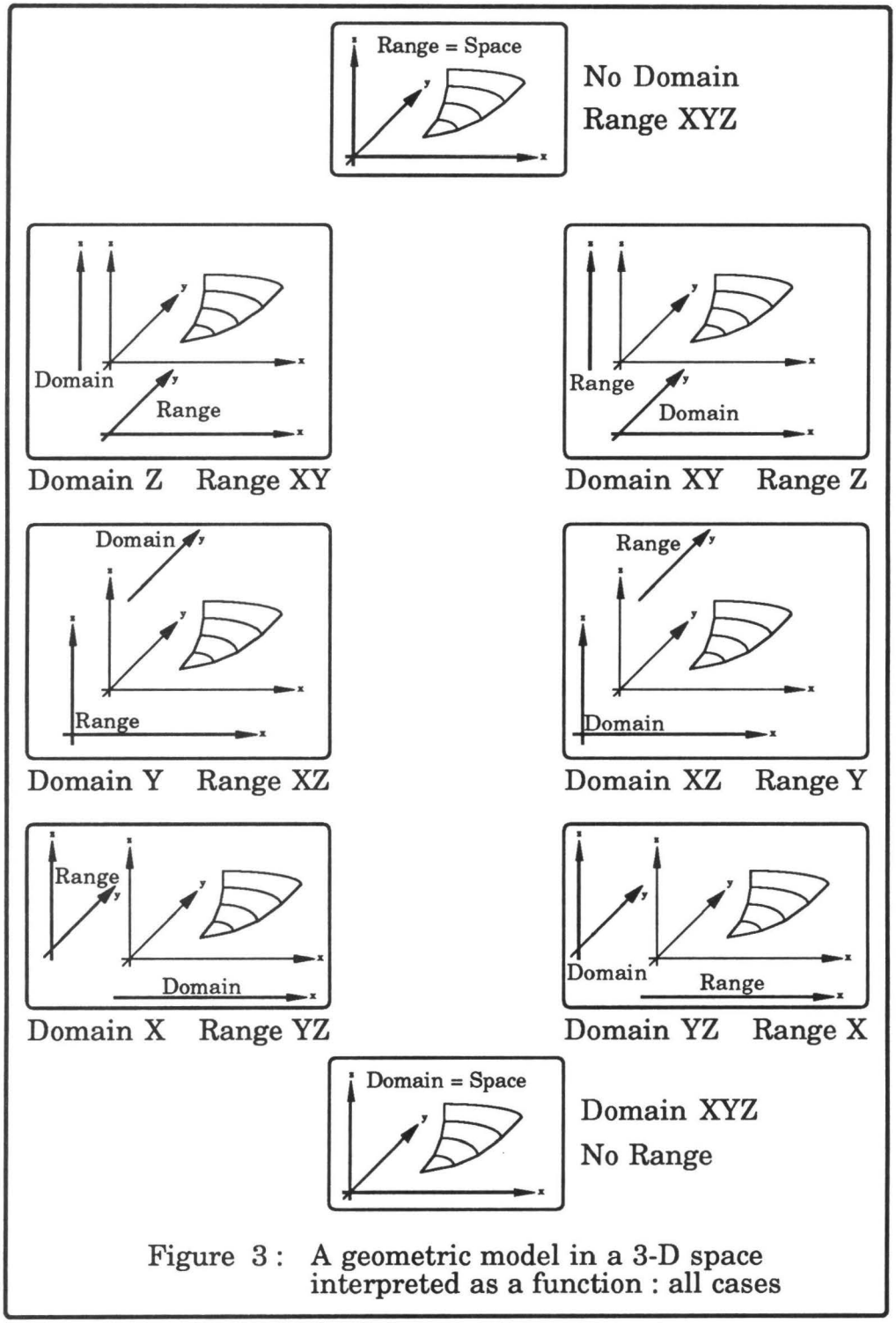


Figure 3 : A geometric model in a 3-D space interpreted as a function : all cases

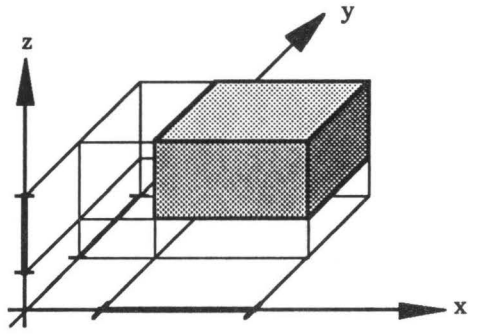


Figure 4 : An Excursion box is the Cartesian product of intervals

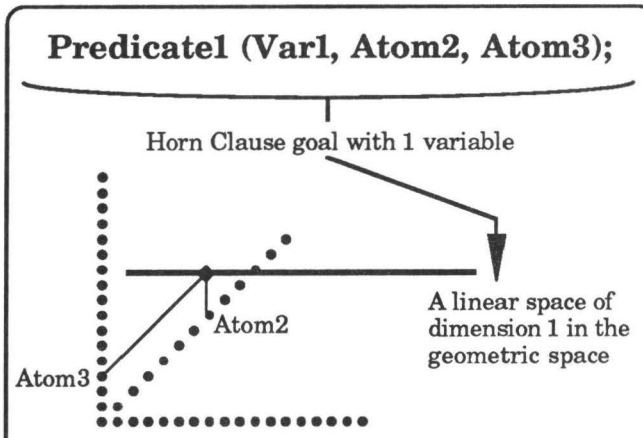


Figure 5 (b) : Geometric interpretation of a Horn Clause goal

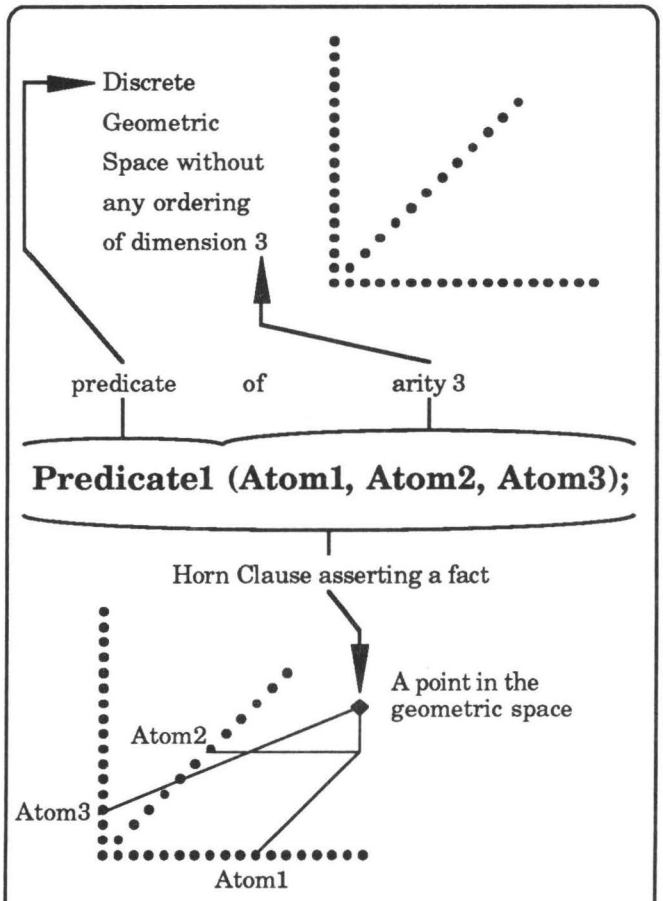


Figure 5 (a) : Geometric interpretation of a Horn Clause fact

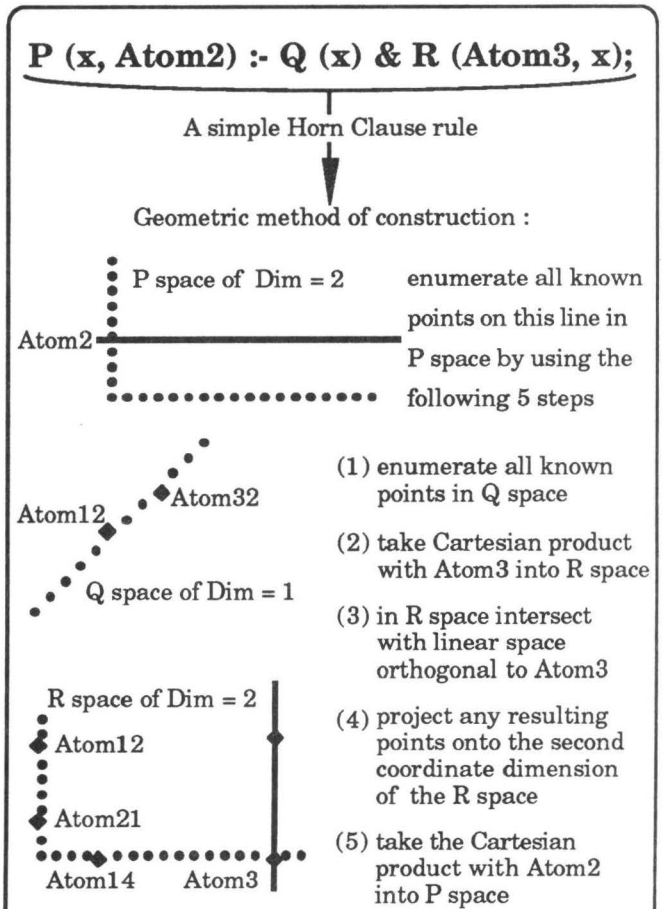


Figure 5 (c) : Geometric interpretation of a Horn Clause rule

The BiCad System

The Implementation of a Productmodelling System for Architectural Design

J.L.H. Rogier

The BiCad System

The implementation of a productmodelling system for architectural design.

ir. J.L.H. Rogier
Institute of Applied Computer Science (ITI)
Netherlands organization for applied scientific research (TNO)
P.O.Box 214 2600 AE Delft, The Netherlands
Telex 38071 zptno nl
Phone +31 15 69 69 00 ext. 7099

Abstract

In this report a description is given of the implementation of a designsystem which is based on the method describing productmodels. The report will describe a design theory, a productmodellingtheory and the integration of both theories in a concrete implementation. The results of a concrete application of the system in a designprocess are found in the report. These results will be referred to in the given examples.

Introduction

In this article a description will be given of the BiCad system. This system is developed and used for designing productmodels of architectural objects. This report frequently refers to an application which has led to the construction of a productmodel for kitchendesign. The aimed was to increase insights into the requirements for defining and using productmodels. In view of the limited set-up the concrete application suffers a number of shortcomings which limits its practical use as a productmodel. The most important restriction concerns the extendability of a productdefinition.
Theoretically speaking, a productmodel for kitchens is part of a productmodel of dwellings or buildings in general. A result of this is that a large number of entities which are part of present productmodel should indeed be placed in a more general model for buildings. On the other hand the concrete implementation of a productmodel is usually separate from its context.
A result of this fact is that the instrument has to have the capacity of generating productmodels that may be incorporated into a larger environment later.
The construction of a productmodel may be seen as being a part of a designprocess. As such the instrument that is used for constructing productmodels may be a part of a larger designsystem.

The implementation of the BiCad system aims at the construction of productmodels on one hand and at the use of productmodels as being a multi-interpretable medium for storage of knowledge about a designobject on the other. The techniques from AI are frequently used in constructing and exploitation of productmodels. The experience aquiered in the IIICAD-project is used to this end. The description of productmodels is based on the AEC-model of TNO-IBBC. The programminglanguage that is used for the implementation of the BiCadsystem is a object oriented dialect of Lisp.

Designtheory

The construction and use of a productmodel may be seen as being a part of a designprocess. According to the designtheory by Yoshikawa this kind of design is classed in the group of designmethods in which an explicit correspondence between the function space and the attribute space. The first implementation of the BiCad system is aimed at the application of Yoshikawa's so-called Catalogue - model.
The designprocess that belongs to this model may be described as a process of selecting a set of standardsolutions to a functionally described designproblem. By definition every designsolution consists of a collection of previously defined subsolutions which are described in a so-called 'catalogue' and a description of the relation between these subsolutions.
In the designprocess related to the cataloguemodel five phases can be destinguished:

1. Give a functional description of the designproblem. This is done by stating a collection of restrictions which limits the solutionspace in which the design solution may be found.
2. Select a collection of promising components.
3. Describe the relationship which has to be established between the selected components.
4. Evaluate the constructed solution in relation to the solutionspace as described in phase 1.

5a. Replace components or relations which do not fit and return to phase 3.
5b. Change the functional description of the designproblem by adding or replacing restrictions and return to phase 2.

According to Yoshikawa, the essential characteristic of a catalogue model is the correspondence between the descriptions of the function space and the attribute space. This means that the descriptions of both spaces should correspond in structure and vocabulary. In the BiCad system both aspects are realised by describing the entities from the catalogue and the functional description of the design problem in the same language; a productmodelling language. The languageconstructs involved in describing, controlling and supporting the designproces belong to the vocabulary of the design language. For the time being in the BiCad system these elements are implemented by a commandlanguage that is used to access the seperate facilities which the system contains. For future implemetations the aim is to formalize these facilities in a designlanguage.

In the cataloguemodel a sharp distinction is made between the description of the object and the description of the function which is used to select the object. This distinction renders the model extremely suitable for an implementation which differentiates between a design language and a productmodelling language.
The productmodelling language is used for the description of the structure of the objectdescription and the composition of the vocabulary that is applicable to the design object and the catagory of objects the design object belongs to.
The designlanguage contains elements which garantees the consistancy of a description of a productmodel in relation to a domain specific knowledge base belonging to the system. It also contains elements which enable the interactive construction a functional description of the designproblem.
The productmodelling language contains all elements needed for describing an object.
The designlanguage contains all elements for the construction of a functional description of the mechanism that is used for selecting entities from a catalogue.
Future implementations of the BiCad system will mainly pay attention to the extention of the possibilities of specifying this selectionmechanism. It is assumed that it is possible to develope the system towards a configuration that is suitable for supporting designmodels such as Yoshikawa's calculationmodel or productionmodel. These designmodels are also based upon an explicit correspondence between the description of the function space and the attribute space. The difference with the previously mentioned catalogue model has two aspects.
The first aspect concerns the possibility of expanding the knowledge base with entities eg. to add entities to the 'catalogue'.
The second aspect concerns the way the specification of the functionspace takes place.
Both models rely on a more flexible definition of the entities in the attribute space. This makes it possible to change values of attributes of design solutions based on a functional description of the designproblem.
The calculationmodel uses mathematical equations in which the user has to specify the values of the parameters. Apart from these equations the productionmodel uses heuristic rules which at least simplify the declaration of parametervalues.

Productmodelling Theory

The integration of a design system and a productmodelling system is based on an elaboration of an aspect of the design theory by Yoshikawa. This aspect concerns the description of entities it the attribute space. By elaborating on this aspect a separate theory enables us to formally include the aspect of the description of elements as a theory in the more general designtheory.
The productmodelling theory is based on two axioms.

A. Every object can be completely described by a collection of attributes.
B. The number of attributes needed to describe an entity is finite.
   This axiom conflicts with one of the axioms on which Yoshikawa's design theory is based. This axiom states that the number of attributes which describe an object is

infinite. The reason to define a conflicting axiom is that the number of attributes used to define an object theoretically speaking may be infinite but that from a practical point of view the definition of an object need not contain more information than is used from it. This means that the number of attributes defining an object is practically speaking limited by the number of attributes really required.

Apart from these two axioms the productmodelling theory is also based on a set of definitions.

a. An **attribute** is an elementary characteristic unit within the description of an object.
b. An **attribute value** is a fixed expression, based on a concrete object, in which the value of an attribute is given. An attribute value is an element from a set of possible values. This set of possible values is referred to as the "value-domain".
c. An **attribute relation** is a description of a relation of the values of different attributes.
d. An **abstract productmodel** is the joining of a set of attributes and a set of attribute relations with which an object can be described.
e. A **concrete product model** is the joining of a set of attributes and attribute values with which an object can be described.
f. A **knowledgedomain** is that subset of the set of attributes and attribute relations to which attention is paid as one set. Examples of knowledgedomains are geometry, construction etc.
   The set of attributes defining a knowledgedomain is referred to as the set of **manifest** attributes for this knowledgedomain. A knowledgedomain is used to establish values for attributes on the bases of the relations between attributes and their values, using known values of other attributes. This procedure is used both for calculating values and for verification of values.
g. A **view** is that subset of the set of attributes and attribute values which are important from a certain point of view. Examples of views are: visual representation, showing a construction calculus etc.
   The set of attributes used to define a view, is referred to as the set of **manifest** attributes (for this view).
   A view is used for explicit declaration and reproduction of values of attributes.
h. A **manifest attribute** is an attribute which is part of a description of a knowledgedomain or view.
   A **structural** manifest attribute is an attribute which belongs to the description of every knowledgedomain and view.
   A **potentially** manifest attribute is an attribute which belongs at least to one knowledgedomain or view.
i. A **latent attribute** is an attribute which does not explicitly belong to the description of a knowledgedomain, but which is associated to this description by means of an attributerelation.
   A **structural** latent attribute is an attribute which does not belong to any

knowledgedomain.
By definition, a view does not include latent attributes. It follows from the definition of the view that it only involves attributes and their values.Since this does not involve relations, there is no reference to attributes which do not belong to the definition of the view.

j. A **relation** is defined by a mathematical or logical expression referring to the attribute values of one or more attributes.

k. A **manifest relation** within a certain knowledgedomain is a relation between manifest attributes.
A structurally manifest relation is a relation between structurally manifest attributes.

l. A **latent relation** in a certain knowledgedomain is a relation which refers to at least one manifest attribute and at least one latent attribute.
A **structurally latent relation** is a relation which is not manifest in any knowledgedomain.
A **completely latent relation** for a certain knowledgedomain only refers to latent attributes belonging to this knowledgedomain.

m. A **limited** knowledgedomain only consists of manifest attributes and manifest relations. An **extended** knowledgedomain consists of the set of manifest attributes and the set of manifest and latent relations. An extended knowledgedomain implicitly requires a set of latent attributes. This set of latent attributes is referred to by the set of (not completely) latent relations.
A **complete** knowledgedomain consists of the set of manifest attributes and the set of all relations. This makes the set of all latent attributes implicitly belong to this knowledge domain.

n. An **applicationprogram** is a mechanism in which one knowledgedomain is integrated with at least one view. Depending on the kind of knowledgedomain which is used in the applicationprogram (def. m) there are **limited, extended** and **complete** applicationprograms.
The difference between a view and an application lies in the fact that a view can only refer to the concrete value of an attribute. An applicationprogram can refer to the implicit value of an attribute as results from the attributerelations belonging to the specific knowledgedomain. To put it in another way in an applicationprogram attributevalues can be calculated on the bases of other attribute values. This implies that the use of different knowledge domains may result in the calculation of different values for the same attribute and based on the same latent, fixed attributevalues.

o. A **description-entity** (which will be referred to as 'entity') is an explicitly defined collection of attributes and relations in the context of an extended knowledgedomain. Entities are used to define a set of attributes and relations as a unit. They structure the knowledge about a product, to begin with a specific knowledgedomain. The manifest relations function to define the internal structure of an entity. The latent relations function to define the (external) relations between entities.

On the bases of these axioms and definitions a number of theorems may be generated which are important to use productmodels as intermediaries for the description of a designobject within a designsystem. In this paper no proof for these theorems will be given. These proofs may be found in the article which is referred to[1]. In this article only the importance of the theorems related to the use of productmodels as an intermediary will be paid attention to.

1. An (abstract) productmodel can be regarded as a set of entities.
   This theorem enables us to describe a designobject by using a limited set of entities and relations. The entities enable us to abstract the description of a catagory of objects using a, for this catagory generic applicable set of entities and relations.
2. The set of structurally latent attributes is empty.
   The theorem is the formal representation of the elucidation of the above mentioned second axiom this theory is based on. To rephrase it, an object need not be described by more attributes than are used randomly chosen representation. There are no attributes within the description of an object as will be used within a knowledgedomain or view.
3. An entity is a mechanism for manipulating a knowledge domain (attributes and relations). As a result of the definition of both the domainknowledge and the different kinds of attributes and relations it is possible to define a hierarchy of different kinds of entities. This structured set of entities is used for constructing productmodels.
   The possibility to structure entities in hierarchies will be used to differentiate between the different levels of abstraction in the description of object. This theorem is based on the possibility to organize the total set of attributes that is used to describe object in different types of structures of entities and relations. These different types of structures relate to the several aspects which are dealt with during a designprocess.
   A **generic productmodel** consists of a set of entities which can be recognized as such by the different applicationprograms which use this productdescription.
   An **industrial type productmodel** is used for the description of a certain class of products characteristic for a specific type of industry.
   A **producttype productmodel** is used in the declaration of a set similar products.
   The sets of entities differ in structure and vocabulary but as they concern the same set of attributes they can be mapped onto one another. The BiCad system describes the relation between a producttype productmodel and a generic productmodel.
4. The **number of entities** needed for the description of a knowledgedomain is finite which means that a knowledgedomain can be defined as a finite set of entities. The theorem stresses the possibility to define a knowledge domain by means of a finite number of entities and relations.
5. An **entity** is an extended knowledgedomain.

[1]PML+IIICAD = BiCad J.Rogier 1988

This theorem states that an entity can be defined as a set of attributes with its manifest and latent relations. The manifest relations determine the internal structure of the entity; the latent relations determine the external relations or in other words the relations between the entities.

6. An **industry** is a complete knowledgedomain.
   The different structures of organisation which are used in grouping attributes in entities can be described in a vocabulary specific to a branch of industry. This theorem implies that although the organisation of the total collection of attributes in entities and relations differs for each branch of industry or producttype, the total collection of attributes is the same everywhere. This aspect of the description of products enables us to exchange information from different branches of industry on attributelevel. A generic productmodel may be used as an intermediary. This model describes an organisation of attributes in entities and relations specific for generally acknowledged knowledgedomains (eg. geometry, physics etc.).
7. If the number of attributes is finite (axiom 2) the number knowledgedomains is finite. If the number of attributes is infinite the number of knowledgedomains is infinite.
   In this theorem the number of restrictions for creating a generic model has been reduced to the determiniation of a finite number of knowledgedomains. The theorem argues that increasing the number of knowledgedomains which can be distinguished on a generic level will lead to an increase of the number of attributes in the total set of attributes.

AEC Model

The generic productmodel referred to in the BiCad system is based on the AEC model as was developed at TNO - IBBC. This model was one of the first successful attempts to standarize productdescriptions in a productmodelstructure. Another consideration which for this choice was the important role the model paid in the international standarisation committees (PDES/STEP, ISO/STEP).
One of most important aspects of the model is the relation between the Functional Unit and the Technical Solution. These basic entities are used for the description of the discrepancy between the entities used for the functional specification of the designproblem and the entities used for describing the technical solution for the designproblem. In the model it is assumed that the relation between the functional unit and the technical solution is explicit: for each functional unit there is one or more technical solutions; elements of the description of every technical solution is a set of indirect references to functional units in which the description of components is specified. The aims of a generic productmodelstructure such as the AEC model, go beyond the use of productmodels as intermediary for designobject specification. Up to now only those parts of this model are used which are involved in the designprocess. Besides the discrepancy between the functional unit and the technical solution the model contains a large number of entities which are used for the specification of geometric and kinethic properties of an object.

The BiCad system contains a part of the entities and relations from the AEC model. On the level of specification the entities functional unit, technical solution, functional requirement and procedural technical solution are distinguished.
In order to be able to realise a mapping from the producttype productmodel on a generic level the entity 'domainspecific attribute' is added. This entity is part of each of the other four entities previously mentioned. The 'domainspecific attribute' realizes the mapping onto the other entities belonging to the AEC model. Figures 1 and 2 give a survey of a part of the entities and relations of the AEC model which are implemented in the BiCad system.

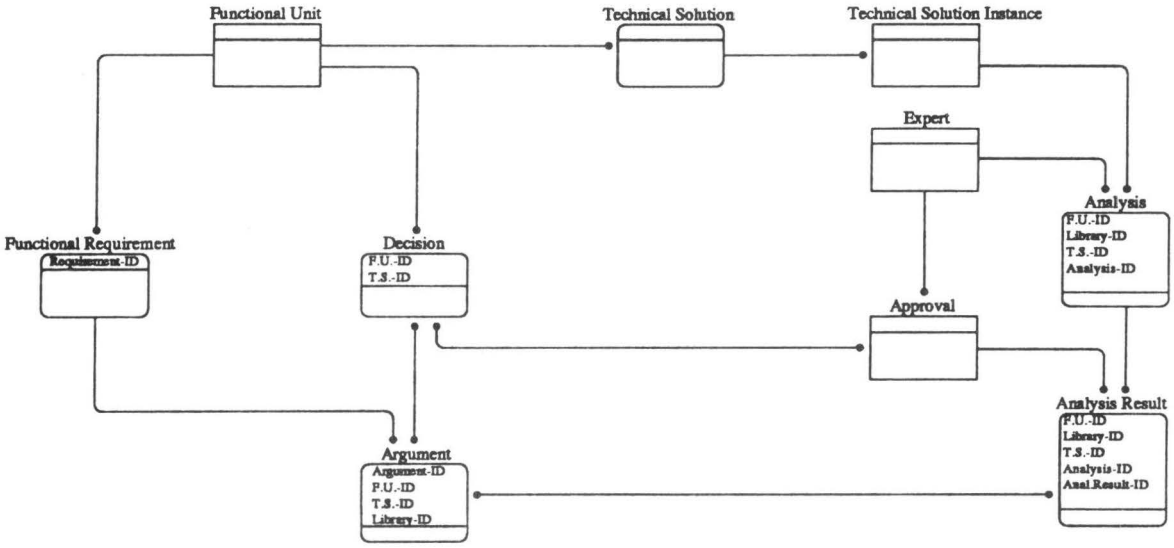


Fig. 1 = fig. B4 Evaluation and selection of technical solutions auteur W. Gielingh

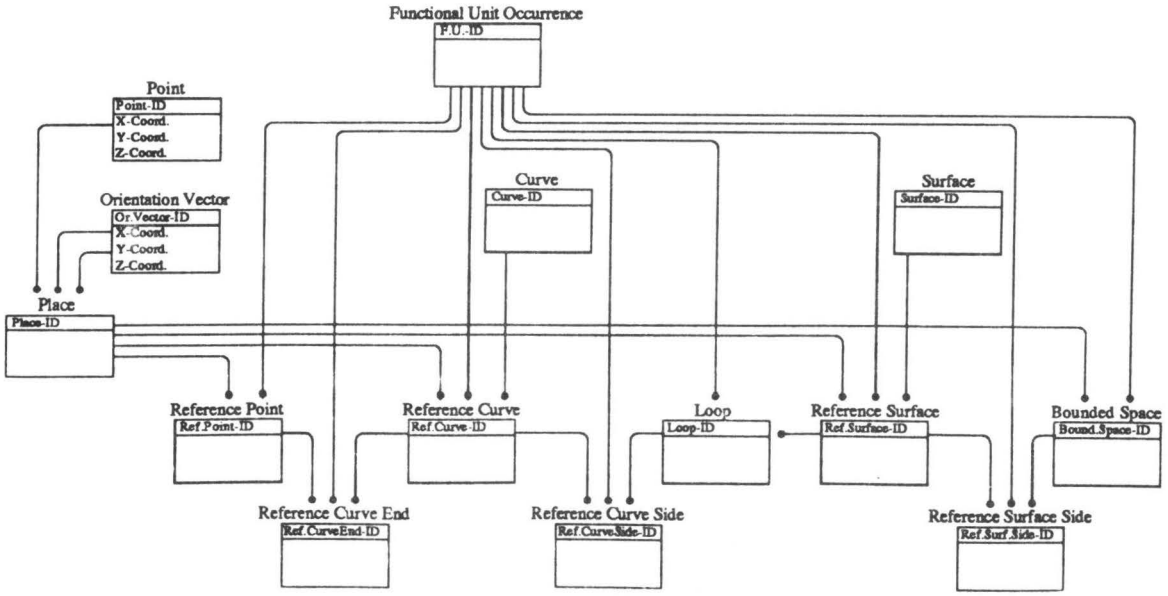


Fig. 2 = fig. C1 Topology & geometry auteur W.Gielingh

Implementation

The BiCad system is an experimental system for generating and using productmodels. Using the system, the process of declaring productmodels is devided into two separate activities.
The first activity consists of the declaration of a structure of entities and relations which will be used during the concrete specification of the productmodel itself.
The second activity consists of the concrete specification of productinformation using the above mentioned structure.

Within the first activity a vocabulary of entities and relations is compiled interactively. This vocabulary concerns itself with a certain catagory of products. It uses the following set of entities of the AEC model.

1. The functional unit. This entity is used for storing object oriented knowledge of a product.
2. The functional requirement. This entity is used for storing procedural oriented knowledge about a product.

Both entities concern that part of the vocabulary which is used by the user-designer. Using this set of entities and procedures a functional decomposition of the designobject may be reached.

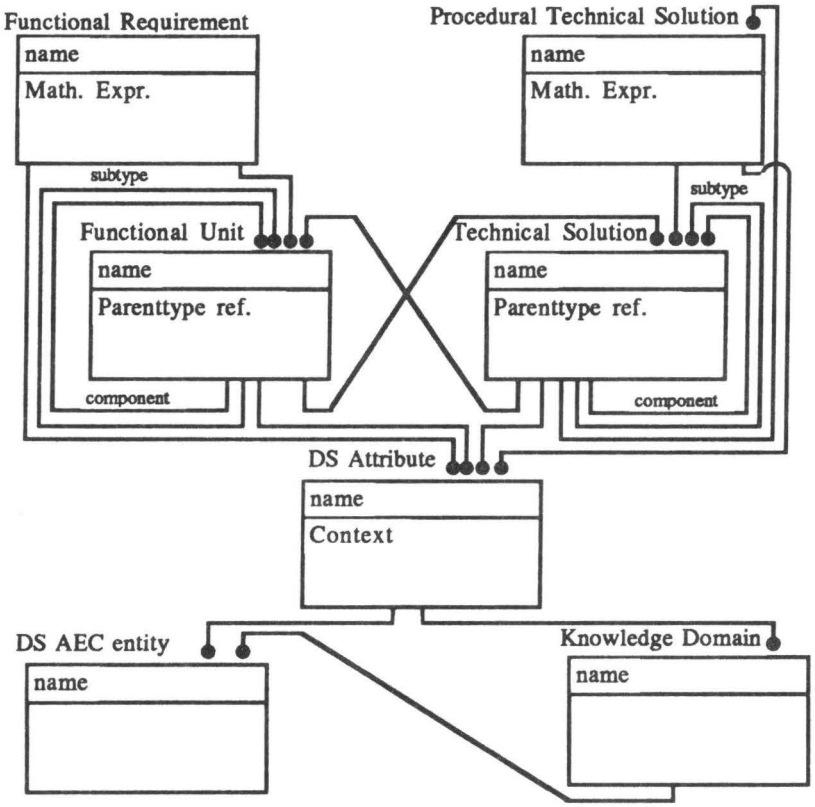
3. The technical solution. This entity is used for describing concrete objects which are specified in a catalogue of objects. The entity is used in describing the solution which can be found in a functionally described designproblem.
4. The procedural technical solution. Part of the description of a designsolution consist of the specification of the relations between the objects that belong to the set of concrete objects which are involved in solving the designproblem. This entity is used for describing the relation between attributes and attributevalues of technical solutions. As such it only refers to attributes which are defined elsewhere. In view of the context of the productmodellingtheory, the procedural technical solution is used for describing the set of attributerelations which are not part of the internal structure of entities.

Both the technical solution and the procedural technical solution concern that part of the vocabulary which is used in describing productmodels that is applied by the user-manifacturer. Using this set of entities and procedures a material decomposition of the designobject may be reached.

Apart form the four entities mentioned above a fifth one is distinguished. This is the entity (domain-specific) 'attribute'. This entity does not involve the collection of elementary attributes which is distinguished on the level of the AEC model. The function of the entity 'attribute' is twofold. In the first place this entity is used to

describe the relation between other entities. In the second place the presence of the entity as being a independant element provides the possibility to map a producttype - productmodel onto a generic model.

The composition between the different entities mentioned above is illustrated in figures 3 and 4.



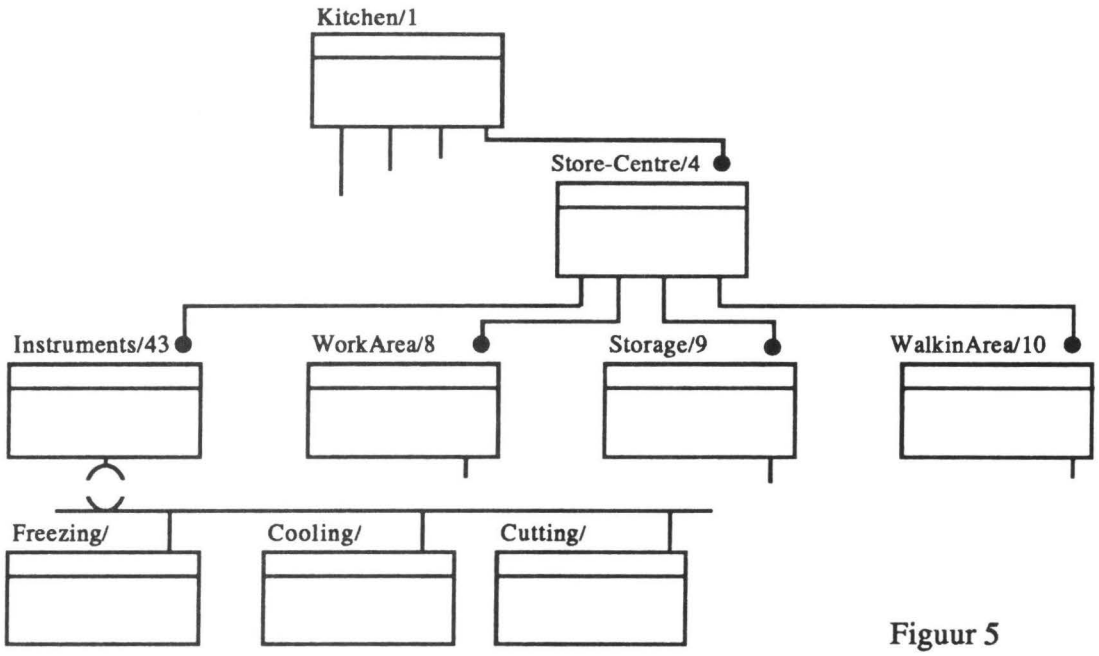
Functional Unit = [<component functional unit reference> + <..>] + [<technical solution reference> + <..>] + [<attribute reference> + <..>] + [Parenttype functional unit reference] + [<subtype functional unit reference> + <..>].	Technical Solution = [<component technical solution reference> + <..>] + [<procedural technical solution reference> + <..>] + [<functional unit reference> + <..>] + [<attribute reference> + <..>] + [Parenttype technical solution reference] + [<subtype technical solution reference> + <..>].
Functional Unit Requirement = [<functional unit reference> + <..>] + [<attribute reference> + <..>] + [Mathematical expression].	Procedural Technical Solution = [<technical solution reference> + <..>] + [<attribute reference> + <..>] + [Mathematical expression].
Attribute (domainspecific) = [<AEC-entity reference> + <..>] + [<knowledge domain> + <..>] + [Context].	

Figure 4

Functional unit

The component functional units which can be referred to in the context of a functional unit, are used to describe a functional decomposition of the object. The references from technical solutions consist of a set of possible solutions for the designproblem. The

parent- and subtype references describe the inheritance mechanism which is used in the declaration of entities. These references imply the inheritance of attributes and attributerelations by different entities.



Figuur 5

Technical Solution

The component technical solution every solution consist of describes the material decomposition of a Techical Solution. The reference functional units describe the set of functional units for which the technical solution may be used as being a solution. This reference is necessary to enable us to generate a functional specification of the designproblem using concrete objects during the declaration process. The procedural technical solution references describe a set of mathematical expressions needed to describe the internal and external relations between attributevalues. In figure given below an impression is given of a technical solution which may be used in the composition of a kitchen.

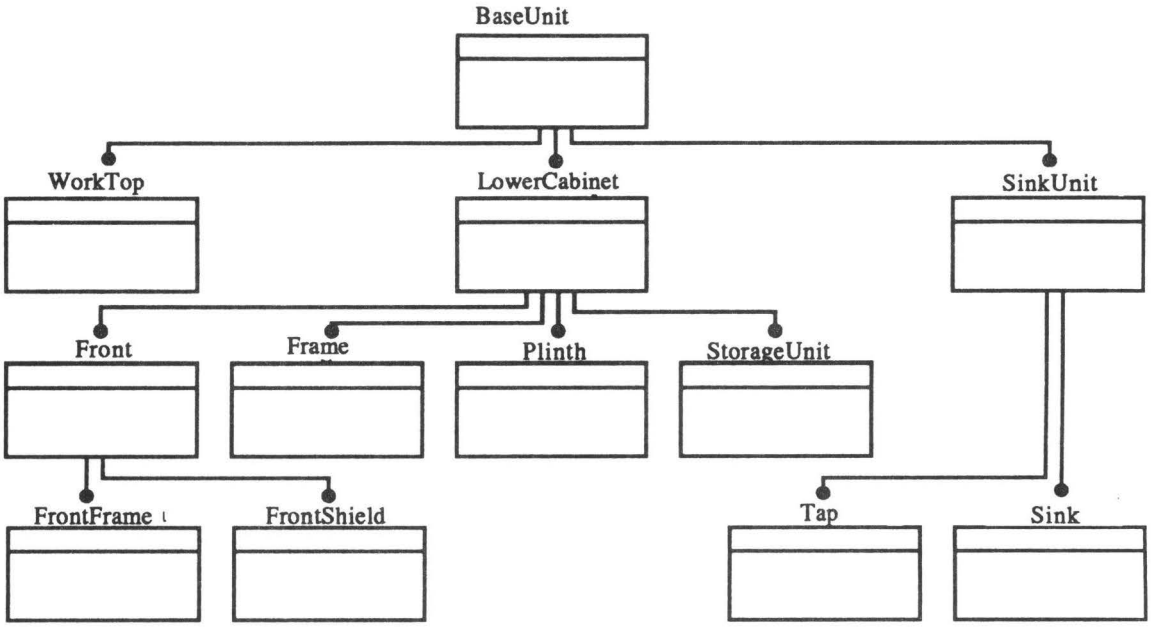


Figure 6 Material decomposition using Technical Solution entities.

Procedural Technical Solution

PTS: The diameter of the hole which is used for placing a tap in a worktop equals the diameter of the thread of the tap added with the standardtolerance.
==> WorkTop.TapHole.Diameter = Tap.Thread.Diameter + StandardTolerance
PTS: The minimal length of the thread of a tap equals the thickness of the worktop added with the thickness of the nut.
==> Tap.Thread.Length = WorkTop.Thickness + Tap.Nut.Thickness

In a procedural technical solution relations between values of attributes of technical solutions are specified. These relations are used to derive information from the technical solution. In the example the declaration of the value of the thickness limits the number of elements of the set of possible technical solutions for the choice of a tap. In its turn the choice of a specific tap implicitly defines the diameter of the taphole in the worktop. Another example of the application of procedural technical solutions is the description of the relative location of technical solutions. This also involves the use of attribute values of attributes belonging to different technical solutions. The technical solution which belongs to the functional unit 'kitchen area' is a 'kitchen'. This technical solution refers to different procedural technical solutions with which (for instance) the location of objects in the kitchen is described.
During the declarationprocess the technical solutions and the procedural technical solutions that are linked to these, function as frame-like structures. The values of attributes function as slots. When some values of this frame are declared values which may be derived are fixed. This excludes inconsistencies due to non-monotonic declaration. In the present implementation of the BiCad system it is only possible to use linear equations in order to specify procedural technical solutions. In other words multiplication of attribute_variables is not yet possible.

Functional Requirement

In the functional requirement relations between attributevalues of functional units are specified. A functional requirements limits the set of technical solutions which is described by the set of functional units. The functional requirements refer indirectly to the attributes of technical solutions.

FUR: "If the kitchen does not contain a dishwasher the drainingarea of the entity 'worktop' of the working area should be at least X square centimetre".
==> IF (no DishWasher) THEN (WorkArea.DrainingArea.Surface >= X cm^2)

In contrast to the procedural technical solution the functional requirement contains knowledge about functional aspects of the designobject. This knowledge is generally specified by means of heuristic rules which as rules may or may not be involved in describing a concept. The role of this set of rules differs from the role of the set of procedural technical solutions. A set of functional requirements is explicitly compiled during the designproces; in other words based on explicit declaration by the user. A set of procedural technical solutions is implicitly compiled; in other words based on the selection of a set of technical solutions. In specifying functional requirement rules IF...THEN statements are frequently used.

Both functional requirements and procedural technical solutions are used to limit the set of technical solutions. The procedural technical solutions are used to determine values of attributes based on values of other attributes. They are used to support the monotonic part of the designprocess. The functional requirements are used for the explicit determination of the solutionspace. They are used to support the non-monotonic part of the design-process. Every specification or selection of a functional unit or a functional requirement results in the determination of attributevalues belonging to the description of technical solutions. In the designlanguage IDDL this concept is denoted as a 'world'. The current implementation of the BiCad system destinguishes only one 'world'. Future implementations of the BiCad system will make it possible to remove previously declared information form the system in such a way that the consequences of this information will also be withdrawn. This mechanism is denoted as the 'multi-world' mechanism in IDDL.

The information that is compiled using the above mentioned entities determine the structure and vocabulary of the producttype productmodel. The actual implementation of the productmodel takes place through 'compiling' the model into a set of object-class descriptions. This set of objects classes is used to instantiate instances.
Different entities fullfil a different role. The functional unit and the technical solution determine the set of classes of objects the model is built from. The nomenclature of both kinds of entities determine respectively the vocabularies of the functional specification language and the technical solution language. The datatemplate of both entities is determined by the set of attributes to which they refer. Because the entities of

this set may themselves be decomposed into entities of the AEC model, indirect mapping between a producttype productmodel and a generic productmodel is established. Apart from the datatemplate, the set of attributes belonging to each entity determines a set of methods belonging to each object. The value of every attribute of an object can be declared explicitly. In order to declare the value of an attribute the internal structure of the AEC entities the attribute is mapped on, may be used.
A procedural technical solution implicitly determines the valuedeclaration of attributes. This entity is used in the specification of an extra set of methods belonging to the technical solution. The functional requirement plays a comparable role for the functional unit but is implemented as a separate objectclass which may be referred to in the context of a designprocess. This set of entities is generated and evaluated during the design-process.

An *Intelligent* Designsystem

In the context of an intelligent designsystem the concept of 'intelligent' has to be interpreted as: 'containing the possibility to manipulate and extend the amount of knowledge stored in the system'. This amount of knowledge stored in the system may be devided into three catagories. Domain specific knowledge, procedural knowledge and meta-knowledge can be distinguished.

In the first place a system contains *domain-specific knowledge*. This knowledge comprises a number of facts and rules concerning the domain in which the system is used. In an intelligent designsystem it should be possible to extend the domain specific knowledge interactively.This implies that the results from each previous designsession may be incorporated in each following design session. In an intelligent designsystem for the description of an aspect may be referred to as the structure and contents of previously defined objects.

The second kind of knowledge used in a design system is *procedural knowledge*. In procedural knowledge the relation between values of attributes of the designobject is described. The function of these knowledge entities is to calculate attributevalues, or to check the consistency of declared attributevalues in the context of the designobject. In an intelligent designsystem procedural knowledge is used as a reference to calculate values of attributes. An important part of procedural knowledge belongs to the domain specific knowledge of a system. The difference between procedural and domain specific knowledge is the way it is selected and manipulated. Domain specific knowledge is selected using libraries or knowledgebases with facts (which may or may not include a description of the context they are used in). Procedural knowledge is stored in knowledgebases with procedures. In contrast to a knowledgebase with facts, for the selection of procedures it is essential to know beforehand if they can be used in the given context. Using domain specific knowledge the presence of a description of the context may extend the application field of a given item, while when using

procedural knowledge the presence of a description of the applicationfield is essential in order to apply the item.

The third kind of knowledge that is used in an intelligent designsystem is *meta-knowledge*. The selection of the other kinds of knowledge entities takes place using this knowledge. In so-called expertsystems this kind of knowledge is pre-defined. The user of an expertsystem declares the values of attributes of a designobject using fixed patterns. In some cases this choice is limited by the system by means of an, at a certain moment selected subset of possibilities that is described in a domain specific knowledgebase. Based on the attributevalues declared by the user, the system automatically selects a set of procedures in order to check the consistancy of this set . In an intelligent designsystem the set of knowledge entities (both facts and procedures) can be extended. Compared to an expertsystem an intelligent designsystem has to use a more flexible way of selecting knowledge-entities.
The selection mechanisms that are used in an intelligent designsystem may be compared to patternrecognition. In the case of facts the description of the fact it is examined in order to find out if this fact may play a role in the given context. In the case of procedural knowledge it is examined whether the procedure may be used in the given context but is also examined if the specific procedure is applicable using the given set of attributevalues used to describe the context.

The three kinds of knowledge which are described before have to be incorporated in a intelligent designsystem. Now we will describe in which way these kinds of knowledge are implemented in the BiCad system.
The domainspecific knowledge is formalized in producttype productmodels. In the intro it was explained that a kitchenmodel is a part of a model for dwellings.This implies that a productmodel may be part of another model. In the BiCad system this aspect is implemented by making it possible to refer to previously declared models in the context of a new model. Procedural knowledge in the BiCad system is implemented using functional requirements and procedural technical solutions. The consistency of the application of procedures is reached in two ways. First of all explicit mapping takes place between the entities of a producttype productmodel and those of a generic model. In the second place use of an object oriented language results in a limited effect of the application of procedures. A special aspect associated with the application of procedures is the restricted application of a subset of procedures depending on the knowledgedomain they belong to. With the declaration of each attribute that is a part of the description of a functional unit or technical solution the user is asked to specify the knowledgedomain the attribute is applicable to. This property of each attribute makes it possible to limit the computerapplication used in a designprocess to a restricted number of knowledge-domains (designing from a specific viewpoint eg. visual, financial etc.). Analogous to the set of attributes, this set of knowledgedomains belongs to a specific catagory of producttype productmodels and is independant from the set of knowledgedomains that is specified in the context of the generic productmodel (eg. geometry, kinethics).

The first impulse to the use of meta-knowledge which was referred to before, may be found in the possiblity to refer back from a technical solution to a set of functional units, for which the technical solution may be used as a solution. In future implementations of the BiCad system this possibility of referring will be used as a mechanism to extend the functional description of the designproblem.

An important aspect of the extention of the amount of knowledge in the system, is the maintenance of consistancy within the system. This implies that extention of the amount of knowledge the system contains may not lead to misinterpretation of previously declared productinformation.
In the BiCad system an object oriented representation of the vocabulary that is used to define a class of productmodels, is used. For each succeeding concrete project the amount of knowledge eg. the set of language elements, is extended explicitly based on an already existing vocabulary resulting form previous designactivities. The mechanism that prevents that extention or change of the vocabulary results in the misinterpretation of previously built designobject description is based on two mechanisms. In the first place the system is written in an object oriented language. In the second place declaration of knowledge is kep strictly separate eg. knowledge declaration takes place in the context of a set of knowledgedomains.
Extention of the amount of knowledge may be characterised as follows:
In the first place the number of entities the object is represented by, may be extended. This may result in:
1: the creation of new classes of objects; and
2: a reorganisation of the inheritance-hierarchy which is used in describing objects.
In the first case there will be no consequences for previously declared productmodels because these do not use those new entities. In the second case the inheritance mechanism as is used in object oriented languages itself guarantees a consistant evaluation of previously declared productmodels. In other words as long as there will be no structural redefinition of objects and methods, a consistant evaluation of previously declared productmodels will be guaranteed. Future redefinition of a producttype productmodelling language therefore may only take place by extention.
The second kind of extention concerns an increase of the number of applicationfields for a class of producttype productmodels. This kind of extention results in the extention of the number of knowledgedomains that supports a certain class of producttype productmodels. Such an extention may cause an increase in the number of producttype attributes, the entities the model is build from. In that case evaluation of previously declared productmodels will result in a lack of information within these models when it is tried to represent them in the added knowledgedomains. This lack finds its origin in the second axiom of the productmodelling theory. This axiom claims that no more information can be extracted from an object as is declared. Therefore a solution to this problem may be found in the retrospective addition of information. This makes it possible to evaluate previous productmodels in an extended context. Inconsistencies

within the added knowledgedomains however can not be prevented. The possibility to limit the number of knowledgedomains that is used for the evaluation of the productmodel makes it possible to handle the consequences of these inconsistencies.
During the specification of the vocabulary of the producttype productmodelling language the BiCad system uses the mechanism of a stack in order to store all implicitly declared language-elements. The aim of this stack is not to bother the user with the existance of unknown entities during the specification of entities. In future implementations of the system this mechanism will also be used to extend the language element definitions in the context of previously declared productmodels.

Conclusion

The BiCad system is an experimental implementation which tries to integrate two important trends for designsystems into one system. In the first place a trend exists in which is tried to integrate techniques from artificial intelligence in design systems. In the second place a trend exists which tries to standarize the technique used for describing an arbitrary product using so-called productmodels.
The latest trend has led to the development of the productmodellingtheory. Through fitting a productmodellingtheory in a designtheory we try to create a formal basis for a designsystem that presents its results as productmodels. In doing so a practical need for integration of designknowledge and designrepresentation knowledge is met.

In the current implementation of the BiCad system attention is mainly paid to the construction and use of productmodels. The system can not be used yet to formalize designknowledge. This implies that during a designprocess results form previous designactivities may be used but the system is not yet capable to predict the users intentions. A mechanism as 'forward reasoning', which is necessary, may be implemented later based on the possibility to refer back from a technical solution to the functional unit this technical solution may be used for as a solution.

The BiCad system is based on a catagory of designmodels which are characterised by an explicit similarity in the definition of the function space and the attribute space. These are designmodels in which it is possible to specify the designproblem using the same vocabulary as is used in describing the designsolution. As a result the BiCad system is not applicable for the definition of designproblems where there is no explicit similarity in the description of the function space and the attribute space. In this catagory of designmodels, from which the 'paradigm model' as described by Yoshikawa is an example, a similarity as previously mentioned has to be established incrementally. The mechanisms that are required to translate a global definition into a vocabulary as needed to define both functional unit and technical solution, have not been implemented in the BiCad system. For future implementations it is possible to incorporate this kind of mappingmechanism as a 'shell' around the already existing BiCad system. However the question is if a certain kind of layered system is the right way to implement the paradigm model in a designsystem.

References:

F. P. Tolman Over Talen voor Productmodelleren
IBBC-TNO rapport: B-88-021

W.F. Gielingh Productmodellering in de bouw
IBBC-TNO rapport: B-87-581

W.F. Gielingh General Reference Model for AEC Product Definition
IBBC-TNO rapport: BI-87-87/68.5.4201

J.L.H.Rogier PML + IIICAD = BiCad
interne publicatie januari 1988

H. Yoshikawa General Design Theory and CAD system
Man-Machine Comm. in CAD/CAM, IFIP, 1981

H. Yoshikawa Automation of thinking in design
Computer appl. in production and engineering, IFIP, 1983

H. Yoshikawa, T. Tomiyama Knowledge engineering and CAD
Int. symp. on design and synthesis 1984 Tokyo

H. Yoshikawa, T. Tomiyama Requirements and principles for Intel. CAD systems
Knowledge engineering in CAD, IFIP, 1984

H. Yoshikawa General design theory as a formal theory of design
Workshop on Intelligent CAD, IFIP W.G 5.2

P.H. Winston, B.K.P. Horn LISP
Addison-Wesley Publishing Company 1984

T. Winograd Frame Representations and the Declarative/Procedural Controversy
(R.Brachman & H.Levesque Readings in Knowledge Representation)
Morgan Kaufmann Publishers, Inc. Los Altos Calif. 1985

M.Betz XLISP: An experimental Object Oriented Language
version 1.7 , Manchester NH, june 1986

P.Veerkamp, V.Akman Integrated Data Description Language Syntax and Semantics
In Intelligent Cad systems 2: Implementational Issues
Editors: P. ten Hagen, V. Akman
Springer Verlag Heidelberg 1988 to appear.

Generating 2D_Object from Axis–independent Information

S.–T. Wu

GENERATING 2D_OBJECT FROM AXIS-INDEPENDENT INFORMATION

Wu Shin-Ting

Technische Hochschule Darmstadt
Fachbereich Informatik - GRIS
Wilhelminenstrasse, 7
D-6100, Darmstadt
Federal Republic Germany
tel. 06151-1000-57, telex 4197367 agd d,E-mail wu@zgdvda.uucp

ABSTRACT

In this paper we present a new approach for specification and reasoning of 2D_object. A method is devised to construct the object's geometrical model from geometric constraints, which consist of a set of assertions about the shape, dimensions and spatial relationships of its boundary. The construction process consists in converting a set of imposed constraints in a system of equations and solving it through numerical methods. The results of applying the method to process any polygon's specifications (concave or convex) are shown.

Keywords: Engineering CAD, Geometry Reasoning, Geometry Specification, Graphical Data Structure.

1. Introduction

The use of traditional CAD systems to provide computer support is normally reserved for part of the final stage of the design process, which consists of a set of detailed and complete engineering drawings for designed products. Other design activities, like work planning, proposition of solutions and alternatives, their analyses and selection of the adequate solution [3], are also aided by computers, but as isolated activities.

This research is funded in part by the CAPES Grant (Brazil) under the doctorate program.

Researchers and scientists are currently engaged in searching for a way to integrate these computer-aided tasks. Such an integration would achieve higher productivity.

Ingenuity and creative thinking are essential for the success of engineering designs. The algorithms that each designing engineer uses differ a great deal and there is still no way to represent them and introduce them into the computer. Therefore, the designer (normally, a team of designers) and the computer need to become a working team. Engineering drawings play an important role in the communication of the design ideas, solutions and alternatives between the two.

One step towards the integration of "computer-aided designing activities" is to provide:

1) a systematic way for translating geometric specifications, such as size (e.g. length of segments), spatial relationships (e.g. which side is connected to another, the angle between them), shape (e.g. straight curve, cylindrical curve, etc), etc, in a concret object.

2) procedures for inferring geometry of object, even if the specification generated either by designer or by computer is not complete.

In this way we supply the gap that exists between engineering software packages, that deal with object's geometric properties' assertions (geometry), such as required dimensions of the room in a building design, and the drawing software packages, that deal with object's visualization (graphics). The user will not need to interpret geometric raw data and introduce it into the computer any more. The system by itself has an ability to automatically analyse the high level geometric data-input and provide the missing information.

Although the presented algorithm is limited to 2D_polygons (concave or convex), we believe that the same basic principle here discussed is suitable for any kind of geometric objects. Concerning with more complex shapes or geometric constraints would require more equations to express them analytically. The principles of the method and the architecture of the programs implementing it would be the same. Some of the features of the algorithm are:

1) supporting component-oriented specification (it is only necessary to specify what is the geometric primitives that bound the object and their relationship);

2) classifying the specification in: under-, exact-, over-determined, or inconsistent;

3) processing incomplete specified polygonal shape.

2. Related work

Lee and Andrews have developed a method to compute the location and orientation of each component from the spatial relationships imposed on the component in an assembly [1]. The principle is similar to ours, except in the application field. Therefore, the "mating conditions" are different and other appropriate methods for devising a system of equations, analysing and solving it are required.

[5] presents an algorithm, which has the ability to accept geometrical elements that are not completely defined by their own data. However, the requirements that coordinates of "starting- and finishing-points" are given explicitly is sometimes not very convenient. In our opinion a object's geometry is a proper characteristic of the object, which is independent of the object's location.

Sunde [9] has suggested a set of rules, with them geometric reasoning about a set of facts specified by the user can be carried out. This method sounds very efficient, if all the geometric knowledge is known by the computer. We think that some pure geometrical analysis-steps can be saved, if we can treat the problem in an analytical and numerical approach.

3. Topics

The article is organised as follows:

In Section 4 the background of the proposed algorithm for polygonal figure is explained. Two basic sets of constraints are used:

1) Local constraints: consist of axis-independent parameters that describe characteristic points of object (vertices of object). It is shown which information is needed to express coordinates of polygon's characteristic points in the Euclidean space as a whole;

2) Global constraints: define the analytical conditions that the coordinates of characteristic points must satisfy regarding the desired final geometry. The conditions that the characteristic points of a polygon should fulfill are detailed and modelled through algebraic equations.

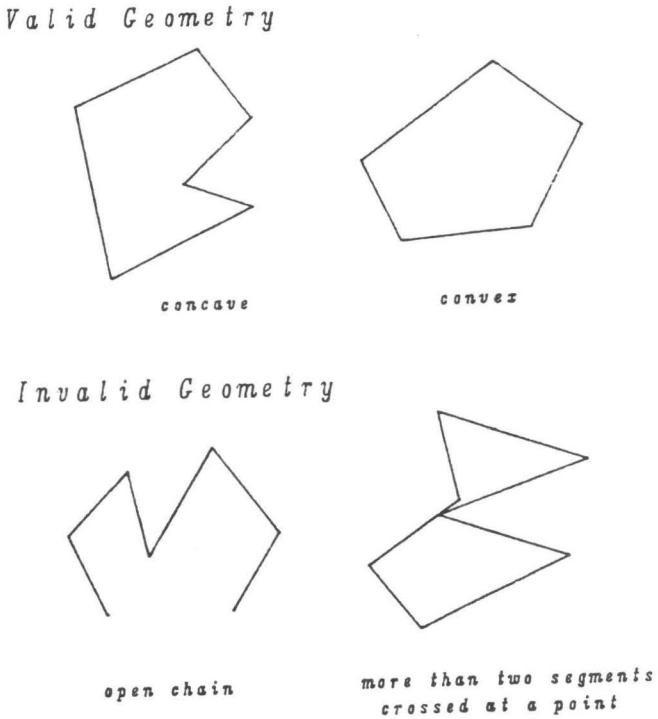


Fig. 1: Valid Constructions of Polygons

In Section 5 the required input data needed for processing polygonal figure is described. The way that the geometric information is processed allows a new approach of geometric data-input. The user does not need to have to define orientation of the object in the Euclidean space. He should only assert the subsets of the object's boundary and their adjacency-relationship.

In Section 6 the criteria for the classification of input data (polygon's specification) into an over-, exact- and under-determined one are given. The analysis task with the help of Algebra is rather simple. Auxiliary procedures to accelerate the decision are also presented.

Section 7 includes a presentation of a numerical method that we have used to get the solution for a not completely defined specification. We also discuss how the initial approximation is guessed and how the procedure is watched as the iteration is progressed in order to avoid the generation of an invalid shape.

Finally, in Section 8 we give some concluding remarks.

4. Background of the algorithm

To understand the philosophy of discussed object's construction process, it is necessary to distinguish two sets of constraints and to perceive the connection between them.

It is known that it is possible to represent any object by its boundaries and their aggregation - characteristic of boundary modelling. In such a

modelling an object is completely defined, if the coordinates of vertices are explicitly assigned. Thus, it requires the definition of a reference point and the input of coordinates of vertices relative to this reference point either explicitly or implicitly through equations that these coordinates should satisfy. These equations are expressed in the form:

$$\vec{f}(\vec{x},\vec{y},\vec{z}) = 0 \qquad (4-1)$$

However, the vertex coordinates can be defined, explained later, as a function of local constraints by establishing the adequate local coordinate system. In this way, it is obtained a set of axis-independent analytical descriptions of the form:

$$\vec{f}(\vec{x}(\Lambda),\vec{y}(\Lambda),\vec{z}(\Lambda)) = 0 \qquad (4-2)$$

where Λ is a vector of local constraints' variables.

4.1. Local constraints for polygon

The basic geometric primitive of polygon is segment (straight line). It is characterized by its length and its orientation in the space, or by two characteristic points: the initial and terminal points. It is chosen a local coordinate system for its analytical description, in such a way that their characteristic points can be expressed in terms only of its length. The construction of the local coordinate system obeys the following rules (fig. 2):

1)the abscissa of a local coordinate system lies at the segment, with the orientation to its end-point;

2)the origin of the local coordinate system lies at the beginning-point of the segment.

Then, the characteristic points of each segment in its local coordinate system are given by:

$$\begin{aligned} P_{initial} &= (0,0); \\ P_{terminal} &= (0,length). \end{aligned} \qquad (4-3)$$

The spatial relationship of each segment and its adjacent neighbors is given through the planar angle between them. With these two sets of information (length and angle), it is possible to compute the coordinates of characteristic points of the object relative to any specified reference point. The coordinates of $segment_{(i+1)}$ relative to its neighboring segment's local coordinate system i (in the clockwise orientation) are obtained by the successive transformations of translation of d_i (length of $segment_i$) along the x-axis:

$$TT_i = \begin{bmatrix} 1 & 0 & d_i \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (4-4)$$

and of rotation of φ_i (connection angle of the segment to its neighbor) about z-axis:

$$TR_i = \begin{bmatrix} \cos(\pi-\varphi_i) & -\sin(\pi-\varphi_i) & 0 \\ \sin(\pi-\varphi_i) & \cos(\pi-\varphi_i) & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (4-5)$$

of these coordinates, which can be compacted in the following generalized transformation matrix:

$$T_i = \begin{bmatrix} \cos(\pi-\varphi_i) & -\sin(\pi-\varphi_i) & d_i \\ \sin(\pi-\varphi_i) & \cos(\pi-\varphi_i) & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (4-6)$$

,where

φ_i = internal angle between $segment_i$ and $segment_{(i+1)}$. Its value domain is $[0,2\pi]$.

d_i = length of the $segment_i$. The value domain is $(0,\infty)$.

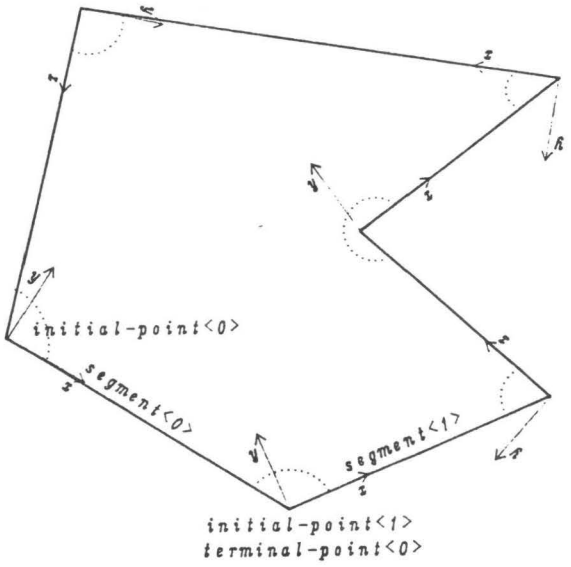


Fig. 2: Segments and their local coordinate system

A combination of the generalized matrices produces a partial or overall specification of a polygon's characteristic points. Consider

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = TT_i * TR_i * TT_{(i+1)} * TR_{(i+1)} * TT_{(i+2)} * TR_{(i+2)} * \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad (4-7)$$

which shows the partial specification of the initial-point's position of $segment_{(i+3)}$ relative to the local system of $segment_i$. Overall specification is obtained by choosing a global reference system and calculating each characteristic point's coordinates relative to it. Without losing generality, we choose the local coordinate

system of $segment_0$ as a reference coordinate system. Therefore, the coordinate of the initial-point (one vertex) of $segment_i$ relative to it is evaluated by the product of generalized transformation matrices:

$$New\_P_{(i+1)} = T_0 * T_1 * \cdots * T_j * \cdots * T_i * \begin{bmatrix}0\\0\\1\end{bmatrix} \quad (4-8)$$

Observe that by our proposed construction process the initial-point of $segment_{(i+1)}$ and the terminal-point of $segment_i$ should have the same coordinates (fig. 2) in regard to a global reference system.

Equation (4-8) can be used as a basis for computing the coordinates of any polygon's vertex in a function of metric values (length and angle). It is a parametric expression of each point, a subset of the polygon's boundary.

4.2. Global constraints for polygon

The coordinates of polygon's vertices should fit the required analytical descriptions of the polygon. In our system are considered the facts that the segments that surround the polygon form only one loop and that no more than two segments can intersect at a point in the space. The closed-loop condition for a polygon of <n> sides can be expressed as follows:

$$New\_P_n = \begin{bmatrix}x\\y\\1\end{bmatrix} = terminal\_point_{(n-1)} = New\_P_0 = \begin{bmatrix}0\\0\\1\end{bmatrix} \quad (4-9)$$

or,

$$\begin{bmatrix}g_1(\Lambda)\\g_2(\Lambda)\\1\end{bmatrix} = T_0 * T_1 * \cdots * T_j * \cdots * T_{(n-2)} * \begin{bmatrix}d_{(n-1)}\\0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\1\end{bmatrix}, \quad (4-10)$$

where Λ is a vector of length- and angle-variables and g_1 and g_2 are 2 analytical constraints (equations) on them. Notice that the last row plays no role in the calculation of characteristic points' coordinates.

If the angle between $segment_0$ and $segment_n$ is given, the connected angle between the line defined by $(terminal\_point_{(n-1)}, initial\_point_{(n-1)})$ and the line by $(initial\_point_0, terminal\_point_0)$ should be equal to this angle. This condition can be translated into:

$$g_3 = \varphi_{(n-1)} = \arccos\left(\frac{P_0 \cdot P_{(n-1)}}{|P_0| \cdot |P_{(n-1)}|}\right) \quad (4-11)$$

The equation (4-11) adds one more angle variable $\varphi_{(n-1)}$ into a system of global constraint equations.

These conditions are not sufficient to assure the validity of the constructed polygon. Another condition must also be taken into consideration: At each vertex of polygon (fig. 1) there are only two segments. As and when the coordinates relative to the same reference coordinate system are evaluated, conventional tests of intersection between segments are carried out. If the intersection is detected, the procedure is interrupted. If it is possible, adjustment action is taken.

The equations (4-10) and (4-11) constitute the chain between local and global constraints on a shape of a polygon. The metric dimensions are integrated in a system of global constraint equations. Hence, the unknown dimensions can be obtained algebraically and some anticipated information, that helps in the solution of the system, can be obtained analytically, as it is shown in the next section.

It is intersting to observe that the quantity of equations varies with the quantity of imposed global constraints. If constraints such as the parallelism of any two assigned sides are added, new equations, that describe them, should certainly be inserted.

Of course, there are also variations on the modeling of geometric primitives in regard to the geometric knowledge and geometric point-of-view of the expert that models the local and global constraints [8]. According to each case different set of algebraic equations can be produced. However, the principle of the description of geometric properties independent of a reference point from the user is always the same.

5. Input Data

Considering what we have discussed in the last section, it is perfectly possible to describe an object through axis-independent information and calculate from it object's vertices (coordinates) relative to any reference point, since the analytical description of restrictions and the model of primitives are known. The analytical description and the way of modeling primitives constitute geometric knowledge of a system. Particularly, for a system that supports processing of polygonal figure, it requires the knowledge about:

1) the segment's model for local constraints;

2) the polygon's model for global constraints.

With these two models implemented in a system only the structural and metric specification of a polygon is requested as input, that is:

1) the quantity of primitives that build the boundary of a polygon;

2) the spatial relationship of the primitives (the connection between segments and the value of its connection);

3) the local geometric information of the primitive (the length of segment).

6. Classification of polygon-specifications

A set of specifications for segments and their combination to form a polygon is valid, if and only if, from the geometric point of view, it satisfies the conditions explained in Section 4. To verify, whether a given set of specifications fulfill these requirements, an algorithm based on the analysis and the solution of equations (4-10) and (4-11) - the algebraic model of a polygon - is developed. In this algorithm, according to the number of dimensional and spatial local constrains (length and angle), four cases are distinguished in the definition of a polygon of <n> sides:

1) if the sum of the local constraints is more than (2n-1), the given data is overdetermined. The only tests to be performed are the non-intersection test and one to determine whether New_P_n equal to New_P_0;

2) if the sum of the local constraints is (2n - 1) or (2n - 2), the given data is also overdetermined. It is known that, if there are more equations than unknowns in non-linear system, it is likely that no solution exists. Nevertheless, if one does exist, it should be a solution that satisfies equations (4-10) and (4-11) [7]. A solution consists of a set of implicit local constraints;

3) if the sum of the local constraints is (2n-3), the given data can be under-, exactly determined with implicit local constraints, or inconsistent. It is based on the fact that, if the number of non-linear equations is equal to the number of unknowns and none of the equations is either redundant to the remaining equations or contradictory to the remaining equations, then discrete solutions exist [7];

4) if the sum of the local constraints is less than (2n-3), the number of variables exceed the number of equations, then, if a solution does exist, it is an infinite number of candidates.

A numerical method used to solve any kind of non-linear equation system is explained in the next section. Methods are added to control the progression of iteration, in order to assure the validity of resulted polygon's geometry. It should be noticed that this process produces **only one valid solution**, although a solution space for a system could be infinite. This limitation is acceptable, as we are in principle interested in any geometry that fits the imposed requirements.

To speed up the recognition of inconsistent specifications, some geometric reasoning can be accomplished before constructing the equations (4-10) and (4-11):

1)the sum of the internal angles of polygon with <n> sides must be equal to $\pi*(n-2)$ [2];

2)the length of each side of polygon must be shorter than the sum of the length of the other sides [2];

3)the connected segments should not form any loop.

7. Numerical methods

The method used for solving a system of <m> equations in <n> variables:

$$\begin{aligned} f_1(\lambda_1,\lambda_2,\lambda_3,\ldots,\lambda_n)&=0 \\ &\vdots \qquad \text{or} \quad \vec{f}(\vec{\lambda})=0 \qquad (6-1) \\ f_m(\lambda_1,\lambda_2,\lambda_3,\ldots,\lambda_n)&=0 \end{aligned}$$

is Newton-Raphson method, which uses the following iteration [7]:

$$\vec{\lambda}_{(k+1)} = \vec{\lambda}_k - \vec{f}'(\vec{\lambda}_k)^{-1} * \vec{f}(\vec{\lambda}_k), \qquad (6-2)$$

where $\vec{f}'(\vec{\lambda}_k)$ is the derivative of $\vec{f}$ at $\vec{\lambda}_k$, represented by the matrix of partial derivatives (Jacobian matrix):

$$J = \begin{bmatrix} f_1'\lambda_1 & \cdots & f_1'\lambda_n \\ f_2'\lambda_1 & \cdots & f_2'\lambda_n \\ \vdots & \cdots & \vdots \\ \vdots & \cdots & \vdots \\ \vdots & \cdots & \vdots \\ f_m'\lambda_1 & \cdots & f_m'\lambda_n \end{bmatrix} \qquad (6-3)$$

To use the Newton_Raphson's method, we must provide the formula for computing Jacobian matrix of equations (4-10) and (4-11). Each Jacobian element of equation (4-10) is easily obtained from:

PROPOSITION 1: For any chain of segments, the end-point of this chain has the following coordinates relative to the reference coordinate-system constructed in the way described in Section 4:

$$g_1 = x = (\pm 1)^n * [d_{(n-1)} * \cos(\varphi_0+\varphi_1+...+\varphi_{(n-2)}) + \cdots + d_1 * \cos(\varphi_0) + d_0]; \quad (6-4)$$

$$g_2 = y = d_{(n-1)} * \sin(\varphi_0+\varphi_1+...+\varphi_{(n-2)}) + \cdots + d_1 * \sin(\varphi_0).$$

PROOF:This follows from the expansion of equation (4-10), using trigonometry properties [2]:

$$\sin(x \pm y) = \sin x \cos y \pm \cos x \sin y;$$

$$\cos(x \pm y) = \cos x \cos y \mp \sin x \sin y.$$

and for the equation (4-11) the Jacobian components are given by the sum of products of the derivatives of the function g_3 at the characteristic points (P_0,$P_{(n-1)}$, P_0,$P_{(n-1)}$) and the derivatives of each of these points in the local constraint-varaibles' space:

$$g_{3'\lambda_j} = \sum_{i=1}^{i=4} \partial \frac{g_3}{P_i} * \partial \frac{P_i}{\lambda_j} \qquad (6-5)$$

where λ_j and P_i are respectively the metric variable and the coordinates of characterisitic point.

If the number of variables exceeds the number of equations in a system formed by equations (4-10) and (4-11), the Jacobian matrix is non-square. Therefore, a method must be devised to find a generalized inverse for computing Λ_k in each iteration. [4] has got satisfactory results with the Moore-Penrose inverse [6] in his algorithm. We have also applied it for finding the increment $\vec{\Delta}$ $(\vec{g}'(\Lambda_k)^{-1} * \vec{g}(\Lambda_k))$ in each iteration.

As the iteration progresses, we need to watch it in case it converges to the wrong solution, e.g. to a multiple-loop geometry. This test is performed after each iteration, in such a way that as soon as the possible wrong solution is detected, a new increment vector is chosen (graphically, it means a new direction of approximation) for the current iteration. The strategy that we have used for computing a new direction is based on the fact that in each iteration the new geometry should be valid: non-intersection between the segments, but between the adjacent segments.

The initial guess is important to assure the convergence of the method. The criteria that are used for it is based on the geometric properties. It is known that the sum of polygon's $segment_{(n-1)}$ is $\pi*(n-2)$. Though, we distribute equally the difference of the sum of known angles from this total among the unknown angles. For the unknown segment sizes, we take as the first approximation the major value among the known segment sizes. The resulting chain must not have any intersection, except between $segment_0$ and $segment_{(n-1)}$. If it fails, new attempts are carried out recursively.

Figures (fig. 3) and (fig.4) show two polygons processed by the implemented program. The not necessarily complete specifications are given on the left side and a set of valid values generated by the program are shown on the right side. From this set a polygon is automatically drawn.

```
Definition of primitives:
  s1(segment,10.0);
  s2(segment,15.0);
  s3(segment,30.0);
  s4(segment,?);
  s5(segment,?);
  s6(segment,?);
  s7(segment,30.0);
  s8(segment,10.0);
  s9(segment,?).

Definition of relations:
  r1(s1,s2,?);
  r2(s2,s3,120),
  r3(s3,s4,120);
  r4(s4,s5,120);
  r5(s5,s6,120);
  r6(s6,s7,120);
  r7(s7,s8,?);
  r8(s8,s9,?);
  r9(s9,s1,?).
```

```
Results:
  s4(segment,30.0);
  s5(segment,30.0);
  s6(segment,30.0);
  s9(segment,29.0);

  r1(s1,s2,42);
  r7(s7,s8,217);
  r8(s8,s9,50);
  r9(s9,s1,300).
```

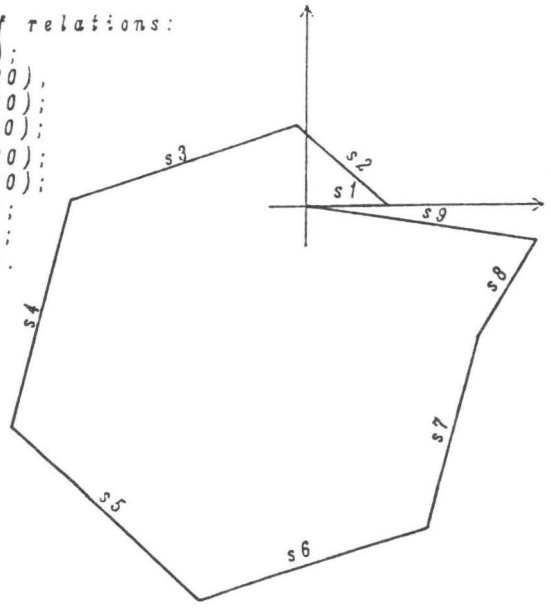


Fig. 3: Polygon specified by 10 constraints

```
Definition of primitives:
  s1(segment,10.0);
  s2(segment,15.0);
  s3(segment,30.0);
  s4(segment,?);
  s5(segment,?);
  s6(segment,?);
  s7(segment,30.0);
  s8(segment,10.0);
  s9(segment,?).

Definition of relations:
  r1(s1,s2,240);
  r2(s2,s3,120),
  r3(s3,s4,120);
  r4(s4,s5,120);
  r5(s5,s6,120);
  r6(s6,s7,120);
  r7(s7,s8,240);
  r8(s8,s9,?);
  r9(s9,s1,?).
```

```
Results:
  s4(segment,34.0);
  s5(segment,34.0);
  s6(segment,25.0);
  s9(segment,25.0);

  r8(s8,s9,77.0);
  r9(s9,s1,103.0).
```

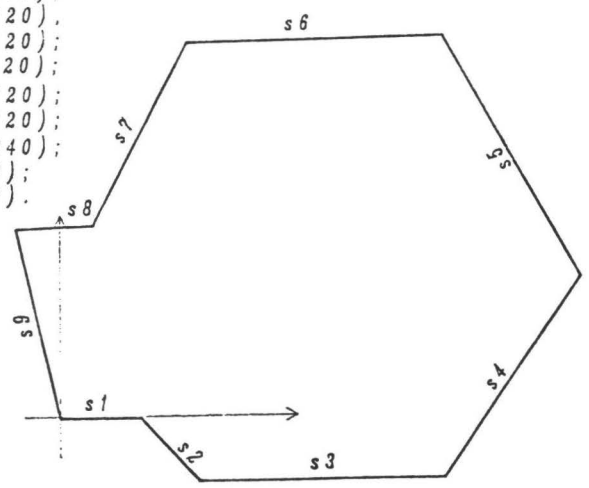


Fig. 4: Same specifications of (fig. 3) with two additional constraints

8. Conclusions

Based on a set of geometric constraints, the proposed algorithm provides a uniform way to infer a 2D_geometry. One principal feature of this algorithm is the elimination of an explicit definition of reference coordinate-system in the input-data. Only the relative information between the elements, that define the geometry of a figure, and the size of them are supplied. The advantages of this approach, among others are:

1) only intrinsic geometrical properties of figures are considered, which should be the same in any system;

2) description of object geometry without worrying about its location;

3) avoidance of error propagation caused by useless transformations from system to system.

The capability of the algorithm to process incomplete specifications is extremely useful during the product design, as the user can get "approximate drawing" of the product in the progress of the design. From this first image the designer can interactively introduce modifications until the satisfactory product is reached.

A program that was developed to verify the concept presented in this article has been implemented in C and runs on PCS computer. GKS-GRAL-2D, which is written in FORTRAN-77, is used to provide necessary graphical functions.

9. References

[1] Andrews,G. & Lee,K. "Inference of the positions of components in an assembly: part 2", CAD, Vol 17 No. 1, 1985

[2] Bronstein-Semendjajew, "Taschenbuch der Mathematik", Verlag Harri Deutsch, 1979

[3] Encarnaçao,J., Schlechtendahl,E.G., "Computer Aided Design", Springer-Verlag, 1983

[4] Dai,F, THD-GRIS, Darmstadt, Federal Republic Germany, private communication

[5] Hinduja,S. & Chen,S.J., "Geometry of 2D components", CAD, 1987

[6] Ben-Israel,A & Greville,T.N.E., "Generalized Inverses: Theory and Applications", Wiley, New York, 1974

[7] Pizer, S.M., "Numerical Computing and Mathematical Analysis", SRA, 1975

[8] Rogers, D.F. & Adams,J.L.,"Mathematical Elements for Computer Graphics", McGraw-Hill, 1976

[9] Sunde,G., "CAD System with declarative specification of shape", Draft Proceedings of First Eurographics Workshop on Intelligent CAD Systems, Netherlands, 21-24 April 1987