Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

V. Akman, P.J.W. ten Hagen, T. Tomiyama

Design as a formal, knowledge engineered activity

Computer Science/Department of Interactive Systems Report CS-R8744 September

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Design as a Formal, Knowledge Engineered Activity

Varol Akman, Paul ten Hagen, Tetsuo Tomiyama

Department of Interactive Systems
Center for Mathematics and Computer Science (CWI)
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract We summarize the framework and the preliminary results of the project *Intelligent Integrated Interactive CAD (IIICAD)* conducted at CWI. In e.g. mechanical computer aided engineering, currently much research aspires at using knowledge engineering but truly unifying approaches (i.e. "more" than expert systems) are lacking. IIICAD aims at filling this gap using the theory of CAD. Here, we show the concepts and the architecture of IIICAD. We then formulate how IIICAD can help in integrating existing CAD tools and establishing a workbench for design. We demonstrate the relevance and use of qualitative physics in machine design. As for the IIICAD software development methodology, we consider several aspects of software engineering as well as knowledge engineering, for CAD systems tend to be very large. IIICAD has a kernel language based on logic programming and object oriented programming paradigms. We explain the combination of these paradigms and give a taste of our IIICAD prototype which is presently under construction.

CR Categories (1987) D.2.m, I.2.1, I.2.3, I.2.4, I.2.5, J.6

Keywords intelligent CAD, design theory, logic, theory of knowledge, qualitative physics, object oriented programming, logic programming, knowledge engineering, prototyping

Note This paper has been prepared for an upcoming special issue of *IEEE Computer Graphics and Applications* on mechanical computer aided engineering.

1. Introduction

As a well-established integral part of Computer Integrated Manufacturing (CIM) [47], Mechanical Computer Aided Engineering (MCAE) is the backbone of today's highly industrialized world. It helps engineers develop products ranging from the simple and ordinary (e.g. irons, chairs, bicycles) to the complex and sophisticated (e.g. cars, robots, aircrafts). It multiplies the productivity many times and renders, using CIM techniques, robust products usually freed from the errors of fallible human designers.

However, the use of MCAE in the industry is not without problems. It is commonly accepted that current CAD systems are large software systems difficult to master and inflexible to adapt to growing needs. They can only deal with limited domains and occlude attempts to integration. They do not properly support a crucial ingredient of design, viz. interaction. What

---

The authors are members of Group *Bart Veth* which additionally includes (in alphabetical order): Peter Bernus, Jan Rogier, and Paul Veerkamp. Tomiyama is now with the Dept. of Precision Machinery Eng., Univ. of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan.

Report CS-R8744
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

is worse is that they lack a distinguishing characteristic of human designers. They have little or no intelligence [41].

Efforts to provide a wider perspective of CAD are now underway at several companies, universities, and research institutes. Since design is basically a mental activity, researchers have long felt the need for making CAD systems more intelligent. There is now a visible amount of ground-breaking work in this area which is commonly known as *intelligent CAD*. The reader is referred to [18, 17, 45, 41] for some representative articles.

Our work at the Center for Mathematics and Computer Science is directed towards contributing to the theory and practice of intelligent CAD systems. Our research project titled *Intelligent Integrated Interactive CAD (IIICAD)* is firmly based on knowledge engineering and theory of CAD to obtain a design system that is more substantial than the so-called "expert systems" [13, 5]. This paper gives a brief yet quite complete overview of IIICAD.

After this section, the rest of the paper is organized as follows. In Section 2, we summarize our evaluation of the current state of CAD along with an enumeration of the key problems waiting to be solved. Section 3 studies IIICAD in detail. In particular, concept, goals, and architecture of IIICAD are elaborated. Development strategies for IIICAD are presented in Section 4 which views IIICAD from software engineering and knowledge engineering angles. This section also offers a summary of the theory of CAD—our mathematical basis. Theory of CAD combines three theories: theory of design, theory of design objects, and theory of knowledge. Section 5 details the functionalities embodied in the kernel language that we use to build the IIICAD components. The kernel language unifies logic programming and object oriented programming paradigms. Section 6 concludes the paper by citing our future goals.

2. Problems with Existing CAD Systems

MCAE has evolved rapidly. Ivan Sutherland's revolutionary SKETCHPAD [39] in the 60's generated much enthusiasm for using interactive graphics in engineering. This in turn motivated a stormy decade when turn-key two-dimensional drafting systems gradually replaced the drawing boards in the professional environment. Finally, the dust settled with their general acceptance by the industry. Three-dimensional modeling systems became available, and they were also widely accepted by the industry as indispensable tools in product development.

Nevertheless, borrowing an analogy from [5], it is not unjust to claim that all these systems follow the *low road* approach. They are programmed in an *ad hoc* manner using prehistoric languages such as Fortran and regard design processes/objects from singular viewpoints, e.g. as geometric activities and information. Thus, despite their popularity, there are many problems with the existing CAD systems. (As we shall see, even the recent research cannot escape various inherited pitfalls.) In this section we highlight these problems in detail. We take mechanical parts design (machine design) as our domain of discussion since it has the strongest industrial appeal.

2.1 What is Methodologically Wrong?

As with any other discipline, the critique of other approaches to CAD systems development presupposes a starting point, i.e. a "view" of design. Briefly, we see design as an *intellectual* activity performed by human designers. We think that the essential thing in a designer is that he builds us his world. Thus we believe that design systems should provide a framework where designers can exercise their faculties at large. With this view, we support more or less the idea

of apprentice (assistant)—as opposed to autonomous—CAD systems. More on this rather crude division will be told in Section 3.2. Suffice it to say that we carefully distinguish our view from other common views about the nature of design such as

- Design is a routine process.
- Design is an inventive (innovative) process.
- Design is a problem solving process.
- Design is a decision making process.
- Design is an optimization process.

The first view regards design as a rather straightforward activity where the designer selects from previously known sets of well-understood alternatives. A recent example is the AIR-CYL system [8]. Clearly, this view is ingenuous and does not reflect the intricate nature of design. The second view embraces the exciting ideas of Artificial Intelligence (AI) to create novel devices by using knowledge of naive physical relationships, qualitative reasoning, planning, analogical reasoning, brainstorming, and discovery heuristics. It is quite early to predict whether this can be achieved in domains more involved than the usual toy worlds of AI, e.g. EDISON [14]. While the remaining views regard design partly as an activity requiring intelligence and creativity, they underplay its holistic nature. They are mostly implemented as expert systems [13] which solve specific problems of a specific design process. An example of this *middle road* approach is the PRIDE system which nevertheless is an interesting system with useful ideas behind it [33]. An annoying and often cited problem with expert systems is that they cannot deliver genuinely expert performance since they have no underlying mechanisms to understand what is going on [28]. The problem manifests itself when a particular expert system is unable to solve a simple problem in spite of its proven expertise with difficult problems. This discrepancy contributed to the emergence of terms like *deep* and *shallow* (although there are several drawbacks to such usage) in the AI literature [25].

High road systems are, in the preceding sense, deep systems and IIICAD is aiming at them. The knowledge of such systems is expected to represent the principles and theories underlying the subject "design." This may require that we try to demystify several aspects of design by way of formal, mathematical methods. In case of IIICAD, the fundamentals of General Design Theory which is based on axiomatic set theory can be found in [42]. It should be noted that we are not claiming that we know all the problem solving components of general design and can offer a comprehensive detailed model of it. Nor do we deny that there are many domain-specific sides to design. For instance, Very Large Scale Integrated Circuit (VLSI) design is mostly two-dimensional while mechanical design is inherently three-dimensional. IIICAD incorporates similarities in design, leaving the application-dependent issues to further consideration as side requirements. We believe that only through a clean design theory and formalization we can arrive at testable conjectures of design and build computer models of it. We shall later see that we regard logic as the principal tool for this formalization [22].

2.2 Key Problems to be Solved

It is useful to identify the problems of the existing CAD systems before we offer our proposal to solve them. The following list is not meant to be exhaustive:

• *CAD systems support few design processes and models—the latter being only partial and without integration.*

Producing final drawings is where current CAD systems excel. On the other hand, they are virtually powerless with respect to e.g. initial sketching. There are systems that can accept rough drawings but there is no system which can handle such crude information during the design process. The same holds true for natural language input but this is a controversial issue. At present, it is not clear why natural language should be useful in design even if it were reasonably available.

Another weakness in terms of design processes can be seen by carefully comparing software development and industrial machine development cycles. There is no corresponding tool in MCAE for what is known as *formal specifications* in software. The same goes for *documentation* which is *de rigueur* in software practice. Last but not the least, there are few established methods other than outright testing following manufacture to guarantee the *correctness* of design objects. This issue is admittedly more involved than it seems since the same problem still exists in the software world of today, although to a lesser extent. Furthermore, there is hope for CAD in that emerging areas such as naive physics and common sense reasoning may help (cf. Section 4.1.2).

Integration of models is essential since mechanical design deals with complicated gadgets. A design object must be viewed from various angles using different models. A simple example is a wrist watch which can be viewed as an intricate assembly of gears working in harmony, or as a simple device with two hands rotating about a pivot, or as an abstract machine pointing to numbers denoting the time. More complicated examples follow when we consider e.g. the kinematic, dynamic, finite element, and control-theoretic models of a robot manipulator.

• *CAD systems do not support error checking.*

Final outputs of conventional systems are so impressive that many errors go unnoticed for they exceed the mental capacity of designers. A remedy is to provide continuous error checks and to make sure that only the correct commands are accepted. Unfortunately, semantic error checks are difficult. A few systems provide good user interfaces which make suggestions against e.g. spelling mistakes but this is clearly only a small part of what is eventually needed.

• *Data entry is problematic.*

This mainly has to do with the lack of task domain terminology in the system. Because a conventional CAD system has no common sense knowledge of machine design and cannot follow the designer's intentions, one is likely to enter a good deal of information to state simple requests like "Here I need a hole to insert the shaft I just created." Things deteriorate quickly as the requests become more complicated. It simply becomes impossible to get anything achieved without undue emotional trauma. The ultimate solution is that CAD systems must accept substantially reduced yet comprehensive data instead of raw data.

• *Temporality, ambiguity, and inconsistency are not allowed.*

In design, instead of sticking to one particular idea we may want to experiment with several ideas. We may like to use the paragon of top-down stepwise refinement. This brings a time dimension to design. We may purge things we have previously introduced or introduce things that we have not considered before. We may require the system temporarily ''forget'' a particular facet of a design object since we are not concerned with it at the moment.

We also frequently want to separate the structure of a design object from the values of its attributes so that we can first decide about its ''shape'' during the conceptual design stage. For example, it is more important to recognize first the topology of a part if it is going to be inserted into another. Similarly, we may sometimes wish to acknowledge the existence of a point rather

than specify its exact location. The problem has been studied in database systems where it is known as *null values*. Simply stated, the fact that an entity has attributes is a different notion than the notion that an attribute has a value.

• *Unstructured code is commonplace.*

CAD systems are large. Software engineering tells us that maintainable software must be modular both "in the small" (to allow modification of minor components in specific applications) and "in the large" (to allow changes in major components based on say, the advances in technology) [46]. It is difficult to observe these characteristics in today's systems which are not open-ended and sturdy. Since they are not developed on a strong theoretical basis, they consist of huge amounts of code with unclear specifications, functions, and interactions. More often than not, efficiency is made a central concern and this gives rise to cryptic, spaghetti-like programs.

To return to the first point made in this section, there is a lack of integration. Mechanical engineering data exchange may cause deterioration of meaning. When we have a three-dimensional solid modeling system based on say, boundary representation we cannot easily exchange data with another solid modeling system based on say, constructive solid geometry. (The problem has connections with a fundamental mathematical problem known as *conversion between intensional and extensional descriptions* and will be touched upon in Section 4.1.3 [42].) Finally, current design systems do not offer a framework for design. Instead they give the designer a set of *ad hoc* functionalities through which he is supposed find his way out and make the best (cf. Section 3.2).

• *Symbolic and numerical computing are not coupled.*

Mechanical engineering systems normally use complex numerical and optimization procedures during design. However in many cases, *insight* into the problem solving process is not present. Insight is also needed to interpret the outcome of some computation. As Richard Hamming said, "The purpose of computing is insight, not numbers." Traditionally, mechanical design systems contain a good deal of numerical knowledge (e.g. bulky libraries of numerical code) but no intuitions. Users are left alone in analyzing the results of long, confusing computations. Recent research in *coupled* systems is directed towards integrating the explanation and problem solving ability of expert systems with the precision of numerical computing [25, 2].

The need for coupling in MCAE is especially felt in high-tech domains. Design of a space shuttle cannot be accomplished by merely symbolic or numerical techniques. Fluid dynamics, thermodynamics, finite element analysis, etc. should be integrated with symbolic techniques of database management, truth maintenance, logic, and symbolic algebraic computation. Since a numerical algorithm is a series of transformations on a set of numbers, procedural languages such as Fortran and Pascal are useful to implement it naturally and efficiently. On the other hand, a symbolic algorithm operates on symbols which are best represented in languages like Lisp and Prolog. For instance, the popular symbolic algebra package MACSYMA[†] uses Lisp as a base language.

[†] MACSYMA is a registered trademark of Symbolics, Inc.

3. IIICAD Principles

IIICAD must cover the three fundamental aspects of design: intelligence, integration, and interaction. It should support designers in the entire design process, using unified models with rich functionalities for various design activities. It must have models of the design objects which have maximum similarity to the designer's own images about them. At the same time, it must have knowledge about the design processes. This is achieved by explicit descriptions of the design processes and a control mechanism to guide the designer, viz. an intelligent supervisor.

3.1 Architectural Overview of IIICAD

The Supervisor (SPV) is at the core of IIICAD and controls all the information flow. It adds intelligence to the system by comparing user actions with *scenarios* which describe standard design procedures, and by performing error handling when necessary. While SPV corrects the obvious user errors, it does not have the initiative for the design process itself because IIICAD is envisaged to be a designer's apprentice, not an automatic design environment (cf. Section 3.2).

The Integrated Data Description Schema (IDDS) regiments the data and knowledge bases and relieves the user from the burden of specifying where and how to store and retrieve data. IDDS has a kernel language called Integrated Data Description Language (IDDL) spoken by all system elements. IDDL is the means to code the design knowledge and the design objects to guarantee integrated descriptions system-wide. IDDL will be based on logic and accordingly knowledge engineering is the key factor in building the IIICAD system. IDDL is the most crucial part in developing IIICAD and will be examined in Section 5.

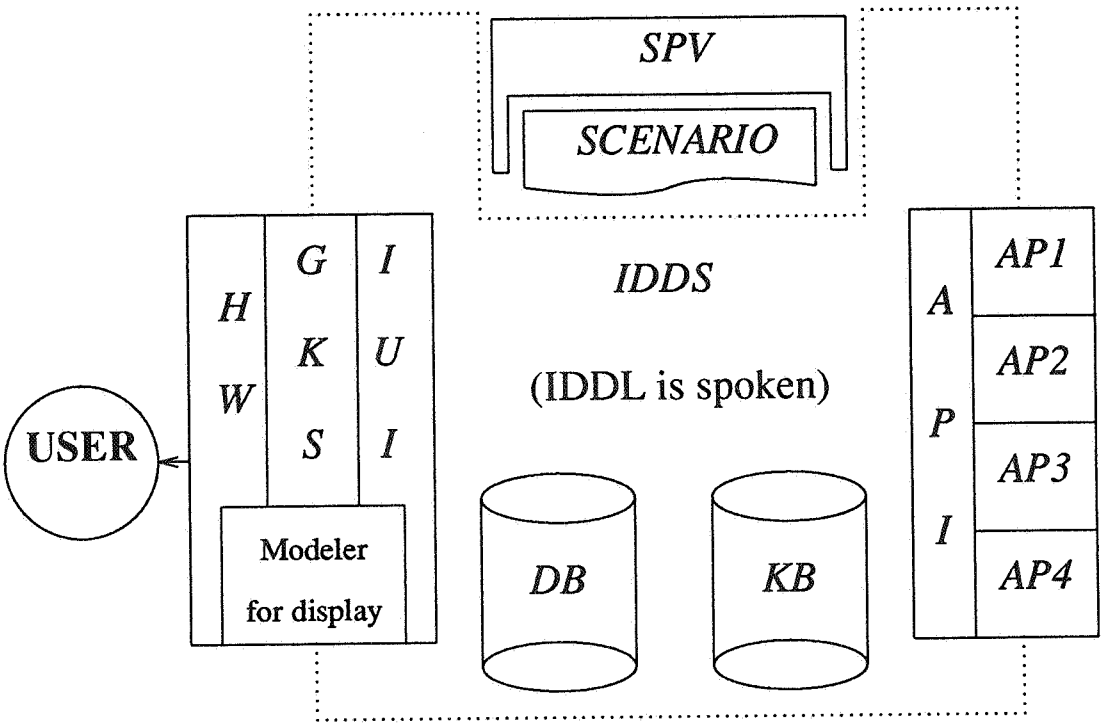

Fig. 1. IIICAD Architecture

In addition to the above principal elements, IIICAD has a high level interface called Intelligent User Interface (IUI) which is also driven by scenarios written in IDDL. IUI accepts

messages from the other subsystems of IIICAD and sends them to other lower level interface systems such as DICE (Dialogue Cells) and GKS [31, 23]. Syntactical errors of the user are processed by IUI whereas semantic errors can be processed by SPV. Application Interface (API) secures the mappings between the central model descriptions about the design object and individual models used by application programs. The following is a first-order approximation to the desirable application programs:

- Conceptual design systems to handle vague information.
- Consultation and problem solving (expert) systems for engineering applications.
- Basic/detailed design systems coupled with a geometric modeler.
- Engineering analysis systems such as finite element packages.

Figure 1 shows the elements of IIICAD in block diagram level.

3.2 Yet Another UNIX†?

There are several fundamental ways of looking at intelligent CAD system architecture. The following dichotomies are quite common [41]:

- CAD vs. Automated Design (AD).
- Design Apprentice (Assistant) vs. Autonomous Design System.
- Glass Box vs. Black Box.

The boundary between CAD and AD is indeed hard to delineate. We cannot object to the view that the ultimate aim of the computerization of design is to arrive at completely automatic design systems which can compete and even surpass the best human designers. However, the interactive nature of design will probably dictate that for a long time to come CAD as man-machine cooperation will dominate. The same holds true for apprentice vs. autonomous systems. An *apprentice* system has less hard-wired knowledge than an autonomous system but knows better how to interact. It also has a generic model of design. An *autonomous* system is very powerful for narrow domains. Besides in such domains there may not be a need for much interaction anyway. It is relatively easy to extend an apprentice by ''teaching'' it new skills. It is unwieldy to extend an autonomous system since its very constitution warrants myopia.

A more natural look at these dichotomies is via the metaphor ''glass box vs. black box'' [11]. If a CAD system has *glass box* structure then the user can at any time look through it to see partial results and processes (Figure 2(a)). On the other hand, a *black box* system resembles to a batch processing environment; one submits tasks to be executed and the system reports back with the results (Figure 2(b)).

The seasoned researchers of CAD remember those times when ideas such as ''general CAD'' have become fashion and then have been silently abandoned [24]. Today, demands for integration suggest that we may want to reconsider this sweeping panorama of design. The view which regards design as a large collection of intelligent tools [11] is different from the view which regards design systems as UNIX-like frameworks. The *intelligent tool* approach assumes that if you have a cooperating set of experts which can communicate with each other then you can solve many problems (Figure 3(a)). The *framework* approach regards the ''shell'' of design

† UNIX is a registered trademark of AT&T Bell Laboratories.

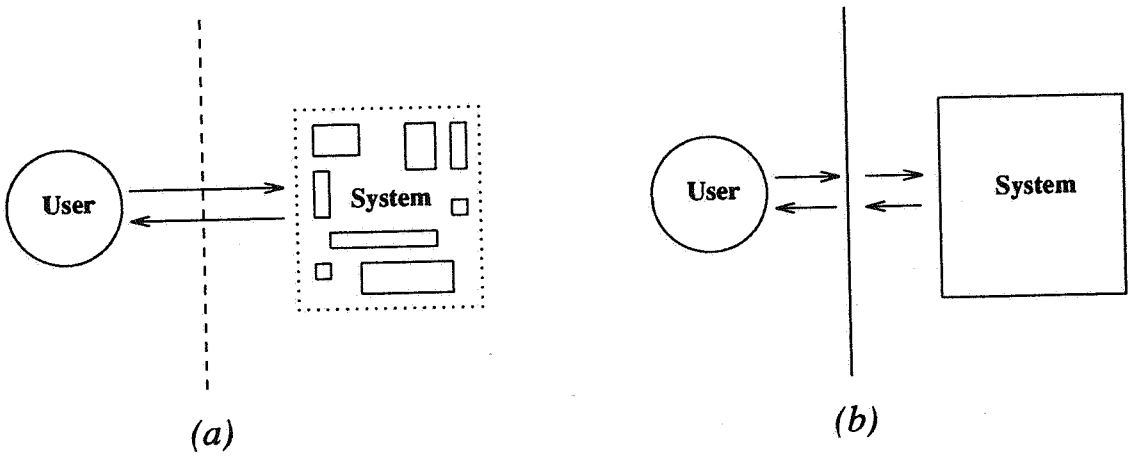

Fig. 2. Glass box vs. Black Box

systems as their biggest advantage; the domain specific issues can be dealt with separately, and using the facilities provided by the shell (Figure 3(b)). The shell may especially take care of system-wide communications and database management. In this sense IIICAD design is influenced by the initial design ideas of UNIX (although it may be claimed that currently UNIX is more like a collection of experts).

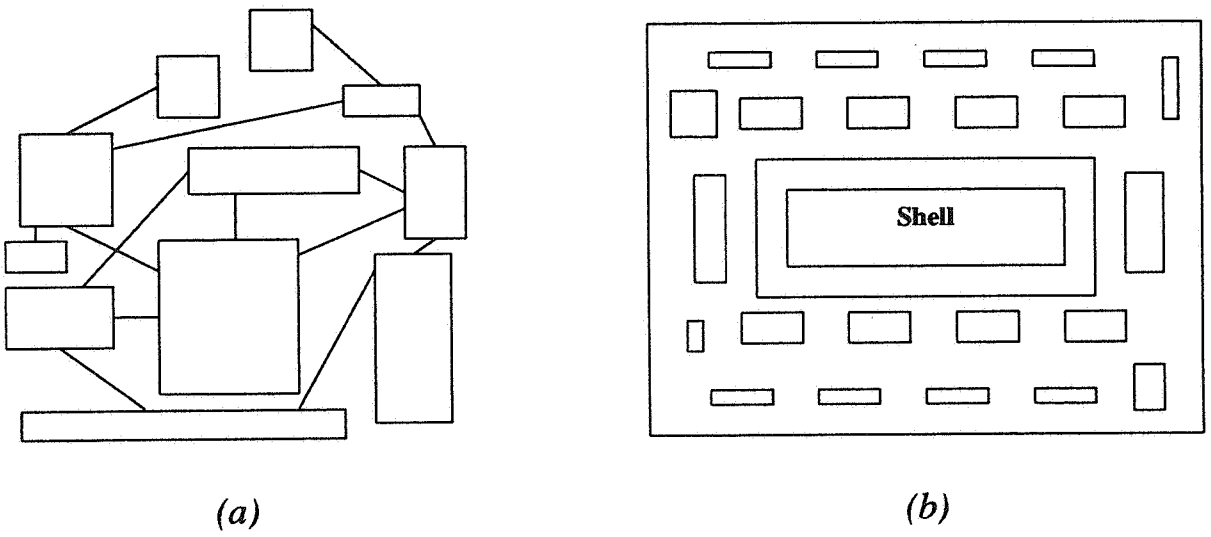

Fig. 3. Intelligent Tools vs. Frameworks

4. Strategies of Development for IIICAD

We must be aware of two caveats in IIICAD development. The first caveat states that everything should be based on a formal theory; only a firm basis allows us to avoid the infamous hacker trap—supplying numerous "features" which overwhelm the user. The second caveat states that we should be careful about software development; while knowledge engineering is qualitatively different from software engineering, the knowledge bases will still have to be maintained [5]. In other words, completed AI programs are just starting points for more advanced programs, obviating the traditional idea of "stable" databases.

4.1 Theory of CAD

There are three aspects of design: processes, models, and activities (Section 3). This implies that we need theories corresponding to each. A theory of CAD is then the aggregate of the following three theories:

- *Theory of design:* A theory which describes the design processes and activities [45].
- *Theory of design objects:* A theory which deals with the (models of) design objects. For the purposes of this paper, this is a theory of machines; in VLSI design it would be a theory of VLSI.
- *Theory of knowledge:* A metalevel theory to describe our knowledge about design.

We shall now look at each theory in more detail.

4.1.1 A Taste of General Design Theory and Logic

In General Design Theory [42], a design process is regarded as a mapping from the function space onto the attribute space (Figure 4). Both spaces are defined on an entity set. A design process is an evolution process about a metamodel. A *metamodel* is a set of attributive descriptions of a design entity. During design, new attributive descriptions will be added (or existing ones will be modified) and the metamodel will converge to the design solution. In other words, design specifications are initially presented in functional terms and the design is completed when all relevant attributes of the design object are determined so as to manufacture it.

To further illustrate a design process, we need to recognize three major components: *entities, attributes* of entities, and *relationships* among entities. A design process is thus a sequence of small steps (forward and backward) to obtain complete information about these components. Note that we also observe entities which do not change during design. For instance, when we design a bicycle lock, we use several pins and screws as building blocks which cannot be changed. We call such objects *invariants*. Description of the lock at a particular design stage is, oppositely, dynamically changing. We call these descriptions *variants* in a design process. Clearly, the concept of e.g. "hole" should not change while designing a lock; however values of the attributes such as the center coordinates of the hole may change. We call such changeable concepts associated with variants *covariants*.

In Section 2.2, it was underlined that to control the stepwise refinement of the design process we need to express unknown, uncertain, and temporal information about the design objects. We can accomplish this using an amalgam of intuitionistic (three-valued) logic, modal logic, temporal logic, inheritance, and situational calculus [22]. We take it for granted that descriptions of a design object are given by a set of propositions (such as well-formed formulae

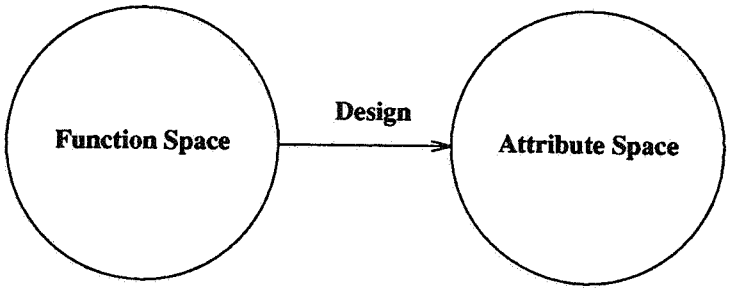

Fig. 4. Design According to General Design Theory

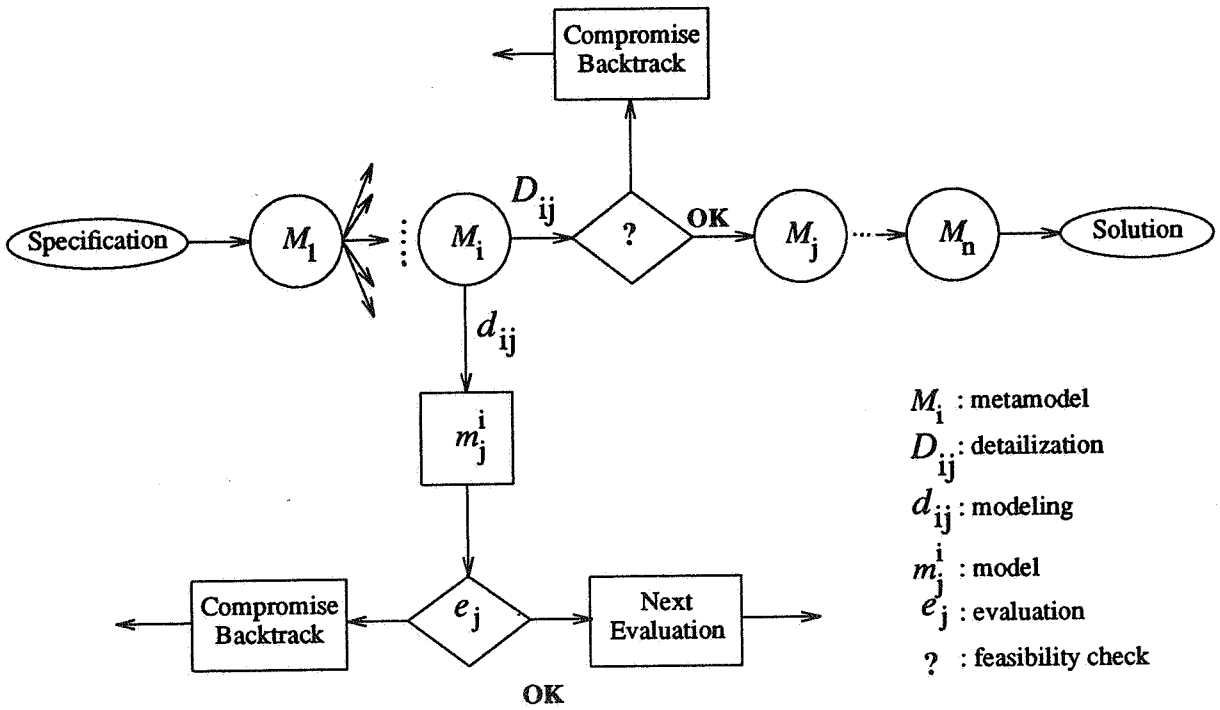

Fig. 5. Design Process Model

in first order predicate logic). A simplified model is then as follows (Figure 5):

$$\text{Specification: } s = T^0 \rightarrow \cdots \rightarrow T^n \rightarrow \cdots \rightarrow T^N = g.$$

$$\text{Metamodel: } M^0 \rightarrow \cdots \rightarrow M^n \rightarrow \cdots \rightarrow M^N.$$

$$\text{Propositions: } q_0 \rightarrow \cdots \rightarrow q_n \rightarrow \cdots \rightarrow q_N.$$

In this model, s is the original design specifications and g is the design "goal." Each design step has an associated set of propositions. Two central concerns here are (i) how to choose $\{q^n\}$

(i.e. how to proceed with design) and (ii) what if we discover $\neg q_n$ (i.e. how to deal with contradictions). We define a few other things:

$$q_n = p_0 \wedge \cdots \wedge p_k$$

where each p_i is an atomic fact concerning the metamodel,

$$\text{Model: } q_n \vdash_{C_n} m^n$$

where m^n is a model and C_n is control knowledge or modeling knowledge, and

$$\text{Detailing: } q_n, m^n \vdash_{D_n} r$$

where D_n is detailing knowledge and r is a proposition which should be added to q_n to arrive at the next description, q_{n+1}. With this notation the following partial classification of design becomes plain:

$$\text{INVENTION: Given } s \text{ find } \{q_n\}, \{C_n\}, \{D_n\}, g.$$

$$\text{NEW PRODUCT DEVELOPMENT: Given } s, \{C_n\} \text{ find } \{q_n\}, \{D_n\}, g.$$

$$\text{ROUTINE DESIGN: Given } s, \{C_n\}, \{D_n\} \text{ find } \{q_n\}, g.$$

$$\text{PARAMETRIC DESIGN: Given } s, \{q_n\}, \{C_n\}, \{D_n\} \text{ find } g.$$

Since the preceding discussion obviously views design as making changes to a database of facts, we are in fact quite close to the world of logic programming (understood in a general sense). Logic with modes of truth, viz. modal logic with necessity and possibility, can therefore be used as a notational tool. Here we have not only affirmations such as that some proposition is *true*, but also stronger ones such as that p is *necessary* (denoted $\blacksquare p$) and weaker ones such as that p is *possible* (denoted $\blacklozenge p$). If p is asserted *necessary*, then we can also assert p *true*. If p is asserted *true*, then we can also assert p *possible*.

Predicate calculus of higher order is useful to talk about inheritance (cf. Section 4.2.1). The following is provable in the second order predicate logic:

$$\forall F \quad [F(x) \supset G(x)].$$

(If an individual x has every property then x has any property G.) In third order predicate logic, we can prove that

$$\forall F \quad [V(F) \supset V(G)].$$

(Whatever is true of all functions of individuals is true of any function of individuals G.) While they are, theoretically speaking, well understood, the real challenge of higher order logics lies in their implementation.

For temporal logic, we can use the following notation. Let $t \; \alpha \; p$ denote that p holds *after* time t and $t \; \beta \; p$ denote that p holds *before* time t; $[t_1, t_2]$ denotes a time interval. Several useful equalities suggest themselves:

$$t \; \alpha \neg p = \neg (t \; \alpha \; p),$$

$$t \; \alpha \; (p \wedge q) = t \; \alpha \; p \wedge t \; \alpha \; q \quad \text{(ditto for } \vee),$$

$$[t_1, t_2] \; \alpha \; p = t_1 \; \alpha \; p \wedge t_2 \; \beta \; p \wedge t_1 < t_2.$$

Using temporal logic, we can describe inference control for production rule systems in a more explicit way. For instance, in Prolog the order of rules matter [4]. In general, this knowledge is embedded in the interpreter of this language. By disclosing this control we may introduce suppler control. As an example, detailing knowledge D_n above may be a set of rules of the sort

$$t_1 \,\alpha\, q_1 \wedge t_2 \,\alpha\, q_2 \wedge t_1 < t_2 \supset t_2 \,\alpha\, q_3,$$

$$t_1 \,\alpha\, q_1 \wedge t_2 \,\alpha\, q_2 \wedge t_2 < t_1 \supset t_2 \,\alpha\, q_4.$$

Intuitionistic logic can naturally be incorporated into temporal logic. Let t_p be the time when proposition p is proved. By definition, we have $t_p \;\alpha\, p \equiv \mathit{true}$. Now, introducing a logical symbol *unknown* we can formalize intuitionism in terms of temporality:

$$t_p \;\beta\, (p \vee \neg p) \equiv \mathit{unknown}, \quad t_p \;\alpha\, (p \vee \neg p) \equiv \mathit{true}.$$

We note that incorporating the complete functionalities of these assorted logics may very well result in high (even intractable) computational complexity. To avoid this bottleneck, we include only those functionalities which are relevant to our design requirements. For example, in case of temporality we are satisfied with only α and β although there is more to temporal logic than these simple operators.

4.1.2 Naive Physics and Machine Design

In machine design, it is not yet precisely known in which symbolism we can describe the functions of machines. There is, however, a view that functions can be represented in terms of the physical phenomena that the machine exhibits [42]. From this point of view, the representation of functions can be reduced to the representation of physical phenomena and qualitative reasoning about them.

Naive physics observes that people have a ''different'' kind of knowledge about the physical world. This knowledge can best be described as common sense and is attained after long years of interaction with the world. Accordingly, naive physics ideas are useful in machine design and we want to codify them in the IIICAD system. To this end, we follow the proposal of [22] and hope to capture naive physics in *logic*. Additionally, we use as a mathematical tool *qualitative reasoning* based on simple *symbolic* manipulations.

Consider the following concepts underlying ''change'' in physical systems: state, cause, equilibrium, oscillation, force, feedback, etc. Naive physics regards these concepts in a simple qualitative way. It maps continuous variables to discrete variables taking only a small number of values (e.g. +, –, and 0). Accordingly, differential equations are mapped to qualitative differential equations (also known as confluences). Consider Figure 6 which depicts a pressure regulator and is adapted from [27]. The confluence

$$\partial P + \partial A - \partial Q = 0$$

describes this device in qualitative terms. Here P is the pressure across the valve, A is the area available for flow, and Q is the flow through the valve; ∂ means change. Let us look at the way the regulator works. (In the following $\Rightarrow$ means ''implies'' and $\uparrow$ and $\downarrow$ stand for ''increase'' and ''decrease,'' respectively.) An $\uparrow$ in pressure at a $\Rightarrow$ an $\uparrow$ in pressure at b $\Rightarrow$ an $\uparrow$ in flow through b $\Rightarrow$ an $\uparrow$ in pressure at d $\Rightarrow$ diaphragm at e pushes downward $\Rightarrow$ valve at b tightens $\Rightarrow$ a $\downarrow$ in area for flow $\Rightarrow$ a $\downarrow$ in flow through b. Thus the regulator maintains a constant pressure

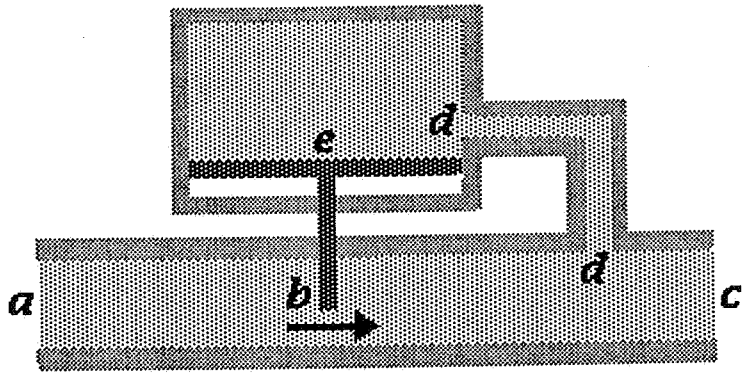

Fig. 6. A Pressure Regulator

at c in spite of fluctuations at the supply side a. Note that if the valve is closed (i.e. $\partial A = \partial Q = 0$) then it is predicted that P is constant. The correct confluence for this state should instead read $\partial Q = 0$; we simply do not say anything about P. Such time periods during which a device shows significantly different behavior are known as *episodes* [16].

The use of qualitative reasoning unquestionably facilitates the analysis of the working of physical devices. However, this usage is not without price. Qualitative techniques may cause ambiguities. Assume that a certain quantity M varies with N_1/N_2, i.e. $M \propto N_1/N_2$. If $N_1 \uparrow$ while N_2 remains constant then M also $\uparrow$. However, we cannot reason without a knowledge of the individual numerical changes when we are told that both N_1 and $N_2 \uparrow$ or $\downarrow$. The problem is solvable via symbolic and numerical coupling mentioned in Section 2.2. In any case, it is evident that we need symbolic computation aids to work with confluences.

In the pressure regulator example we are employing a powerful principle in our reasoning, viz. the *mechanistic* world view. This asserts that every physical situation is regarded as a device made up of individual components, each contributing to the overall behavior. However, the laws of the substructures of a structure may not presume the functioning of the whole—the principle of *no-function-in-structure* [27]. Additionally, assumptions that are specific to a particular device should be distinguished from the *class-wide assumptions* which are common to the entire class of devices [27]. A simplistic view of modeling devices comprises three kinds of constituents: *materials* (e.g. oil, air), *components* (e.g. shafts, wheels), and *channels* (e.g. water pipes, electric cables) [16, 27]. Channels transfer material from one point to another. It is assumed that a channel is always full of material and obeys the law of conservation.

After modeling a device we can reason about it. In *envisionment* we start with a structural description to determine all possible behavioral sequences [26]. Thus, in envisioning, we momentarily forget about the real values of the problem variables and try to see all possible outcomes of some action. Envisionment has close ties with *causation*. This relationship between two states of affairs such that the first brings about the second can be modeled using temporal logic operators such as α and β in a situational calculus setting [22]. As an example of partial envisioning, imagine a box OBJ, with a smaller block obj inside, sliding on a surface inclined to water. OBJ will slide down the inclined surface and reach its corner. Then it will fall and hit the water. If OBJ breaks then obj will be released to the water and will possibly sink. Modal logic

(cf. Section 4.1.1) comes handy to codify these facts. Thus, we may write statements like

$$on(OBJ, inclined\text{-}surface) \wedge \neg fixed(OBJ) \supset \blacklozenge slide(OBJ),$$

$$plunged(obj, water) \wedge a\text{-}kind\text{-}of(obj, sugar) \supset \blacksquare dissolve(obj).$$

Naive physics concepts are crucial for design because in many cases design objects will have a physical existence and accordingly obey natural laws. If we want to create designs corresponding to physically realizable design objects then we will have to refer to naive physics notions such as solids, space, motion, etc. Furthermore, if we want to reason about a design object in its destined environment (think of our pressure regulator to be installed in a nuclear reactor) we will need naive physics procedures such as envisioning, simulation, and diagnostics.

4.1.3 The Need to Study Thinking

Since IIICAD will be a tool for dealing with an intellectual process, we need theories that are directly related to human thinking. We should certainly study epistemological theories about our perceptions from philosophy, cognitive science, psychology, etc. since they contribute to our understanding of design. In database research [6], the counterpart of such knowledge is known as *conceptual modeling* and has been instrumental in establishing a sound basis for database management.

In the history of AI, there are many interesting debates on foundational matters such as *procedural vs. declarative* controversy and *logic vs. special knowledge representation* techniques [29]. Neither of these debates is settled yet; it is quite possible that there are no unequivocal answers. The need for coupled systems for symbolic and numerical computation suggests an appropriate mix of procedural and declarative languages. In case of knowledge representation, we accept that logic is syntactical and hence sterile in regard to meaning. However, it should not be forgotten that it is the best formalization man has ever built. Furthermore, it is possible to implement e.g. frames entirely in logic [21].

Theory of knowledge deals with issues of the preceding sort and is necessary especially to place design knowledge in a particular framework to utilize it in the IIICAD architecture. The following discussion is based on [43] and draws a distinction between two opposing knowledge representation methods, i.e. *extensional* vs. *intensional* descriptions. We do not here take up the philosophical [9] distinctions between these methods although that also is relevant to knowledge representation (and will be treated in a future paper).

In order to discuss design, we need to describe entities, their properties, and relationships among entities as mentioned in Section 4.1.1. In an extensional description method, the fact that an entity e has property p is described by $p(e)$ and the fact that entities e_1 and e_2 stand in relationship r is described by $r(e_1, e_2)$. In an intensional description method these two facts can be represented by $e(p)$ and $relation(e_1, r, e_2)$, respectively. Extensional descriptions do not imply any preconceptions while intensional descriptions make assumptions such as that e's property is limited to one particular p. When an intensional description has to be changed, this results in modifying those predefined conditions. For instance, a shaft might be represented intensionally by

$$shaft(diameter, length, bearing_1, bearing_2).$$

If we now want to add new attributes, such as transferring power, this results in a redefinition of the *shaft* predicate. On the other hand, an extensional description might consist of the following

facts:

$$shaft(s),$$

$$equal(diameter(s), D),\ equal(length(s), L),$$

$$supported\text{-}by(s, b_1),\ supported\text{-}by(s, b_2),$$

$$bearing(b_1),\ bearing(b_2).$$

In an extensional description we need to write numerous obvious descriptions. However, modifying such a representation is just adding or deleting facts. On the other hand, an intensional description might be difficult to change but shows better performance. In the previous example, it is easily guessed that the computation to determine two bearings supporting a shaft is reduced to an address calculation.

CAD applications request a flexible data description scheme which is easy to modify [15]. Independent multiple views of design object demand independent small partitionings in the database. This is easily achieved by an extensional description method since we only have to pick up the relevant facts. In an intensional description this requires to create a totally new scheme.

As discussed in Section 4.1.1 there are variants which change during the design process. The extensional description can be used to recount these variants. We know that logic programming [4] is useful to implement such description methods. On the other hand, we use invariants as building blocks in design. Such blocks do not change their structural properties although values of their attributes might change. Invariants, therefore, can be represented intensionally and their properties can be depicted as covariants.

4.2 Rapid Prototyping and CAD Software

In building knowledge based systems such as IIICAD, we are trying to develop a system which will be open-ended. IIICAD will evolve over time, with the definition of its purpose being refined as it becomes a fuller system. We therefore believe that in the development of intelligent CAD systems the underlying strategy should be "Plan to throw one away. You will anyhow." [7]. This suggests that recent techniques of software engineering such as exploratory programming [4] and rapid prototyping [36] are not to be overlooked. These techniques are more permissive than the rigid method of formal specification in that they advocate iterative enhancement. This leads to an evolutionary life cycle. We start with a skeletal implementation (rapid prototype) and add new modules until the system is reasonably complete for demonstration. In rapid prototyping we are interested in perceiving a glimpse of a future system in order to assess its strengths and weaknesses.

4.2.1 Why Smalltalk†?

With regard to exploratory programming and rapid prototyping we think that Smalltalk [19] is an excellent implementation tool. First, we appreciate the emphasis of Smalltalk on interactive graphics and favor its sophisticated user interface [20]. Second, Smalltalk is not just an isolated programming language; it is a programming environment. For instance, the ability to ''inspect'' any data structure recursively to any depth gives a programmer a great ability for exploratory programming.

Smalltalk embodies the innovative ideas and decade-long experience of several good researchers. From a philosophical viewpoint, Smalltalk's insistence to express everything as objects and messages using a simple syntax is conceptually very clean, powerful, and consistent. This, in our view, places it in the same category with other languages such as Lisp and Prolog which are the flagships of functional and logic programming, respectively.

Our experience with Smalltalk also suggests that it is not yet perfect. First and foremost, learning Smalltalk and using it effectively are not easy tasks. Although there is a natural associated learning curve [35], a common joke tells that it is easier to learn Smalltalk in an established Smalltalk-using organization where colleagues who are Smalltalk experts are available [34]. The methods hierarchy is simply too big; Smalltalk code that performs some function rarely resides in a well-defined location but is distributed all over the system. The concept of *information hiding* then becomes somewhat wearisome since we need to have an idea about how something works. Long chains of message passing and super- and subclass responsibilities make it difficult for the novice to understand how some action is taking place. This is perilous because it tends to de-emphasize one of the major goals of Smalltalk—encouraging the *reuse* of software. The scale of the system and its generalist design may frustrate even the professionals with a wide background in other programming languages. It is no wonder that Alan Kay considers Smalltalk no longer suited for children [34].

The class hierarchy is rigid. Especially, the inheritance mechanism, even when it is enriched with multiple inheritance, is seen by some researchers as a drawback for CAD systems [3]. (See, for example, the requirement for flexible data descriptions in Section 4.1.3.) Recently, a new scheme called *delegation* is being popularized for pliable sharing of knowledge in object oriented systems [30]. In its most common form, inheritance is a way to arrange classes in a tree so that they inherit methods from other classes. Delegation is more general; an object can delegate a message to an arbitrary other object rather than being limited to the paths of a class hierarchy (or lattice, in case of multiple inheritance).

4.2.2 Advantages of Loops

We close this section with a short overview of Loops [37], a language which is becoming popular due to its multiparadigm nature [38]. The reader is cautioned that we do not have hands-on experience with Loops. However, our first impression of Loops is that it offers several additional and interesting facilities to build intelligent CAD systems.

Loops supports a mechanism for *annotated values*. This helps programmer monitor arbitrary values without previously planning such access. There are two kinds of annotated values: property annotations and active values. The latter associate with any value a demon to be invoked when a data store or fetch is requested. The former associate with any value an optional property list to annotate the value with useful information.

From a CAD viewpoint, another useful construct of Loops is *composite object*. Composite objects are objects that contain other objects as parts. A car may be described structurally as consisting of a body, a mechanical system, and an electrical system. The body has four doors, six windows, etc. Parts can contain other parts; e.g. a door has an ashtray and a handle. Objects may belong to more than one structure; e.g. the power window controller can be viewed as a

† Smalltalk-80 is a registered trademark of Xerox Corporation. In the sequel, we shall simply write Smalltalk instead of Smalltalk-80.

component of the mechanical system or the electrical system.

Loops also offers *perspectives.* Perspectives are a form of composite object and provide a way to implement the multiple views of the same design object. Thus we may represent the concept "Pressure Regulator" using perspectives like *PR-As-A-Mechanical-Assembly, PR-As-A-Display-Object,* and *PR-As-A-Function-Box.* The first two perspectives both treat coordinates, but with different interpretations. In the first perspective the coordinates refer to physical dimensions of the regulator to be manufactured whereas in the second perspective they are just measures on a display screen. In the third perspective we are interested in the functioning of the regulator as a control-theoretic device with feedback. Perspectives should be compared with the Smalltalk concept of *model-view-controller triad,* a standard implementation strategy for user interfaces [40, 10].

Loops also offers a remedy to the complicated method hierarchy of Smalltalk (cf. Section 4.2.1). In Loops, a method is defined over a set of classes. This means that methods do not belong to a particular class.

5. IDDL as a Kernel Language for IIICAD

IDDL will be the kernel (base) language upon which the subsystems of IIICAD will be built. This section first summarizes the requirements for IDDL and then presents our prototype implementation. For the sake of brevity we do not justify our requirements here; this has been done in [44]. Besides, many of the requirements have rather limpid reasons behind them. It is also noted that the following description is incomplete; the current version of IDDL is fully described in [1] and is presently under reconstruction.

5.1 IDDL Requirements

We start with somewhat technical items. IDDL should deal with two kinds of names: system names are unique and internal whereas user-defined names are modifiable and external; this follows from our decision to use extensional descriptions as much as possible. Similarly, IDDL should make a distinction between the facts that an entity has an attribute and an attribute has a value. It should also be able to describe status and control information of the system with origins, destinations, and time stamps.

IDDL, as a language to construct knowledge bases, should support incremental programming and easy maintenance. It should be able to describe not only design objects but also design processes. It must embed the stepwise nature of the design process. It should be able to describe knowledge to detail a metamodel, to check its feasibility, and to control the detailing process. It should allow multiple views of a design object; these views are possibly independent but still correlated.

IDDL should let positive and negative information, known and unknown, and modalities such as necessity and possibility. Inconsistencies need to be resolved with some sort of truth maintenance system [12] when transferring to a next metamodel. This requires that IDDL should incorporate assorted logics such as modal, intuitive, and temporal. To control the behavior of the system, IDDL must have metaknowledge that selects which "rule" to apply at a certain moment. It should also have a focusing control mechanism which can create a small world where it is clearly defined what kind of information is accessible (Figure 7). From the viewpoint of interactive design, it is desirable for the human designer to be able to mark intermediate design stages, and later go back for examining or resuming from there.

Consequently, the stages of design evolution must be representable on the level of IDDL.

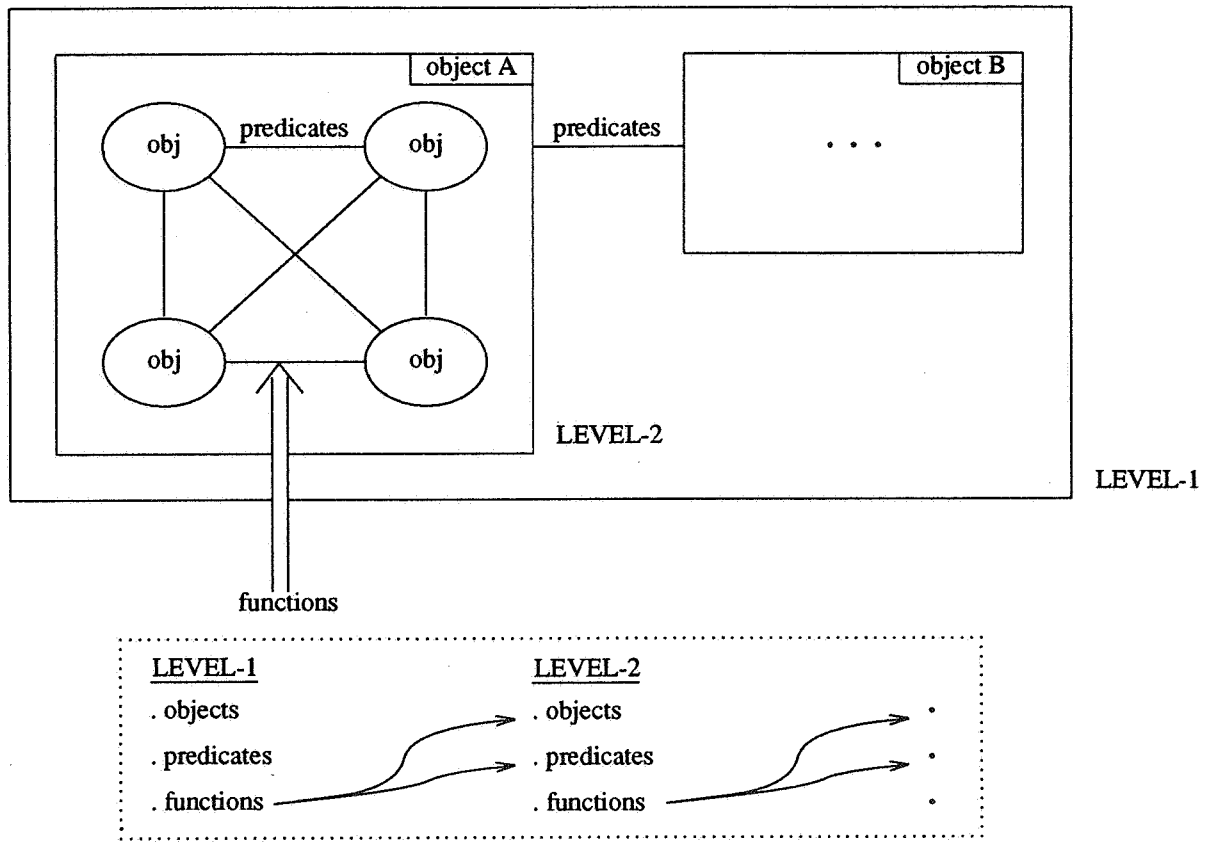

Fig. 7. Enclosure Mechanism for Data Encapsulation

Design produces intermediate results which are incomplete and even inconsistent during time spanning series of transactions. The design object's attributive representation must allow assumptions to be used for the evolution of the design object. IDDL, using nonmonotonicity, should be able to retract assumed (but later on unconsidered) propositions [32].

In spite of the slight implementation level weaknesses cited in Section 4.2.1, object oriented programming languages deliver extensibility, flexible modifications of code, and reusability. Important issues in object oriented programming are data encapsulation and information hiding. Thus an object can be regarded as an independent program which knows everything about itself. Reuse of software may be assisted by the class inheritance mechanism. It is an additional benefit of object oriented programming systems that they offer incremental compilation, dynamic binding, rich system building tools, and good user interfaces. Therefore object oriented programming is a good choice for creating IIICAD. On the other hand, logic programming is powerful for problem solving since it can reflect the reasoning process most naturally and directly. IDDL uses the logic programming paradigm to express the design process for *manipulating* design objects, whereas the object oriented programming paradigm is used to *express* design objects. In IDDL invariants, variants, and covariants will be represented respectively by objects, their internal and external relationships, and behavior of objects.

5.2 IDDL Prototype

Figure 8 shows a typical display from our prototype implementation of IDDL. We have developed this version on a Smalltalk machine. In Figure 8, design browsers are used to manipulate design information such as scenarios, constraints, predicates. In this version of IDDL, *constraints* correspond to the covariants of Section 4.1.1 and are meant to be the design specifications, while *predicates* correspond to the variants and are used for object description.

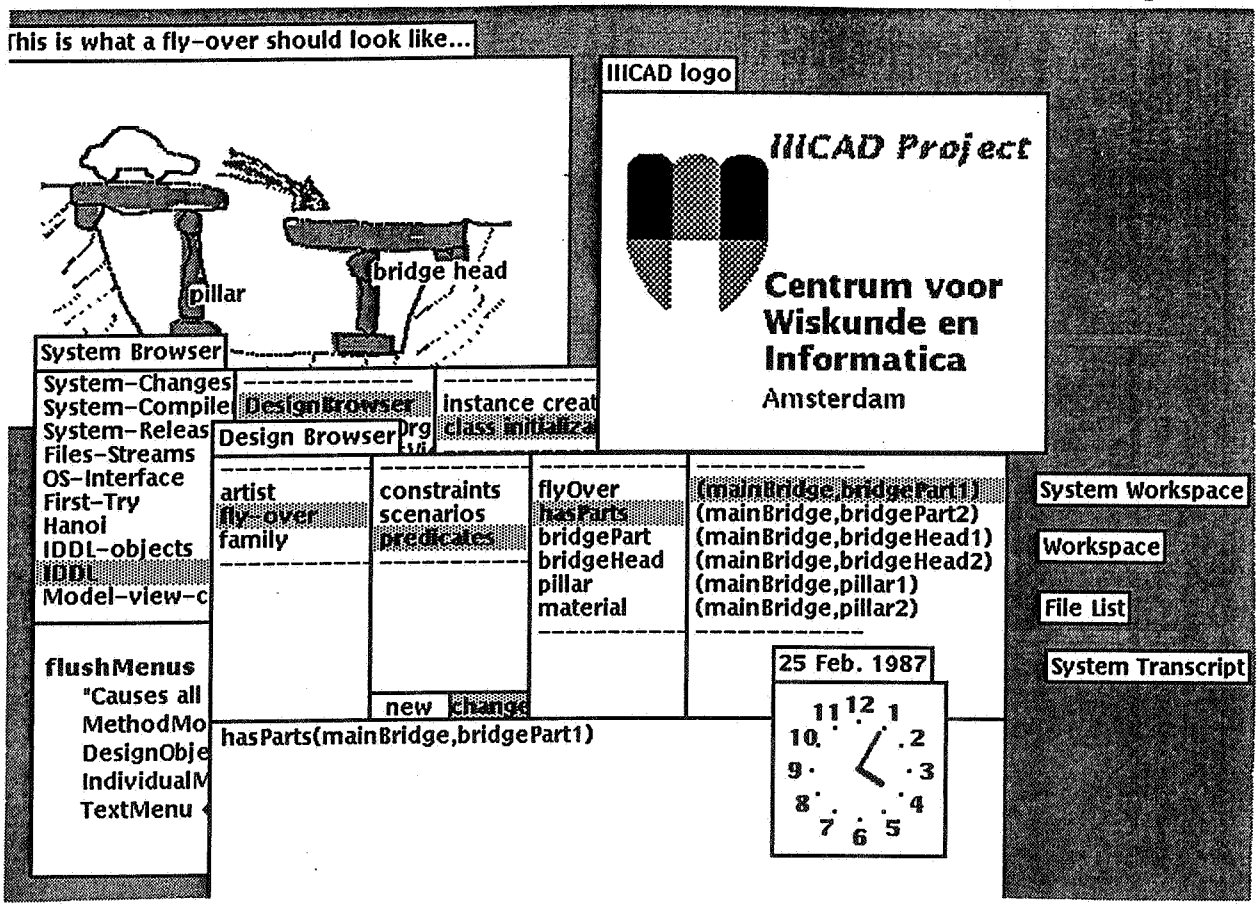

Fig. 8. Example Screen from IDDL Prototype

The two windows on the left of the figure are snapshots of the constraints for a bridge and its subparts. The other two overlapping windows on the right show the part-assembly relationships among substructures. The designer can explore various possibilities by communicating with the system through these windows. It is apparent that Smalltalk's satisfactory user interface will be of much use to build IIICAD's intelligent user interface.

6. Concluding Remarks

We presented a unifying framework to describe and use design knowledge. Our starting point, theory of CAD, allows us to formulate IDDL specifications (cf. Section 5.1). It enables us to understand, clarify, model, and formalize design processes and design knowledge in an intelligent CAD environment. We believe that theoretical ideas of knowledge engineering and naive physics, formal tools such as logic, and practical software techniques such as prototyping are essential components in constructing intelligent design systems.

The development of IDDL was given priority to the development of other subsystems of IIICAD. This is due to the fact that IDDL will be the kernel language of our system. The current goals of the project are then as follows:

- Implement powerful tools for IIICAD development.
- Construct a more complete version of IDDL [1].

Near future work includes:

- Develop subsystems of IIICAD such as SPV, API, and IUI.
- Incorporate existing CAD tools and knowledge based systems, including a qualitative reasoning expert, in the IIICAD framework.
- Justify the methodology of developing IIICAD and eventually prove its effectiveness by case studies.

Acknowledgments

We thank Monique Megens and Eric Weijers for their continuing efforts on IIICAD. Our project is funded by NFI.

The mention of commercial products in this paper does not imply endorsement.

References

1. B. Veth , "IDDL, an Integrated Data Description Language," CWI Report, Center for Mathematics and Computer Science, Amsterdam (1987, to appear).
2. A.M. Agogino and A.S. Almgren, "Symbolic computation in computer aided optimal design," in *Expert Systems in Computer Aided Design*, ed. J. Gero, North-Holland (1987, to appear).
3. F. Arbab, "A paradigm for intelligent CAD," in *Intelligent CAD Systems 1: Theoretical and Methodological Aspects*, ed. P. ten Hagen and T. Tomiyama, Springer-Verlag, Heidelberg (1987, to appear).
4. D. Bobrow, "If Prolog is the answer, what is the question? or What it takes to support AI programming paradigms," *IEEE Trans. Software Engineering* **11**(11), pp. 1401-1408 (Nov. 1985).
5. D. Bobrow, S. Mittal, and M. Stefik, "Expert systems: Perils and promise," *Comm. ACM* **29**(9), pp. 880-894 (Sept. 1986).
6. M.L. Brodie, J. Mylopoulos, and J.W. Schmidt (ed.), *On Conceptual Modeling*, Springer-Verlag, New York (1984).
7. F.P. Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading, Mass. (1975).
8. D.C. Brown and B. Chandrasekaran, "Knowledge and control for a mechanical design expert system," *IEEE Computer* **19**(7), pp. 92-100 (July 1986).
9. R. Carnap, *Meaning and Necessity: A Study in Semantics and Modal Logic*, The Univ. of Chicago Press, Chicago, Ill. (1947).
10. W. Cunnigham, "The construction of Smalltalk-80 applications," Working Paper, Tektronix Computer Research Lab, Beaverton, Oregon (Nov. 1985).
11. B.T. David, "Multi-expert systems for CAD," in *Intelligent CAD Systems 1: Theoretical and Methodological Aspects*, ed. P. ten Hagen and T. Tomiyama, Springer-Verlag, Heidelberg (1987, to appear).
12. J. Doyle, "A truth maintenance system," *Artificial Intelligence* **12**, pp. 231-272 (1979).
13. J. Doyle, "Expert systems and the 'myth' of symbolic reasoning," *IEEE Trans. Software Engineering* **11**(11), pp. 1386-1390 (Nov. 1985).

14. M.G. Dyer, M. Flowers, and J. Hodges, "EDISON: An engineering design invention system operating naively," *Artificial Intelligence in Engineering* **1**(1), pp. 36-44 (1986).
15. J. Encarnacao and F.L. Krause, *File Structures and Databases for CAD*, North-Holland, Amsterdam (1982).
16. K. Forbus, "Qualitative process theory," *Artificial Intelligence* **24**, pp. 85-168 (1984).
17. J.S. Gero (ed.), *Knowledge Engineering in Computer Aided Design*, North-Holland, Amsterdam (1985).
18. J.S. Gero (ed.), *Expert Systems for Computer Aided Design*, North-Holland, Amsterdam (1987).
19. A. Goldberg and D. Robson, *Smalltalk-80: the Language and its Implementation*, Addison-Wesley, Reading, Mass. (May 1983).
20. A. Goldberg, *Smalltalk-80: the Interactive Programming Environment*, Addison-Wesley, Reading, Mass. (1983).
21. P. Hayes, "The logic of frames," pp. 46-61 in *Frame Conceptions and Text Understanding*, ed. D. Metzing, Walter de Gruyter and Co., Berlin (1979).
22. P. Hayes, "The second naive physics manifesto," pp. 1-36 in *Formal Theories of the Commonsense World*, ed. J. Hobbs and R. Moore, Ablex, Norwood, New Jersey (1985).
23. F.R.A. Hopgood, D.A. Duce, J.R. Gallop, and D.C. Sutcliffe, *Introduction to the Graphical Kernel System (GKS)*, Academic Press (1983).
24. R. Jacquart, P. Regnier, and F.R. Valette, "GERMINAL: Towards a general and integrated system for computer aided design," *Proc. 11th Design Automation Workshop*, Denver, Colo., pp. 352-358 (June 1974).
25. C.T. Kitzmiller and J.S. Kowalik, "Symbolic and numerical computing in knowledge based systems," pp. 3-17 in *Coupling Symbolic and Numerical Computing in Expert Systems*, ed. J.S. Kowalik, Elsevier, Amsterdam (1986).
26. J. de Kleer, "Qualitative and quantitative knowledge in classical mechanics," Report AI-TR-352, MIT Artificial Intelligence Lab, Cambridge, Mass. (Dec. 1975).
27. J. de Kleer and J.S. Brown, "A qualitative physics based on confluences," *Artificial Intelligence* **24**, pp. 7-83 (1984).
28. B. Kuipers, "Qualitative simulation," *Artificial Intelligence* **29**, pp. 289-338 (1986).
29. H.J. Levesque and R.J. Brachman, "A fundamental tradeoff in knowledge representation and reasoning," pp. 42-70 in *Readings in Knowledge Representation*, ed. R.J. Brachman and H.J. Levesque, Morgan Kaufmann Pub. Inc., Los Altos, Calif. (1985).
30. H. Lieberman, "Delegation and inheritance: Two mechanisms for sharing knowledge in object oriented systems," in *3eme Journees d'Etudes Langages Orientes Objects*, ed. P. Cointe, AFCET, Paris (1986).
31. R. van Liere and P. ten Hagen, "Introduction to dialogue cells," Report CS-R8703, Center for Mathematics and Computer Science, Amsterdam (Jan. 1987).
32. D. McDermott, "Nonmonotonic logic II: Nonmonotonic modal theories," *Journal of ACM* **29**(1), pp. 33-57 (Jan. 1982).
33. S. Mittal, C.L. Dym, and M. Morjaria, "PRIDE: An expert system for the design of paper handling systems," *IEEE Computer* **19**(7), pp. 102-114 (July 1986).
34. J. Nielsen and J.T. Richards, "Comments on the learnability and usability of Smalltalk for casual users," Report RC-12080 (#53339), IBM Thomas J. Watson Research Center, Yorktown Heights, New York (April 1986).
35. D.J. Penney, "Implementing a Smalltalk-80 file system and the Smalltalk-80 system as a programming tool," pp. 287-297 in *Smalltalk-80: Bits of History, Words of Advice*, ed. G. Krasner, Addison-Wesley, Reading, Mass. (June 1984).
36. C. Ramamoorthy, S. Shekhar, and V. Garg, "Software development support for AI programs," *IEEE Computer* **20**(1), pp. 30-40 (Jan. 1987).
37. M. Stefik and D. Bobrow, "Object oriented programming: Themes and variations," *AI Magazine* **6**(4), pp. 40-62 (Winter 1986).

38. M. Stefik, D. Bobrow, and K. Kahn, "Integrating access oriented programming with a multiparadigm environment," *IEEE Software* 3(1), pp. 10-18 (Jan. 1986).

39. I.E. Sutherland, "SKETCHPAD—A man-machine graphical communication system," *Proc. Spring Joint Computer Conference* (1963).

40. Tektronix, Inc., *4404 Artificial Intelligence System: Introduction to the Smalltalk-80 System*, Computer Research Lab, Beaverton, Oregon (1986).

41. P. ten Hagen and T. Tomiyama (ed.), *Intelligent CAD Systems 1: Theoretical and Methodological Aspects*, Springer-Verlag, Heidelberg (1987, to appear).

42. T. Tomiyama and H. Yoshikawa, "Extended general design theory," pp. 95-130 in *Design Theory for CAD*, ed. H. Yoshikawa and E.A. Warman, North-Holland, Amsterdam (1987).

43. T. Tomiyama and P. ten Hagen, "Representing knowledge in two distinct descriptions: Extensional vs. intensional," CWI Report, Center for Mathematics and Computer Science, Amsterdam (1987, to appear).

44. B. Veth, "An integrated data description language for coding design knowledge," in *Intelligent CAD Systems 1: Theoretical and Methodological Aspects*, ed. P. ten Hagen and T. Tomiyama, Springer-Verlag, Heidelberg (1987, to appear).

45. H. Yoshikawa and E.A. Warman (ed.), *Design Theory for CAD*, North-Holland, Amsterdam (1987).

46. P. Wegner, "Capital-intensive software technology," *IEEE Software* 1(3), pp. 7-45 (July 1984).

47. R.W. Yeomans, A. Choudry, and P. ten Hagen, *Design Rules for a CIM System*, North-Holland, Amsterdam (1985).