Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

V. Akman, P.J.W. ten Hagen, J.L.H. Rogier, P. Veerkamp

Knowledge engineering in design

Computer Science/Department of Interactive Systems Report CS-R8745 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.).

Knowledge Engineering in Design

Varol Akman, Paul ten Hagen, Jan Rogier, Paul Veerkamp

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

ABSTRACT

We present in a unifying framework the principles of the IIICAD (Intelligent, Integrated, and Interactive Computer-Aided Design) system. IIICAD is a generic design apprentice currently under development at CWI. IIICAD incorporates three kinds of design knowledge. First, it has general knowledge about the stepwise nature of design based on a set-theoretic design theory. Second, it has domain-dependent knowledge belonging to the specific design areas where it may actually be used. Finally, it maintains knowledge about the previously designed objects; this is somewhat similar to software reuse. Furthermore, IIICAD uses AI techniques in the following areas: (i) formalisation of design processes; extensional vs. intensional descriptions; modal and other nonstandard logics as knowledge representation tools, (ii) common sense reasoning about the physical world (naive physics); coupling symbolic and numerical computation, (iii) integration of object-oriented and logic programming paradigms; development of a common base language for design.

Categories and Subject Descriptors: I.2.3 **[Artificial Intelligence]:** Deduction and Theorem Proving — *logic programming, nonmonotonic reasoning and belief revision;* I.2.4 **[Artificial Intelligence]:** Knowledge Representation Formalisms and Methods — *predicate logic, representation languages, representations (procedural and rule-based);* J.6 **[Computer-Aided Engineering]:** *computer-aided design (CAD)*

General Terms: Design, Languages

Additional Key Words and Phrases: nonstandard logics, object-oriented programming, naive physics, extensional/intensional descriptions

---

The authors are members of Group *Bart Veth* which additionally includes Peter Bernus (CWI) and Tetsuo Tomiyama (University of Tokyo). Acknowledgements are made to NFI for its financial support and to Tomiyama, Monique Megens (CWI), and Eric Weijers (CWI) for their invaluable contributions to the work reported in this paper. A preliminary version of this paper was presented at *Second World Basque Conference on Artificial Intelligence,* Basque Country, Spain (Sept. 1987).

Report CS-R8745
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

"In this paper, I have tried to argue that there is an important class of problems in knowledge representation and commonsense reasoning, involving incomplete knowledge of a problem situation, that so far have been addressed only by systems based on formal logic and deductive inference, and that, in some sense, probably can be dealt with only by systems based on logic and deduction." [22]

1. Introduction

Recent research in Computer-Aided Design (CAD) (cf. [36, 9, 8, 28] for some representative articles and [17] for a good, nontechnical review) has shown a visible interest in the *intellectualisation* of design. We find this a healthy trend since we believe that the ultimate aim of CAD is the automation of the several knowledge-intensive activities performed today solely by highly-specialised, hard to come by, expensive, and unfortunately error-prone human experts. In fact, it is not too early to claim that manufacturers are already very interested in "intelligent CAD" since several useful research efforts have been made, e.g. Briggs & Stratton's Engineering Design Assistant [25]. A common view goes like this: "Right now there are several designers who know a little about all facets of engine design, but there is no individual who could effectively design an entire engine. Eventually, however, it may be possible for one person, using a collection of expert systems, to do a considerable amount of the engine design process." [*ibid.*]

Intelligent CAD is practiced mainly by applying the already existing ideas of Artificial Intelligence (AI) and Knowledge Engineering (KE) in several aspects of the design process/object modeling. In addition to making CAD more intelligent, this approach has the advantage that while studying the appropriate AI and KE techniques, CAD researchers contribute to the existing body of knowledge in these areas.

The need to make CAD more sophisticated is real and being felt today especially in the high-tech domains. As technologies advance, the management of the complexity as a result of the amount and variety of information to be handled by design systems is becoming a gigantic task. It is hoped that advances in AI will be a keystone in bringing promising long-term solutions to structuring this potentially explosive domain of knowledge. The weaknesses of the existing CAD systems today are basically due to the facts that (i) they have no task-domain knowledge to reflect the thinking processes, terminology, and intentions of designers, and (ii) the system software is written in an unstructured, *ad hoc*, and hard to maintain/upgrade way, with no sound basis (in terms of a formal design theory).

We see intelligent CAD as a theory resting on a triad: a theory of knowledge, a theory of design processes, and a theory of design entities (objects). Our work at CWI is aimed at contributing to these theories and evaluating their usefulness through prototype implementations treating real design problems. We should immediately add that we are not interested in "expert systems for design" *per se* although we consider them as a part of the grand picture [2].

Clearly, the proposition that one has a "theory" to deal with design is rather pretentious and even dangerous. We are aware of the fact that design is a "mysterious" activity which is currently done in its entirety only by intelligent human designers. Yet, to quote Lansdown [17]:

> "[I]n broad terms, most people would accept that designing is a cyclical process in which concepts are devised and then tested against some criteria of performance, cost, or appearance. The tests: logical, physical, or just intuitive, lead to the concepts either being incorporated into the design or being rejected. In any event, the testing process gives rise to the formulation of new concepts and, importantly, then to new criteria for

testing. The whole of designing thus is governed by what Ernst Gombrich calls "schema and correction" — almost a trial and error process where experimentation precedes correction which in turn leads to further experimentation."

We appreciate the difficulty of identifying and incorporating all the planning, heuristic, and inventive knowledge that good designers tend to have. (Cf. [3,21,7] for several researchers' views on the various aspects of design.) Nevertheless, the issue here is mainly that of a formal language to "communicate" our results to the outside world. We believe that even in the vague domain of design where any kind of formalisation would probably look superficial, a formal outlook is the only way to do scientific research. We see logic as the essential framework of this formal outlook. First, logic is precise and unambiguous with a well-understood semantics that connects the formulas of logic and the real world that they talk about. Second, in its purity, logic provides a high level abstraction since it is entirely nonprocedural. It also acts as a formal specification since the knowledge is not buried in procedures. This issue is of substantial help in writing software engineered CAD code.

The organisation of this paper is as follows. In §2, we briefly look at a logical formalisation of the design processes. In §3, knowledge representation issues in design are reviewed from the angle of intensional vs. extensional descriptions. A theory of design entities based on naive physics and coupled systems is summarised in §4. Combining object-oriented and logic programming styles to arrive at a design base language is studied in §5. Finally, §6 summarises the key propositions of our approach and suggests future directions.

This paper is a partial overview of our research. The reader is referred, in addition to several other foundational articles by Tomiyama and Yoshikawa to be referenced later, to [33,34,35,29,30,32] for a detailed exposition of our work.

2. A Logical Formalisation of Design Processes

2.1. The Stepwise Nature of Design

We use General Design Theory [31] as a basis for formalising design processes and design knowledge. The theory is based on axiomatic set theory and models design as a mapping from the *function* space where the specifications are described in terms of functions, onto the *attribute* space where the design solutions are described in terms of attributes. Roughly speaking, one starts with a functional specification of the design object and ends with a manufacturable description encompassing all its attributes.

The basic ideas behind a logical formalisation of design processes are as follows:

- From the given functional specifications a candidate is selected and refined in a stepwise manner until the solution is reached, rather than trying to get the solution directly from the specifications.
- Hence design can be regarded as an evolutionary process which transfers the model of the design object from one state to another. We call this model, being a set of attributive descriptions, a *metamodel.* (This rather confusing and uninformative name is kept for historical reasons.)
- During the design process new attributive descriptions will be added (and some existing ones will be modified) and the metamodel will hopefully converge to the solution. Since dead ends are natural occurrences in design, a technique to step back and select more

promising paths at any time is also required.

- To evaluate the current state of the design object (i.e. the metamodel), various kinds of models of the design object need to be derived from the metamodel in order to see whether the object satisfies the specifications or not. We call those models of the design object *worlds* and they can be regarded as interpretations of the design object seen from certain points of view — the concept of "multi-worlds." (In machine design, one such model would be the finite element model of the design object, for instance.)

Considering the metamodel evolution model, the system starts from the specifications, s, of the design object and continues with the design process until the goal, g, is reached. We define q_j^i as the set of propositions at the state of metamodel M^i with an interpretation in world w_j. In other words, if we have m worlds then $q_1^i, ..., q_m^i$ constitute the current state of knowledge about the design object. There are two possibilities: either the current state of knowledge is complete and consistent or there is an incompleteness/inconsistency. In the first case, g has been reached and we have finished the design process. In the latter case, there is a need to proceed to the next metamodel M^{i+1} in order to resolve the incompleteness or inconsistency. (Note that we don't care about the inconsistency of a *particular* world.)

The nature of design then, understood in the above sense, is to modify/add properties about the design object. This means that we need language constructs to evaluate a metamodel by creating worlds and to derive new properties or to update uncertain/unknown properties in such a world in order to get more detailed knowledge about the design object. The crucial point is how to proceed from M^i to M^{i+1}. Alternatively, we can pose the following questions. How do we define q_j^i? How do we derive new information from the current world and compare different worlds? It has to be realised that we are not aiming at an automated design environment; our system is meant to be a *designer's apprentice*. The designer should take the initiative for directing the design process. This is where the interactive nature of design comes into play. Accordingly, the designer, regarding a certain world, can modify/add attributes about the design object. The system evaluates the metamodel after these updates and checks it for consistency.

The reader may notice that it seems natural to choose modal logic as a representational language since modal logic deals with interpretations of a model (understood in the "logical" sense) in multiple worlds (again understood in the logical sense). We don't elaborate on the relationship between the meanings we attach to "model," "world" and the usual interpretation of these words as employed in modal logic, cf. [12] for details.

Design, at the highest level, is accomplished in IIICAD by interacting with the so called *scenarios* which are (conceptually) frame-like structures describing standard design procedures. The classical definition of frames is "... a data structure for representing a stereotypical situation like being in a certain kind of living room or going to a child's birthday party" [20]. Just like frames, scenarios have information about the design objects/processes that play a role in stereotypical design situations as well as the various relationships between these stored information. Each scenario tells something about the way it is to be used and gives clues as to what to do if something goes wrong with the current design while it is active. The notion of default values for slots has the counterpart "assumed" attribute values for design entities. A scenario base is then a collection of scenarios structured in terms of some organisational principles. The following principles are well-known [23]:

- *Classification/Generalisation:* One can associate a scenario with its generic type. Thus a scenario to design e.g. a bicycle lock belongs to the generic type "locks." The

generalisation relation between types is a partial order (lattice) and structures types into an *isa-hierarchy*. An isa-hierarchy provides the means for the overall organisation and management of a large scenario base. Additionally and more technically, isa-hierarchies reduce the storage requirements by allowing properties associated with general objects to be passed to more specialised ones.

- *Aggregation:* This relates a scenario to its components (parts). Aggregation can be applied recursively to represent the parts of the parts. For example, the parts of a bridge are its toll booths, supports, traffic lights, pavements, etc. In this case, different "subscenarios" would be used to design the overall bridge; they would, most conveniently, be activated by their mother scenario. Notice that, a bridge can also be viewed as an abstract object with an address, a state highway classification number, an architectural style, a maintenance cost-per-year, etc. Regarding design chiefly as a *geometric* activity has been the classical pitfall of the CAD systems and we want to take heed of that.

Scenarios, like frames, allow other looser principles such as the notion of "similarity" between two scenarios. The easiest way to do this would be pattern matching, cf. [20] for a detailed exposition of frame similarity and additional techniques to achieve it.

2.2. Modal and Other Nonstandard Logics for Design

Modal logic can be seen as the logic of *necessity* and *possibility* [12]. We will show the basic notions that a system of modal logic is intended to express. We use the conventional notation for the modal operators, necessary and possible, and introduce new notation for the default and unknown operators.

Among true propositions we can distinguish between those which are merely true and which are bound to be true. Similarly, we can distinguish among false propositions between those which are bound to be false and those which are merely false. A proposition which is bound to be true is called a necessary proposition ($\mathrm{N}p$, it is necessary that p); one which is bound to be false is called an impossible proposition ($\mathrm{N}\neg p$, it is impossible that p). If a proposition is not impossible we call it a possible proposition ($\mathrm{P}p$, it is possible that p). We have now informally introduced the monadic proposition forming operators N and P. These operators are not truth-functional, i.e. the truth value of the proposition cannot be deduced even when the truth value of the argument is given. However, a strategy exists to determine the validity of a necessary or possible proposition. We won't give the exact definition of this validity checking but describe it informally.

A necessary proposition, $\mathrm{N}p$, is valid in a certain world iff p is valid in all worlds *accessible* to that world. A possible proposition, $\mathrm{P}p$, is valid in a certain world iff p is valid in one or more world(s) accessible to that world. Briefly, a world W_2 is accessible to a world W_1 if W_2 is conceivable by someone living in W_1. Consider the following example. We can conceive a world without telephones but if there had been no telephones, it would be the case that in such a world no one would know about what a telephone was and so no one would conceive of a world (e.g. ours) in which there are telephones [12]. More technically, suppose that we have a set of propositions. We can specify what the state of a world is by giving a list of which propositions are true and which are false according to this world. Let Ω be a dyadic reflexive relation over the worlds. Then Ω is called the accessibility relation, i.e. world w_i is accessible by w_j iff $w_i \ \Omega \ w_j$.

We use a different operator D to express *default* values. Thus, $\mathrm{D}p$ means that p is consistent with the theory. A proposition is consistent if its negation cannot be derived within the

theory. A default proposition, $\mathbf{D}p$, is considered to be valid if $\neg p$ cannot be proved. With this mechanism, we have the possibility to deal with nonmonotonicity [19]. During the design process, some properties about the design object may not yet be known; so we can assume some default values. But as soon as contradictory information is derived, we discard the default property and base things on the newly obtained information. Notice that this is nothing but the well-known *truth-maintenance* problem [6].

The modal operator **U** is used to denote uncertainty. A proposition is *unknown* if neither its truth nor its falsity can be derived. An unknown proposition, $\mathbf{U}p$, is considered to be valid if neither p nor $\neg p$ can be found. Note that we now actually have introduced a third truth value (i.e. unknown). The reason we avoid explicitly introducing a third truth value is that we want to keep our logic as simple as possible. This further implies that we have the *open world* assumption [23]. Nevertheless, if we request p the knowledge base must return false if it finds $\neg p$ or cannot find p. Therefore, $\mathbf{U}p$ is required to know about the uncertainty of p.

2.3. Incomplete Information and Null Values

Several ideas to be mentioned in this subsection owe their origin to recent research in databases. We'll follow especially [18, 24] closely, for they too insist on using logic as a framework for databases.

Since our envisioned design system will be based on KE principles, the existence of a knowledge base (KB) is implicit as an integral part. Whatever supervisory mechanism (SPV) we'll have in the system, it would like to query the KB about a particular design application. In design, any KB would be incomplete since it is impossible to identify and store all the information necessary to answer a query. In this case, we should distinguish between what the KB knows and what the truths are in the design domain. A KB may know that a shaft is attached to a motor without knowing the motor's power rating; it may know that one of the cylinders of the motor is faulty without knowing which one. Thus, one cannot treat a design KB as a realistic replica of the application domain. Since design is an open-ended activity, it may turn out that design KBs will never stabilise and one should find ways to deal with this ever-changing character of them.

Assuming that logic is the underlying formalism, for each query κ there are four possibilities: (i) **true** when κ can be inferred from the KB, (ii) **false** when $\neg\kappa$ can be inferred from the KB, (iii) **unknown** when neither (i) nor (ii) holds, and (iv) **contradiction** when both (i) and (ii) hold. We call a KB *consistent* if it contains no contradictory information. In an *incomplete* KB, on the other hand, we may pose queries which have **unknown** as answer. Unknown information may be in several disguises. Consider the following example. We know that "Door D 0023 has a type L 0003 or L 0014 lock," but don't know which. A straightforward way to represent this fact is to have two interpretations of D 0023: one with L 0003, the other with L 0014. This type of unknown is known in the database area as *disjunctive information.* Another common and more challenging unknown is the *null value,* meaning "value at present unknown." Now, if we accept the *closed world* assumption (viz. the negation of any atomic formula can be inferred from the inability to infer the atomic formula, a.k.a. *negation-by-failure* in Prolog) then solving the null value problem is easy since it reduces to the disjunctive information problem with the disjuncts expressing all the possible values (collected from the KB). However, under the open world assumption the value will not necessarily be one of the some finite set of known possible values. Consider the following. We know that "Pipe P 0254 feeds an oil tank" but don't know which. Moreover, this tank may or may not be one of the known tanks T 0001 and T 0002. In first

order theory, we would express this as

$$\exists X,\ oil\text{–}tank(X) \wedge fed\text{–}by(X,\ P0254),$$

then choose a name, ω, for this object and rewrite the preceding as

$$oil\text{–}tank(\omega) \wedge fed\text{–}by(\omega,\ P0254).$$

In fact, ω has long been known in the logic terminology as a *Skolem constant* and provides a way to eliminate the $\exists$ sign in proof theory; databases introduced the more suggestive name null values. It is important to observe that each time a new null value is introduced to the KB, it should be denoted by a new name (distinct from all other names). Thus, the switch below to $\hat{\omega}$ is compulsory to express "Some tank (maybe the same one as $T0001$ and $T0002$) is fed by pipe $P0789$":

$$oil\text{–}tank(\hat{\omega}) \wedge fed\text{–}by(\hat{\omega},\ P0789).$$

In addition, the KB should be made aware of the existence of a null value in general. This means that the allowable entities (e.g. the universe made of $P0254$, $T0001$, $T0002$ in the first example) must be expanded by introducing ω and the axioms should be revised as

$$\forall X,\ [oil\text{–}tank(X) \supset X \equiv T0001 \vee X \equiv T0002 \vee X \equiv \omega]$$

and

$$\forall X, Y,\ [fed\text{–}by(X, Y) \supset$$

$$(X \equiv T0001 \wedge Y \equiv P0254) \vee (X \equiv T0002 \wedge Y \equiv P0254) \vee (X \equiv \omega \wedge Y \equiv P0254)].$$

2.4. Other Nonstandard Logics

Predicate calculus of higher order is useful to talk about inheritance. The following is provable in the second order predicate logic: $\forall F,\ [F(x) \supset G(x)]$. (If an individual x has every property then x has any property G.) In third order predicate logic, we can prove that $\forall F,\ [V(F) \supset V(G)]$. (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 can be written:

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

$$t\ \alpha\ (p \wedge q) = (t\ \alpha\ p) \wedge (t\ \alpha\ q),$$

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

Using temporal logic, we can describe inference control for our system in a more explicit way. For instance, in Prolog the order of rules matter [1]. 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 for a design object may be a set of rules of the sort

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

$$(t_1 \; \alpha \; q_1) \wedge (t_2 \; \alpha \; q_2) \wedge (t_2 < t_1) \supset t_2 \; \alpha \; q_4.$$

where q_i's are understood in the sense of §2.1.

Intuitionistic logic can also be incorporated into temporal logic. Let t_p be the time when proposition p is proved. By definition, we have $t_p \; \alpha \; p$ = **true**. Now, using the logical symbol **unknown** we can formalise intuitionism in terms of temporality:

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

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

Once we extend the first order predicate logic with these operators, we have a powerful notation to describe design knowledge in a flexible manner. Since a design object is constantly updated during the design process, we need to describe it in a dynamic way. The constructs we have envisioned above work with a multiworld mechanism realised in modal logic. This mechanism helps the designer describe a design object seen from several viewpoints and express default and uncertain information about a design object.

3. The Method of Extensions/Intensions

3.1. Philosophical Origins

We begin with a philosophical discussion about knowledge representation and then move to more concrete issues. We'll start with a definition of *L-truth,* a notion also known as logical truth, necessary truth [Leibniz], and analytic truth [Kant]. The subject matter is historical and treated in great detail in [4]. Call a sentence, σ, L-true in a system, Σ, iff σ is true in Σ in such a way that its truth can be established on the basis of the rules of the system Σ alone, without reference to (extra-linguistic) facts. This is, in a sense, what Leibniz meant when he stated "A necessary truth must hold in all possible worlds."

It is customary to regard two *classes,* say those corresponding to the predicates p and q, identical if they have the same elements (e.g. p and q are equivalent). By the *intension* of the predicate p we mean the property p; by its *extension* we mean the corresponding class. The term *property* is understood in an objective (physical) sense, not in a subjective, mental sense. Thus "red" table should mean that the colour of the table (as understood, in the final analysis, as a physical property) is red, not that the person who is looking at it perceives it (for some e.g. psychological reason) as red. Thus, one may state that the table has the character Red whereas the observer has the character Red-Seeing. An good account of intensions and extensions is given in the following passage:

> "Class may be defined either extensionally or intensionally. That is to say, we may define the kind of object which is a class, or the kind of concept which denotes a class: this is the precise meaning of the opposition of extension and intension in this connection. But although the general notion can be defined in this two-fold manner, particular classes, except when they happen to be finite, can only be defined intensionally, i.e. as the objects denoted by such and such concepts. I believe this distinction to be

> purely psychological: logically, the extensional definition appears to be equally applicable to infinite classes, but practically, if we were to attempt it, Death would cut short our laudable endeavour before it had attained its goal. Logically, therefore, extension and intension seem to be on a par" [Bertrand Russell].

For example, let S denote that something is a shaft and let L denote that something is two miles long. The conjunction $S \wedge L$ would mean that something is a shaft and two miles long — denoting an empty yet not meaningless class. On the other hand, $S \wedge \neg S$ would mean shaft and at the same time not shaft — an L-empty statement. No factual knowledge is required for recognising the fact that the last conjunction cannot be exemplified.

The method of intensions/extensions has its roots in the work of Frege who studied it in a less rigorous way and called it the method of *name-relation.* This consists of regarding expressions as names of (concrete or abstract) entities in accordance with the following principles:

- Every name has exactly one entity named by it, i.e. its *nominatum.*
- Any sentence speaks about the nominata of the names occurring in it.
- If a name occurring in a true sentence is replaced by another name with the same nominatum, the sentence remains true.

If the last principle is applied without restriction, contradictions may arise. Frege's solution was to draw a distinction between the nominatum and the "sense" of an expression. A classical example is the two expressions "the morning star" and "the evening star." Although these expressions have the same nominatum they certainly don't have the same sense. It will be seen that nominatum resembles to extension and sense resembles to intension. (In fact, John Stuart Mill used the more descriptive terms denotation and connotation, respectively, for the above concepts.)

3.2. Describing Design Entities

We find the main use of intensions/extensions in describing design objects. Suppose that we are trying to describe a pressure regulator. Normally, we would have a "method" which knows about a certain type of pressure regulator, possibly parametrised so that one can create (i.e. design) instances of it by changing the parameters. Thus, e.g.

$$pres\text{-}reg\,(max\text{-}pressure\,,\ max\text{-}deviation\,,\ input\text{-}area\,,\ output\text{-}area\,,\ valve\text{-}type\,,\ \cdots)$$

would be the way this method can be invoked. The suggestion is to visualise this as an intensional description. The method *pres-reg* comprises all the information one would require to deal with this kind of regulator. In that sense, it *embodies* the concept of a regulator. This makes it efficient in terms of design time since all the knowledge is there and one simply has to make the right invocation of this method. On the other hand this intensional description is inflexible since if one now wants to add a new "parameter," e.g. *max-fluid-viscosity,* one would face the problem of studying and changing the whole method.

An extensional description of the same regulator is a collection of facts of the sort:

$$pres\text{-}reg\,(PR)$$

$$input\text{-}area\,(PR\,,\ area_{in})$$

$$output\text{-}area\,(PR\,,\ area_{out})$$

$$max\text{–}deviation\,(PR\,,\,max\text{–}dev\,)$$

$$max\text{–}pressure\,(PR\,,\,max\text{–}pres\,)$$

$$valve\text{–}type\,(PR\,,\,v\text{–}type\,)$$

. . .

and some procedural knowledge to structure these. Now, adding the new fact would be just the addition of the new piece of information *max-fluid-viscosity(PR, max-vis)*. Obviously, the procedural parts should also change but this change is thought to be less and much more "local." It is not difficult to see, on the other hand, that this new method of describing the regulator suffers from inefficiency since there are several facts which should be combined in some way, viz. the available information is in bits and pieces and should be put together.

For design, the advantage of extensional descriptions should be clear. In design, we need an integrated set of models each of which represents a different facet of the design object and possibly changes during the design process. From this viewpoint, intensional descriptions are very rigid and data exchange between two different say, solid modeling systems based on these description methods may suffer from loss of information or twist of meaning.

4. Naive Physics: A Theory of Design Objects?

We'll keep this section short since we are in the process of preparing a longer reply to the question posed in the section title.

4.1. Expressing Naive Physics Knowledge

Since its inception by Patrick Hayes a decade ago [11], Naive Physics (NP) has established itself as an exciting branch of AI. The aim of NP is to represent and simulate the knowledge and thought processes humans have about the physical world. A good example is attributed to Marvin Minsky: "You can pull with a string but not push with it." While we possess such trivial knowledge it is exceedingly difficult to have computers appreciate and use it. Giving such common sense physical knowledge to computers is essentially the aim of NP.

An integral part of NP is Qualitative Reasoning (QR) about the physical processes. This can best be explained with an example. Consider a sealed container full of water. If it is subjected to heat, it will eventually explode. The process that gives rise to this is the transformation of water into steam which applies huge forces. In this style of reasoning we are not really interested in the nuts and bolts of what is going on, i.e. we are hardly interested the "exact" physical relationships, equations, constants, etc. ultimately leading to this explosion. Qualitative Physics (we prefer NP to this term) is a special kind of physics where we use QR instead of dealing with exact mathematical relationships. The main reason for this is that exact mathematical analysis is not what human beings are thought to perform in ordinary circumstances. A more technical reason is that exact analysis is sometimes exceedingly difficult and even impossible (e.g. nonlinear differential equations).

Why are NP notions such as solids, liquids, force, time, etc. useful in design? The answer is that design objects will, when manufactured, exist in the physical world where the above notions will be in effect. Why do we need QR? There may be several answers but one good reason is that we want to determine the impact of unanticipated changes on an object in its destined environment. The common example here is an event such as the Three-Mile Island where it is

now believed that a simple, clear way of reasoning qualitatively about the physical processes and changes leading to the catastrophe would possibly prevent the accident.

Long time ago Hayes [11] proposed that one should use logic in describing NP knowledge. We plan to take this route. For example, we have good examples which demonstrate the suitability of modal logic in encoding situational calculus. (Imagine e.g. modeling the possible outcomes of envisionment [15] with the help of the possible worlds of modal logic.) As an additional tool, we want to use the ''chunking'' of knowledge — as done, for example, by de Kleer [14] in his Restricted Access Local Consequent Methods (RALCM's) — to collect together and use intelligently physical formulas. (Note that this can be done by using a class for each chunk.) For QR, the need for a symbolic algebra based on confluences is immediate [15].

While in QR we have a reasonably complete mathematical model of a situation, this itself is never sufficient for many tasks. QR is expected to interpret the numerical values of several problem variables. Assume that p is a quantity directly proportional to the quotient r/t. If r increases while t stays constant or decreases, a QR system can draw the useful conclusion that p increases. However, consider the case of both r and t increasing, albeit with unknown rates. In this case, a QR system is helpless unless it can read the values from some measuring device and do numerical computation. This need to switch back and forth between traditional computing and qualitative analysis has paved the way to coupled systems.

4.2. Coupled Systems for Expert Computation

One of the main uses of computers since their invention (and in fact, one of the reasons for their invention) has been ''numerical'' computation. It is difficult to define what is exactly *numerical* (as opposed to symbolic) but it may suffice to point out that most of the numerical analysis libraries such as IMSL^{TM} are full of numerical code — code that computes integrals, multiplies or inverts matrices, solves differential equations, etc. One unifying property of these libraries is that they work on numbers and they produce numbers. *Symbolic* computation systems such as MACSYMA ^{TM}, on the other hand, work on symbols and produce symbols.

If "the aim of computing is insight, not numbers," as Richard Hamming has been quoted to advise, then numerical code provides little help to give insights to what is going on, especially in huge computational tasks. More often than not, one gets, after hours of computation, a long list of numbers which hardly say anything explicitly (thus necessitating a post-computational period when the results are ''analyzed'') or quite disturbingly, messages like "underflow while computing M^{-1}."

Traditionally, numerical computing has been used in data processing, simulation, statistics, etc. whereas symbolic computing was employed in data interpretation, cognitive modeling, search and heuristics, and nondeterministic problem solving. It should be added that by using the term symbolic computing we do not confine ourselves to symbolic algebra systems. Many familiar expert systems (e.g. Mycin, Prospector, Dendral) have symbolic computation facilities while they wouldn't be regarded as computer algebra systems. An informal definition would then equate numerical computing with ''number crunching'' intensive processes while symbolic computing is understood as logic, heuristic, and reason intensive.

A *coupled* system "must have some knowledge of the numerical processes embedded within them and reason about the application or results of those numerical processes" [13]. It is natural to assume that in a coupled system a symbolic supervisor is at the top level, scheduling the numerical processes. Such a supervisor would have knowledge about the process's aim,

input/output behaviour, run-time limitations (e.g. the smallest and largest numbers it can deal with; truncation characteristics), and so on.

We close this subsection with a general remark about the necessity of coupled systems. Consider the design of a complex artifact such as a nuclear reactor or a space shuttle. On the symbolic side there is a need for database management, truth maintenance, computing with constraint equations, answering "what-if" questions (possibly for testing and fault simulation), etc. On the numerical side, there is a need to have expert knowledge about computational mechanics, fluid dynamics, earthquake engineering, materials science, Monte-Carlo techniques, etc. Human designers solve problems of this scale with a careful mix of symbolic and numerical techniques. Without a strong coupling of symbolic and numerical code, the automation of these complex tasks cannot be expected.

5. Combining Object Oriented and Logic Programming

5.1. Why Are We Doing This?

Logic languages such as Prolog provide the means to deal with a KB of facts; they especially come up with a uniform computational mechanism such as unification to execute logic formulas. Object-oriented languages such as Smalltalk [10] use encapsulation to structure data and employ message-passing as the underlying computational principle [26]

An obvious shortcoming of existing logic languages is the overhead of an extensive database which is physically homogeneous. This has the result that without some metalevel control, query evaluation may become hopelessly inefficient when the database is bulky. Another weakness is the lack of abstract data types. For existing object-oriented languages a major symptom of unsuitability for CAD has been the fixed (run-time) structure of the inheritance lattice. It is normally impossible to declare new objects which reside somewhere between the already existing parents and children. This normally takes us to issues such as inheritance vs. delegation which we want to avoid presently, despite their importance.

We hope that our draft proposal of a language to overcome these difficulties is pointing more or less in the right direction to combine the paradigms of logic and object-orientation. We have enumerated the requirements (originating from our desire to use it to code design knowledge) for this language in [33] and, for brevity, won't repeat them here. For another account of how to combine programming paradigms (the story of Loops) we refer the reader to [27].

5.2. IDDL, a Design Base Language

In IDDL, constants and variables denote entities. They are both called *objects*. A predicate denotes a relationship among entities and attributes which are expressed by functions. A function represents an attribute of an entity. Note that it is possible to define a function even on a set of predicates. Function definition can be done by procedures.

Logical implication and equivalence are literally so and work as a *watch-dog* in the KB. Suppose that there is a clause $p(X) \rightarrow q(X)$ denoting the transformation rule that as soon as e.g. $p(a)$ is found, $q(a)$ must be added to the KB. (Logical equivalence performs this bidirectionally.) Note that, since we employ intuitionistic logic, we don't assume $\neg p(a)$ even when $\neg q(a)$ is asserted. In the same way, deleting $p(a)$ does not imply deleting $q(a)$ or any other fact derived from $p(a)$.

There are two temporal connectives: *before* and *after*. There will be no two facts asserted at the same time. Therefore, these connectives form a fact set with complete ordering. Every object, well-formed formula, etc. has a set of information about its origin, destination, and time stamp. These are used by the SPV for controlling the inference.

Modal operators based on the system T [12] are available, i.e. *#N* (necessity) and *#P* (possibility). Since these two are based on the system T, they precede only predicates. There is an unknown operator, *%*, which can precede only atomic predicates. The *#D* default operator is another modal operator and can precede only atomic predicates. The necessity and possibility operators deal with different worlds whereas the default operator deals with nonmonotonocity or truth-maintenance within one world.

Two quantifiers are available: *#A* for the universal quantifier $\forall$ and *#E* for the existential quantifier $\exists$. A clause is defined by a list of predicates combined by connectives. Clauses and rules can be quantified.

IDDL is based on intuitionistic logic which implies further the open world assumption. Thus, IDDL uses, deep in its heart, three-valued logic including the unknown truth value rather than the conventional two-valued logic. Intuitionism means that, to check a fact, unless one has positive evidence, one is not able to say yes. The open world assumption is considered in terms of the unknown modal operator on the level of IDDL programs. Three-valued logic including **unknown** besides **true** and **false** is employed only internally. This means that the KB and the SPV distinguish **false** and **unknown** but logically these two values are treated the same. The unknown truth value is explicitly handled by the unknown modal operator. Thus, $\%p(a)$ returns **true** when there is no $p(a)$ and $\neg p(a)$.

IDDL has the concept of a *world*. It is defined as a partition of the KB such that worlds are independent from each other but can be linked so that changes in one world can propagate to others. There must always be at least one currently active world in the KB. Worlds are subject to manipulation. A world consists of (i) objects, (ii) facts, and (iii) available functions. A world is created or declared with these elements. There are *global* worlds and *local* worlds. Local worlds are defined as those belonging to scenarios. Global worlds persist in the KB forever until explicitly removed, while local worlds automatically disappear after the execution of scenarios.

Two types of action are possible: pure *inquiries* and *assertions*. Suppose that we want to ask the KB $p(X)$. If there is $p(a)$ and $p(b)$ then X is instantiated to the set $\{a, b\}$ and **true** is returned. If there are no such facts in the KB, X remains uninstantiated and **false** is returned. Consider now the inquiry $\neg p(a, b)$. If $\neg p(a, b)$ is found, **true** is returned; however, if $p(a, b)$ is found, **false** is returned. If neither $\neg p(a, b)$ nor $p(a, b)$ is found, **unknown** is returned.

#N and *#P* are used to deal with different (currently active) worlds. $\#Np(a)$ returns **true** when all the currently active worlds have $p(a)$; if there are some worlds where $\neg p(a)$ is found, **false** is returned. If some or all of the active worlds do not have $p(a)$, **unknown** is returned. $\#Pp(a)$ returns **true** when there is at least one active world which has $p(a)$; **false** is returned when all of the active worlds have $\neg p(a)$. If there is neither $p(a)$ nor $\neg p(a)$ in the active worlds then **unknown** is returned.

A fact such as $\#Dp(a)$, matches $\#Dp(a)$ and returns **true**. Otherwise, it returns **false** (because the problem is whether $p(a)$ is qualified by *#D* or not). Assertions, on the other hand, are associated with modifying the KB. Again consider asking $p(X)$. If there exist $p(a)$ and $p(b)$, X is instantiated to the set $\{a, b\}$ and **true** is returned. In this case the assertion *succeeded*. If there is no such fact in the current worlds, an object which is referred by X is created

and this fact is added to the current worlds. Finally, **true** is returned as the logical value of this assertion to indicate that it succeeded. By assertion one may create new objects. The assertion $p(a)$ *fails* when there is already $\neg p(a)$ in the current worlds. We note that & (logical and) and | (logical or) operators are different in terms of assertion. For example, consider the assertions of $p(a)$ & $q(a)$ and $p(a)$ | $q(a)$. The and operator puts both $p(a)$ and $q(a)$ in the currently active worlds. If either of them fails, the whole assertion fails. On the other hand, the or operator creates a copy of the currently active world and puts $p(a)$ and $q(a)$ separately into the original world and the copied world. Having an or operator on the right hand side (RHS) of a rule, one implicitly creates a new world. If both of those assertions fail, the whole assertion fails.

There is a built-in predicate *assert* which explicitly does an assertion. This predicate is, by definition, a higher order predicate. Inquiries are specified by the built-in predicate *inquire*. The opposite of *assert* is *remove* which retracts a fact from the KB. In case there is an equivalence definition in the KB, by asserting a fact an equivalent fact might be added to the KB automatically. However, this will not happen when a fact is removed.

By asserting $\#Np(a)$, all currently active worlds will have $p(a)$ and the assertion will succeed. If there are some worlds where $\neg p(a)$ is found, the assertion fails. By asserting $\#Pp(a)$, all the active worlds will have either $p(a)$ or $\#Dp(a)$. Worlds which already have $\neg p(a)$ are not touched. However, if all the active worlds already have $\neg p(a)$, the assertion fails. When a fact qualified by the default modal operator $\#Dp(a)$ is asserted, $p(a)$ is put into the current worlds and labeled as *default*. Facts which are derived from those default facts will be labeled as *derived facts* (from $p(a)$). Sometime in the future, if $p(a)$ is asserted then these labels will be removed. If $\neg p(a)$ is asserted in the future, all the assumed facts and derived facts will be removed from the current worlds and $\neg p(a)$ is asserted instead.

A rule has the well-known *if-then* syntax. Note that there is no logical implication in a rule; this is completely different from the $\rightarrow$ operator. Rules will be purely procedurally interpreted by the SPV. A rule is interpreted as "if *clause-1* is true, then *clause-2* must hold." Thus, unless specified, the following is expected by default. If *clause-1* is found, then *clause-2* is asserted. For the left hand side (LHS) part of rules, normally clauses are regarded as inquiries. For the right hand side part, assertions are assumed unless explicitly specified (such as just an inquiry). If it is impossible to assert the entire clause, the assertion fails. If, for one reason or another, the assertion on the RHS fails, it is taken that the rule failed.

An *instantiation list* is used to keep track of "once matched facts" so that they won't fire again. Quantifiers are used to talk about objects instantiated to variables. Consider an inquiry $p(X) \rightarrow q(X)$ and facts $p(a)$, $p(b)$, $p(c)$, $q(a)$, and $q(b)$ in the KB. $\#A[X]\, p(X) \rightarrow q(X)$ returns **unknown** since there is no $p(c)$. On the other hand, $\#E[X]\, p(X) \rightarrow q(X)$ returns **true** and its instantiation list in this case is $\{a, b\}$. In IDDL quantifiers can also quantify rules.

If on the LHS there are facts labeled *default* or *derived-from*, the facts on the RHS will be asserted with the label *derived-from* the fact in the LHS. By doing so, one keeps track of assumed facts (viz. truth maintenance).

A scenario is defined as a set of rules. A scenario set (or a scenario base) is a set of scenarios. A scenario has the following elements:

- Scenario name (*List-of-Worlds*)
- Flow declaration reference (to resolve the destination references)
- World declaration reference, which further consists of:

  - Object declaration reference
  - Function declaration reference
- Object declaration reference
- Function declaration reference
- Rules

A scenario is active when the control is passed to it by the SPV or another scenario. The argument *List–of–Worlds* defines worlds passed by the caller. A scenario can have those *imported* worlds as well as local worlds declared in the world declaration references. There is a world called *default-world* which can be used without declaration and is local only to that scenario. Object and function declarations on the same level as the world declaration belong to this local world, *default-world*. The object declaration in a world defines local objects which can be used only in that world. Global objects are declared as local objects of a global world. From scenarios there must be a reference to those global objects. Local objects will never be seen from upper level scenarios. The idea of object declaration almost corresponds to the idea of ''typing'' in conventional languages.

Worlds cannot be accessed from scenarios which have no declarations referring to them. In case a scenario has more than two worlds, which world is to be considered is specified by the *enter* built-in predicate. (The opposite is *exit* predicate.) Note that one cannot switch worlds that are created by an ! operator. These copied worlds are equally treated as the original worlds. By declaring objects in the object declaration part, the current world contains only those declared objects. Two or more worlds can share objects by *referring* or by *importing/exporting*. This takes place in such a way that a declared object and its relevant clauses and functions associated with objects are automatically collected and put into the current world.

A world can be created by the *enclose(World, List-of-Objects)* built-in predicate. This predicate creates a new world called *World* with *List-of-Objects*. This is an enclosure mechanism. After the enclosure, the enclosed world will be treated as an object. The type of *World* is defined by its object declaration. Whether *World* is global or local is dependent on that. After enclosure, the contents of the world can be accessed only by functions. The enclosure mechanism can be, therefore, perceived as ''intensionalising extensions,'' and functions are used to ''find the anatomy'' of an enclosed world. Similarly, it is not far-fetched to regard functions as equivalent to messages.

A scenario will be executed in the following way. Examining the rules from the top, the first rule whose LHS is satisfied is selected. Then the RHS of this rule is asserted. If the assertion was successful, search for the next matching rule starts with the rule following the previously executed rule. If the assertion failed, once again search starts and another rule will be selected. In case the execution terminates successfully, all the results will be preserved. In case of failure, all the results will be removed. If the search for the next applicable rule comes back to the most recently executed rule because the search ''wraps around,'' it is judged that there are no applicable rules and the execution of the entire scenario stops (i.e. *no-more-rule* situation). The execution of a scenario can also be stopped by the execution of either *success* or *fail* built-in predicates. When a scenario terminates successfully, worlds related to that scenario are preserved. When a scenario terminates unsuccessfully, related worlds are removed from the KB. When a rule is selected, an instantiation list is created. This list is preserved until the end of the execution of the entire scenario so that the same rule will not be applied to the identical objects in the same situation.

One can "open" an object and regard it as a world (called by the name of the object). This is done by the *open* built-in predicate. To leave that world, one can use the *close* built-in predicate. The former assumes the *enter* predicate, and the latter assumes *exit* as prerequisites. A *select* predicate changes the active scenario to a new one and restricts the active objects used in that scenario to *List-of-Objects*. In case this list is empty, the active objects are not restricted. On the other hand, a *use* predicate adds the new scenario name to the active scenario and restricts the active objects used to *List-of-Objects*. This predicate, therefore, enlarges the set of available rules. The last two predicates can be true when the subscenario is finished by the execution of a *success* built-in predicate or by the no-more-rule situation. They can be false when the subscenario is explicitly terminated by the execution of a *fail*. This means "selection" switches active scenarios while "using" shows details of the presently manipulated objects via more dedicated rules. These two predicates are important to realise the so-called "multiworld mechanism" of §2.1. Finally, in order to restrict active objects without changing scenarios, one can use the predicates *consider(List-of-Objects)* and *unconsider(List-of-Objects)*.

6. Summary and Future Directions

The aim of our work is to develop an integrated, interactive, and intelligent computer-aided design system. IIICAD will be a generic system which may be used in any design domain and will incorporate three types of design knowledge. First, the system has general knowledge about the design processes based on a set-theoretic design theory. Second, it has domain-dependent knowledge belonging to a specific area (e.g. VLSI) where it is actually used. Third, the system maintains knowledge about previously designed entities. This kind of history mechanism enables the system to reuse its knowledge in the forthcoming design activities. It is useful to imagine this as a variant of software reuse.

The work on IIICAD is divided into several areas of interest in which different AI techniques are used:

- Formalisation of general design theory; modal and other nonstandard logics as a knowledge representation language.
- Common sense reasoning about the physical world (naive physics) and coupled systems.
- Integration of object oriented and logic programming paradigms.

As a result, a formal definition of a kernel language for design will be generated. This language for integrated data description (called IDDL) will be used to implement the IIICAD system. IDDL, equipped with nonstandard logics, enables the IIICAD system to describe design knowledge and to control the design process in a highly expressive and robust manner. In §5.2, we gave a taste of IDDL, cf. [35] for full draft specifications. Formalisation of the design theory will take place by means of frame-like structures called scenarios. We use General Design Theory [31] as a basis for formalising design processes and knowledge.

The NP and QR knowledge which will be used during the design process, performs common sense reasoning about the physical world. Depending on the phase of the design process, the declaration of the physical qualities of a design object takes place in logic and by means of references to physics laws. Interface between the IIICAD system and already existing qualitative reasoning systems (such as ENVISION [15] and QSIM [16]) should also be studied.

Declaration of knowledge about a design object may be done by logically manipulating the object's attributes. At the same time, the knowledge itself refers to the specific behaviour of the object. These two characteristics lead to a need to integrate object oriented and logic

programming styles into one language. One of the major areas of interest within IIICAD, therefore, is to find out how this integration to be achieved (i.e. multi-paradigm languages) and what additional properties our draft proposal, IDDL, should have. We use the Smalltalk-80^{TM} [10] programming environment to implement IDDL (and IIICAD) and regard Smalltalk's excellent user interface and debugging tools as major aids for software development in this scale.

> *"What has happened to the design "guru"? Didn't every design and development engineering department once have one? At one of my first jobs the department manager and his assistant sat in their glassed-in offices in one corner of our lab. The rest of us each had our 8-foot section of bench. Except for our guru. He sat outside the bosses' offices at a desk of his very own. And while we toiled at 'scopes and breadboards, he didn't do anything. Nothing, that is, except answer questions the rest of us could not."* [5]

References

1. 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).
2. D. Bobrow, S. Mittal, and M. Stefik, "Expert systems: Perils and promise," *Communications of the ACM* **29**(9), pp. 880-894 (Sept. 1986).
3. D.C. Brown and B. Chandrasekaran, "Knowledge and control for a mechanical design expert system," *IEEE Computer* **19**(7), pp. 92-100 (July 1986).
4. R. Carnap, *Meaning and Necessity: A Study in Semantics and Modal Logic*, The Univ. of Chicago Press, Chicago, Ill. (1947).
5. D. Christiansen, "On good designers," *IEEE Spectrum (Special report: On good design)* **24**(5), p. 25 (May 1987).
6. J. Doyle, "A truth maintenance system," *Artificial Intelligence* **12**, pp. 231-272 (1979).
7. 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).
8. J.S. Gero (ed.), *Knowledge Engineering in Computer Aided Design*, North-Holland, Amsterdam (1985).
9. J.S. Gero (ed.), *Expert Systems for Computer Aided Design*, North-Holland, Amsterdam (1987, to appear).
10. A. Goldberg, *Smalltalk-80: the Interactive Programming Environment*, Addison-Wesley, Reading, Mass. (1983).
11. 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).
12. G.E. Hughes and M.J. Cresswell, *An Introduction to Modal Logic*, Methuen, London (1972).
13. 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).
14. J. de Kleer, "Qualitative and quantitative knowledge in classical mechanics," AI-TR-352, Artificial Intelligence Lab, MIT, Cambridge, Mass. (Dec. 1975).
15. J. de Kleer and J.S. Brown, "A qualitative physics based on confluences," *Artificial Intelligence* **24**, pp. 7-83 (1984).
16. B. Kuipers, "Qualitative simulation," *Artificial Intelligence* **29**, pp. 289-338 (1986).
17. J. Lansdown, "Graphics, design, and artificial intelligence," in *Theoretical Foundations of Computer*

IMSL is a trademark of IMSL, Inc. MACSYMA is a trademark of Symbolics, Inc. Smalltalk-80 is a trademark of Xerox Corp. Mention of commercial products in this article doesn't imply endorsement.

Graphics and CAD, ed. R.A. Earnshaw, NATO ASI Series, Springer-Verlag, Heidelberg (1988, to appear).

18. H.J. Levesque, "The logic of incomplete knowledge bases," pp. 165-189 in *On Conceptual Modelling (Perspectives from Artificial Intelligence, Databases, and Programming Languages)*, ed. M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, Springer-Verlag, New York (1984).
19. D. McDermott, "Nonmonotonic logic II: Nonmonotonic modal theories," *Journal of ACM* **29**(1), pp. 33-57 (Jan. 1982).
20. M. Minsky, "A framework for representing knowledge," pp. 211-277 in *The Psychology of Computer Vision*, ed. P. Winston, McGraw-Hill, New York (1975).
21. 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).
22. R.C. Moore, "The role of logic in knowledge representation and commonsense reasoning," pp. 336-341 in *Readings in Knowledge Representation*, ed. R.J. Brachman and H.J. Levesque, Morgan Kaufmann, Los Altos, Calif. (1985).
23. J. Mylopoulos and H.J. Levesque, "An overview of knowledge representation," pp. 3-17 in *On Conceptual Modelling (Perspectives from Artificial Intelligence, Databases, and Programming Languages)*, ed. M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, Springer-Verlag, New York (1984).
24. R. Reiter, "Towards a logical reconstruction of relational database theory," pp. 191-233 in *On Conceptual Modelling (Perspectives from Artificial Intelligence, Databases, and Programming Languages)*, ed. M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, Springer-Verlag, New York (1984).
25. J. Skylar, "A cut above: A manufacturer of lawnmower engines pushes computer-aided design software to new limits," *Logic (a publication of Control Data)*, pp. 3-7 (Spring 1987).
26. M. Stefik and D. Bobrow, "Object oriented programming: themes and variations," *AI Magazine* **6**(4), pp. 40-62 (Winter 1986).
27. M. Stefik, D. Bobrow, and K. Kahn, "Integrating access oriented programming with a multiparadigm environment," *IEEE Software* **3**(1), pp. 10-18 (Jan. 1986).
28. P. ten Hagen and T. Tomiyama (ed.), *Intelligent CAD Systems 1: Theoretical and Methodological Aspects*, Springer-Verlag, Heidelberg (1987, to appear).
29. T. Tomiyama and P. ten Hagen, "Representing knowledge in two distinct descriptions: extensional vs. intensional," CWI Report CS-R8728, Centre for Mathematics and Computer Science, Amsterdam (June 1987).
30. T. Tomiyama and P. ten Hagen, "The concept of intelligent integrated interactive CAD systems," CWI Report CS-R8717, Centre for Mathematics and Computer Science, Amsterdam (April 1987).
31. 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).
32. T. Tomiyama and P. ten Hagen, "Organization of design knowledge in an intelligent CAD environment," in *Expert Systems for Computer-Aided Design*, ed. J. Gero, North-Holland, Amsterdam (1987, to appear).
33. 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).
34. B. Veth, "Design as a formal, knowledge engineered activity," CWI Report, Center for Mathematics and Computer Science, Amsterdam (1987, to appear).
35. B. Veth, "IDDL, an Integrated Data Description Language," CWI Report, Center for Mathematics and Computer Science, Amsterdam (1987, to appear).
36. H. Yoshikawa and E.A. Warman (ed.), *Design Theory for CAD*, North-Holland, Amsterdam (1987).