Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

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

Organization of design knowledge in an intelligent CAD environment

Computer Science/Department of Interactive Systems Report CS-R8720 April

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.).

Organization of Design Knowledge in an Intelligent CAD Environment

Tetsuo Tomiyama, Paul J.W. ten Hagen
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The concept of so-called intelligent CAD systems has been paid more and more attention as a promising approach toward the next generation integrated engineering environment. In this paper, first we propose the concept of Intelligent Integrated Interactive CAD systems. We introduce a language based on predicate logic and the object oriented programming paradigm, and describe the mechanisms of the system in this language. Then, we discuss the possibility to describe and organize design knowledge using this language. From this discussion we will see how design knowledge should be embedded and work in an intelligent, integrated, and interactive CAD environment.

1980 *Mathematics Subject Classification (1985):* 69H12, 69H21, 69K14, 69L60
1982 *CR Categories:* H.1.2, H.2.1, I.2.4, J.6

Key Words & Phrases: conceptual modeling, data modeling, knowledge engineering, knowledge representation, CAD.

Note: This report was presented at *IFIP Working Group 5.2 Working Conference on Expert Systems in Computer-Aided Design* which was held on February 16-20, 1987, in Sydney, Australia. It will be included in the conference proceedings, edited by J.S. Gero, which will be published by North-Holland.

1. Introduction

There are two major streams for developing a so-called intelligent CAD (*Computer Aided Design*) system [Ger85, SrA86]. One is to give the system powerful problem solving ability. This can be achieved by developing dedicated expert systems for designing in narrow problem domains. The other is to put emphasis on understanding the designer's intention in order to assist her/him intelligently. The concept of intelligent user interface management can be classified into this type of efforts.

However, it is becoming clearer that neither of these approaches can by itself be powerful enough to build a future CAD system which solves the problems of conventional CAD systems [ToY85]. In fact, what we need is developing a general CAD system framework to support the designer rather than pursuing a sophisticated expert system which can solve a design problem or a smart user interface management system.

Thus, we propose the IIICAD (*Intelligent Integrated Interactive CAD*) system. It provides an integrated infrastructure for those problem solving tools together with a smart user interface. This means, in the architecture of IIICAD, the key points are integration of knowledge and intelligent supervisor which controls the system based on this integrated knowledge. In the past, there have been many researchers proposing integrated knowledge based CAD systems. Some of them (for example, see [RHS85]) can offer a system

Report CS-R8720
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

integration environment for existing CAD systems as well as AI tools, such as OPS5 [For81]. However, because every system component should understand the designer's semantics (although there are many problems in using this word), and because this understanding should be common to the entire system, we need a higher degree of knowledge integration rather than the simple integration of the systems. As a consequence of knowledge integration, so-called expert systems will reappear as special knowledge libraries which form an integrated part of the total knowledge base.

In this paper, we consider CAD systems for mechanical engineering as an example and we discuss how the concept of IIICAD will be realized, especially, from a viewpoint of integration of design knowledge. In CHAPTER 2, we propose the concept and architecture of IIICAD which is derived from a design process model. This design process model, called *metamodel evolution model*, suggests also various important concepts for IIICAD, such as the supervisor concept among other things.

IIICAD is integrated by using a common language called IDDL (*Integrated Data Description Language*) which is now under development. The supervisor that plays the most essential role in the IIICAD architecture uses scenarios written in IDDL to control the system. Because there are many issues for IDDL coming from both theoretical and implementational aspects which cannot all be described in this paper, in CHAPTER 3 we define a simple pseudo language which is a subset of IDDL and has similar functions just for explanation purpose.

There arises, then, an interesting question whether or not it is possible to describe design knowledge in IDDL under the proposed IIICAD architecture. In CHAPTER 4, we answer to this question and discuss how to describe and organize design knowledge in IDDL. We use a design process model which was proposed in CHAPTER 2 to derive the architecture of IIICAD. Based on this design process model, we first analyze design knowledge for machine design. We then see some example descriptions of design knowledge in the pseudo language in a form of scenarios. These examples will show how to describe design knowledge, how to realize a flexible interaction between the user and the system, how to provide the metamodel mechanism, etc., by using a rather simple mechanism of the pseudo language. From these discussion, we can consider the distribution of design knowledge in the architecture of IIICAD.

2. Intelligent Integrated Interactive CAD (IIICAD)

2.1. Concept

CAD systems have become indispensable tools in various fields of industry. Especially, in mechanical engineering, CAD systems have brought about revolutional changes by handling geometric information, primarily, due to the progress of computer graphics. However, as systems were penetrating into actual applications, it was widely realized that considering geometric information only would be insufficient for machine design. For instance, technological information is now placed outside the system [KSH83].

In machine design, there are many models which represent a design object, such as a mathematical model for control, a kinematic model for evaluation of the mechanism in motion, a finite element model for evaluation of the strength, etc. In principle, these models should represent the identical design object. However, there is no structure which guarantees their integration, consistency, or completeness.

The amount of data which is required for three dimensional modeling can be very huge. Various techniques to ease inputting the data have been proposed. One of them is intelligent user interface management which even allows hand drawings, voice input, etc. [SaW81]. Unfortunately there exists a limit to this idea which simply eases to input the data but does not reduce the amount of the data as long as the designer has to feed in. Therefore, we have to introduce a new methodology which will allow the system to handle the design object from a very early stage of design. This boils down to the system being able to generate models from semantic knowledge including previously defined objects.

The new generation CAD systems should be, therefore, not only intelligent but also integrated and interactive so that the abovementioned problems can be solved. The concept of IIICAD is proposed here as our solution to those problems.

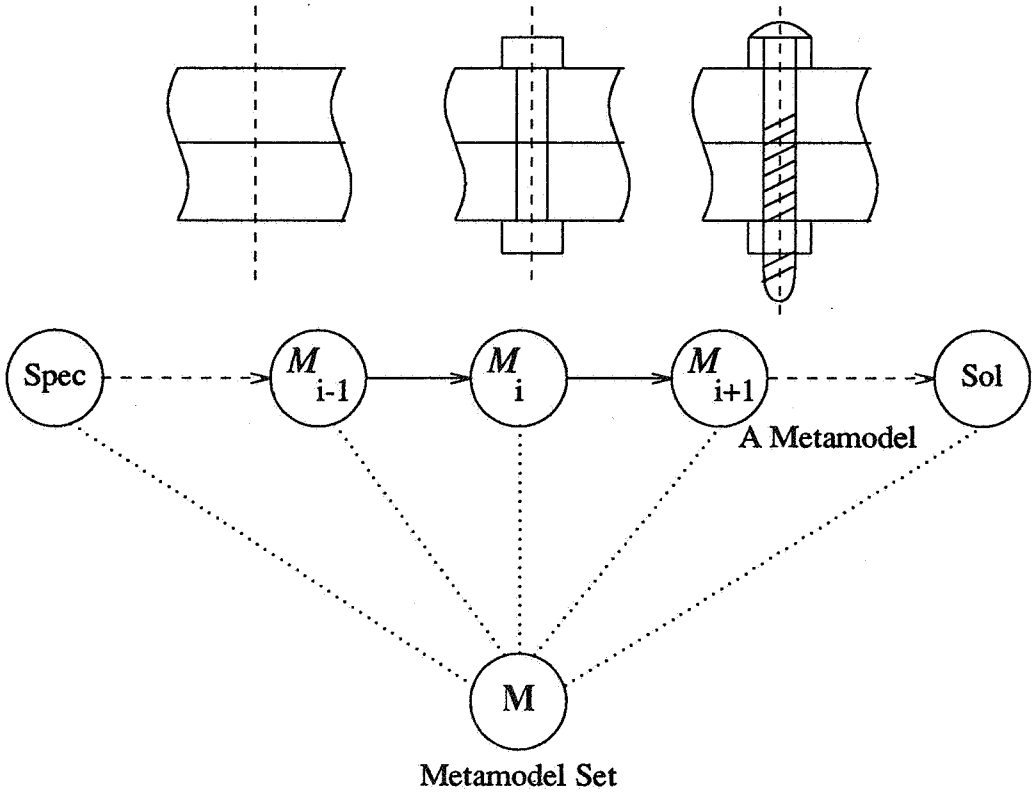


FIGURE 1. Evolution of Metamodel

2.2. Design Process Model

In this section, we propose a model of design processes to derive the architecture of IIICAD in the next section. This model is a fundamental concept in this study and will be used also in later discussions in CHAPTER 4.

According to *General Design Theory* [Yos81], we can regard a design process as a mapping from the function space onto the attribute space. The designer is given a specification written in functional terms and has to find or create a solution described in attributes. This mapping from the function space onto the attribute space is, however, difficult to describe in a formal way. Thus, instead of having direct mapping, we may consider a design process which is of stepwise refinement nature.

FIGURE 1 illustrates such a design process model called *metamodel evolution model* [ToY87]. In this model, the designer given specifications first picks up or imagines the most appropriate candidate with very rough descriptions. This candidate will be detailed during the design process and finally we will get a design solution with full descriptions sufficient for manufacturing as the ultimate state of this detailing process. At each stage of detailing (or *evolution*) process, the designer checks whether or not the candidate is fulfilling the specification from various aspects. These inspection and evaluation require proper models, such as kinematic model, mathematic model, finite element model, etc.

A metamodel, in this design process model, is a set of descriptions, existing at that moment of designing, about the design object (as a candidate for design solution) which is detailed in a design process step by step and from which we derive models for evaluation. Designing is, therefore, a collection of small procedures for detailing a metamodel. A model for evaluation is created from the metamodel as its partial description.

This metamodel evolution model explains real design processes very well. For instance, usually the designer given the specification may imagine a couple of possibilities without information in detail. During the

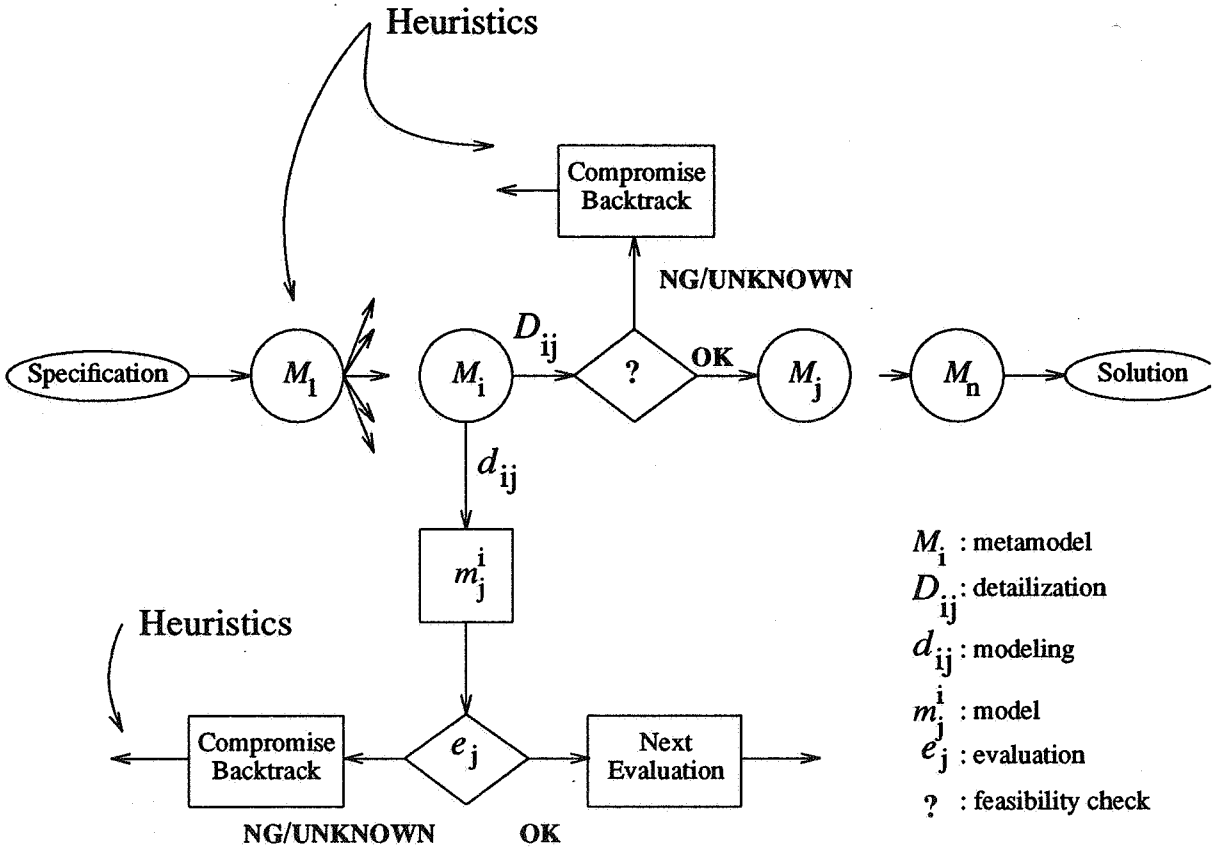


FIGURE 2. Formalization of Evolution Model

designing process, this image will grow and get realistic. Therefore, this design process model can be used as a basic model for building IIICAD. FIGURE 2 shows a more formalized description of the evolution model. It tells several important principles.

(1) A metamodel, M_i, is a set of descriptions about the design object. This means we have a central mechanism to store information about design object model.

(2) Design process knowledge is stored in detailing procedures, or evolution procedures, D_{ij}. Note that this process requires feasibility check to see if the candidate can exist as a physically possible entity.

(3) Creating models for various kinds of analyses from a metamodel is done by modeling knowledge, d_j.

(4) A model, $m^i{}_j$, created by d_j from M_i, is used for analyzing the properties of the design candidate and to represent in a particular type of information. The result of analysis or evaluation, e_j, is used to judge whether or not the candidate is appropriate as the design solution.

(5) There can be three places where heuristic knowledge or so-called expertise is necessary. First, we need such knowledge for planning how to design. This knowledge is used to determine which D_{ij} is used from the metamodel state M_i. Secondly, if something wrong is found in the feasibility check after an evolution procedure, we need to know to which state we make backtrack or how to compromise between the specifications and the solution. Thirdly, the same type of knowledge is required after evaluation e_j.

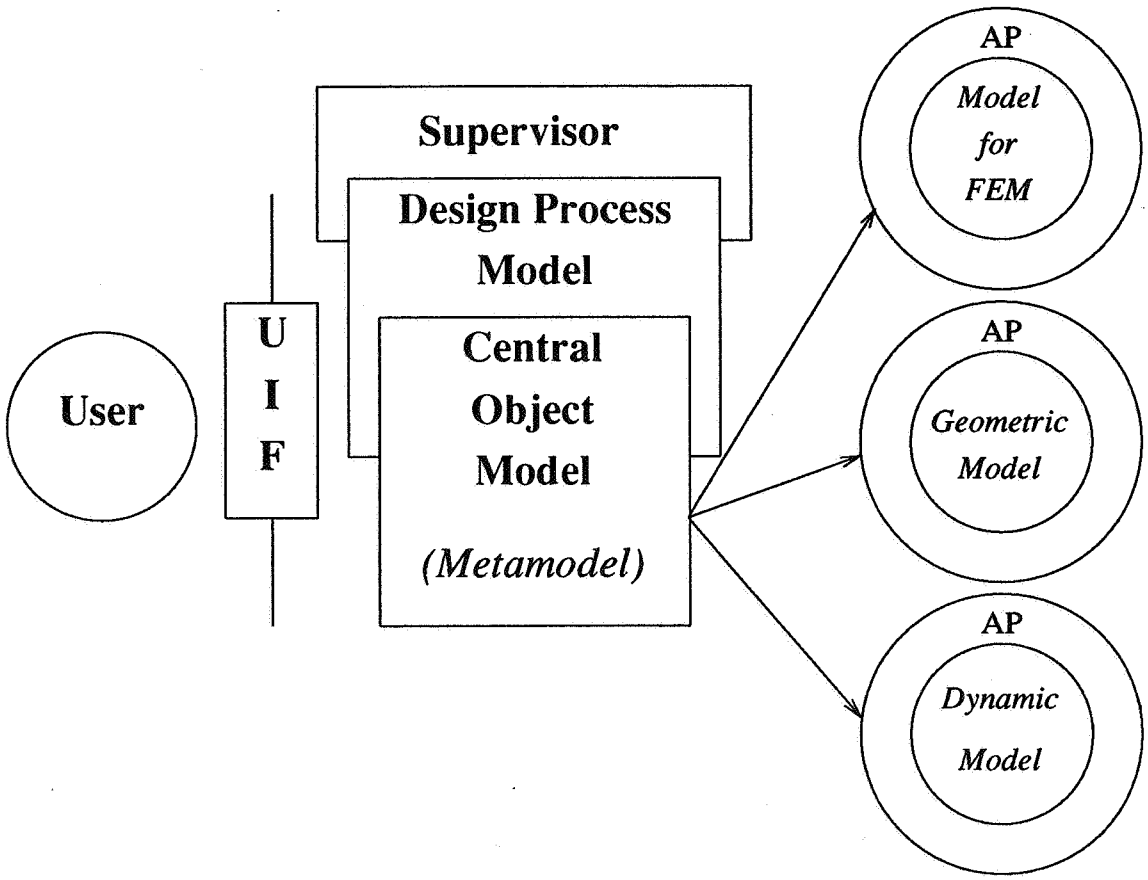


FIGURE 3. Basic CAD Model

2.3. Architecture of IIICAD

As mentioned in the previous section, a designer finds or creates a new entity which may never have existed in the past given specifications written in terms of functions, and she/he describes it in terms of attributes so that one can actually manufacture it. Because it is too optimistic based on the present computer technology to ask computers to find or to create a completely new entity, we define a CAD system as a computer system which assists the designer in such a way that the designer can utilize his/her maximum ability of creation. This means that a CAD system should have *model* or *descriptions* concerning the design object which have maximum similarity to the designer's own image, and that a future CAD system must be controlled by an intelligent *supervisor* which has knowledge about design processes and objects to get advices. CAD systems in general therefore, must have descriptions about the design objects and processes and perform the best in computation and reasoning that can be expected by the present computer technology to assist the designer.

These two components, i.e., the models and the supervisor, will play an important role in the architecture of future CAD systems. From the design process model described in the previous section, these two can be further elaborated into the requirements following below which could be satisfied by a CAD model illustrated in FIGURE 3.

(1) A mechanism for describing a central model of the design object called metamodel, *M*.

(2) A mechanism to describe knowledge about the design process; i.e., knowledge on how to evolve models, how to use heuristic knowledge, etc.

(3) A mechanism and knowledge, *d*, to derive models, *m*, from the central model for the various technological evaluations.

(4) Tools to perform such evaluations, *e*.

(5) A mechanism to communicate with the designer in such a way that she/he can recognize various aspects of the models.

(6) A mechanism to give assistance to the designer; i.e., the supervisor.

The CAD system model in FIGURE 3 can be elaborated and extended to an architecture shown in FIGURE 4. In the following section, we explain its elements.

2.4. System Elements of IIICAD

2.4.1. Supervisor

The *supervisor* (SPV) is the kernel of the system and it corresponds to (6) in SECTION 2.3. It watches and controls all the information flows in the system, such as user operations, status of the system, etc., and tries to understand the intention of the user.

The control is based on *scenarios* which describe standard designing procedures. When the user makes an obvious mistake, SPV should detect it by comparing the user actions with the scenario. In this way, SPV would add intelligence to the system. SPV does not have the initiative for the design process, because the final responsibility of design is left with the designer.

Scenarios are, in other words, descriptions about the design knowledge. They are written in a language called IDDL discussed in CHAPTER 3. SPV and scenarios together fulfill requirements (1) to (3) described in SECTION 2.3. In CHAPTER 4, we will see how scenario mechanism realize such functions in detail.

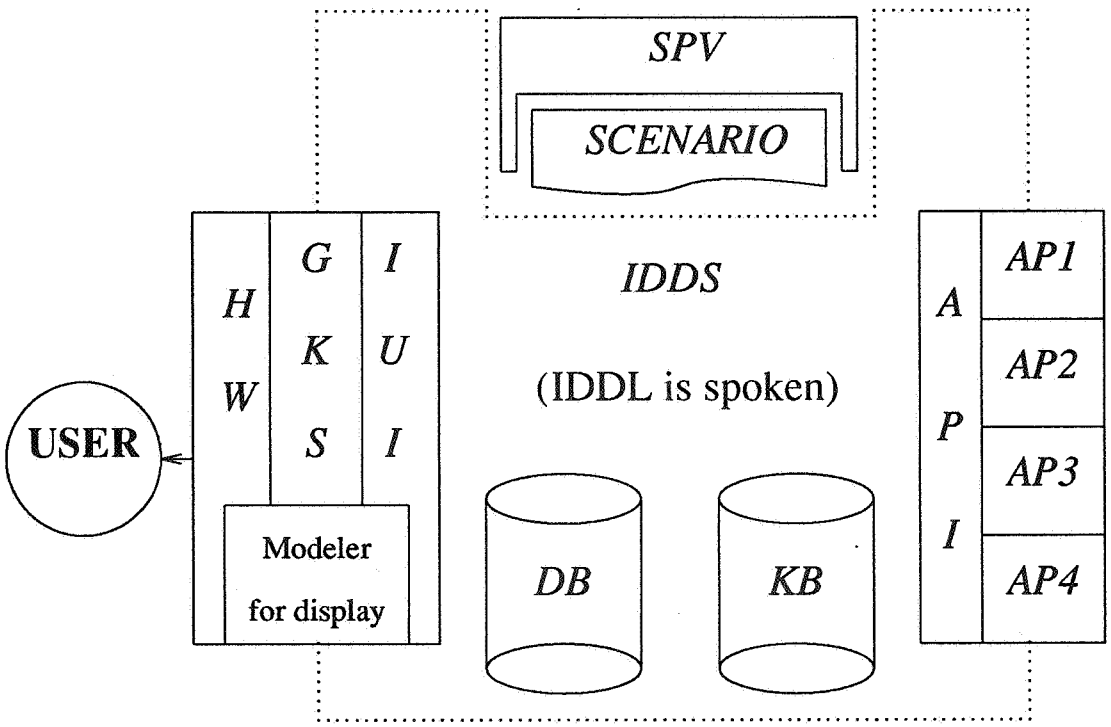


FIGURE 4. Configuration of the IIICAD System

2.4.2. Integrated Data Description Schema

The *Integrated Data Description Schema* (IDDS) is a gateway to the *databases* (DB) and *knowledge bases* (KB), which means the user does not have to pay attention to exactly where and how to store and retrieve information. All the information comes in and out through IDDS, and therefore, DB and KB are transparent to the user.

IDDS has a language called *Integrated Data Description Language* (IDDL) spoken by all the system elements. IDDL is the language to describe the design knowledge and the design object guaranteeing integrated and unified descriptions about the design knowledge.

2.4.3. Intelligent User Interface

The interface between IIICAD and the user has relatively complicated structure in order to realize flexible and intelligent interaction, and it corresponds to the mechanism (5) in SECTION 2.3. There must be several interfacing systems reflecting the necessity to access information from very low level input/output to very high abstracted level.

The highest level interface system is called *Intelligent User Interface* (IUI) which is driven also by scenarios written in IDDL and which accepts messages from SPV or other subsystems and sends them to the lower level interface systems, such as the Dialogue Cell system [BKt82, teD84, vat87] and GKS, to control the physical device. It accepts a user's input from the dialogue cell system and translates it to semantical descriptions in IDDL which in turn will be sent to SPV.

2.4.4. Application Interface

The *Application Interface* (API) accepts a part of the central model descriptions about the design object and translates them into an individual model used by an application program which corresponds to (4) in SECTION 2.3 and *vice versa*. This means API should have;

(1) descriptions about applications, such as their functions, what kind of model description is required, what data is returned from applications, etc.

(2) descriptions about the translation from the central model to the model used in an application program.

(3) descriptions about data requests from applications; i.e., how to request SPV to give a set of data to an application program.

Therefore, API is more or less similar to a run-time system monitor. It should also have scenarios written in IDDL to perform its tasks.

In IIICAD for machine design, as application programs we may expect geometric modeling system, analysis systems typically such as finite element analysis systems, expert systems for consultation, etc.

2.5. System Behavior

2.5.1. System Cycle

The IIICAD system has a basic cycle as follows.

(1) Every system component, such as IUI, API, etc., reports its own status or requests to SPV. Typically, this report can be a user's request through IUI.

(2) SPV asks DB/KB whether there is a proper scenario for the situation created by the status reports from the subsystems. This can be done also by explicit user commands.

(3) SPV executes the selected scenario. Execution is done in the following cycle.

(3-1) Recognition of the situation, i.e., selection of the most matching rule.

(3-2) Execution of the selected rule.

(3-3) If no rule, stop with this scenario.

Execution of a rule means passing requests from one subsystem to the most proper one. The response from the inquired system will be returned to the system from which the request

originated. If it was satisfied by the response, it reports satisfaction to SPV and SPV will proceed to the next step. If not, SPV must execute exception handling which might or might not be described in the scenario.

(4) If there is no more rules which can be applied (*no-more-rule situation*) or SPV encounters an explicit end command, it returns the control to the higher scenario.

2.5.2. Expected Usage of IIICAD

The cycle described in the previous section will make the following usage of the system possible.

(1) The designer would type in or select from a menu (or by any possible interaction method).

Design winch

(2) When IUI gets this command, it creates a request to SPV telling that the designer wants to design a winch.

(3) Receiving this request, SPV asks DB/KB to give him a scenario for designing a winch. If not found, something should happen to tell the designer that the system knows nothing about a winch. The most reasonable thing SPV could do is probably to ask the designer to select an example which is similar to a winch from the system library.

(4) SPV begins the designing process of a winch based on the selected scenario. It might be something like this.

What is the name of the winch? WINCH
How much load do you expect for WINCH? 100
Do you mean 100kgf? YES
.....

(5) The dialogue described here took place in the following way. First, SPV needed to know the name of the winch and it sent a message to IUI to ask the designer. IUI, without knowing what in fact a winch semantically is, asked the question. Because what IUI needed was a name from the user for a winch and because there was knowledge written in the scenario for IUI that a name is given by a character string, IUI invoked a graphics system to put a prompt and to fetch the designer's input probably from the keyboard. IUI got input from the user and sent it directly to SPV.

(6) If IUI is requested by SPV to get an integer value between 0 and 8, for example, and if the designer types a letter key ('s', for instance), this is an obvious error, a mistake, or a misunderstanding of the designer. In this case, an error recovery at some level is done by IUI, for example, by showing possible input to the designer. However, it must be also reported to SPV telling that the designer made an error and this is the fifth of this type and the twenty-eighth in total. This information should be used firstly by IUI to control the dialogue dynamically and secondly by SPV to judge whether the designer really understands the design process of a winch. For instance, if the designer gave an unexpected answer to a question asking the usage of the winch, it should be interpreted either as a misunderstanding or as trying to design some new which the system does not know, but not as an error because he is a beginner. This is an example of SPV's understanding the intent of the designer.

(7) During the design process, the designer always takes the initiative. Thus, basically he must always enter commands to indicate what to do next. If SPV finds a difference between the scenario and the designer's action, i.e., SPV cannot find a rule matching to the input, it would be either an error of the designer, that the designer does something new, or that the scenario is wrong. In any case, SPV must warn the designer. If the designer admitted an error or if he asked a question, SPV must give a help, give suggestions, and ask the designer to specify more precisely his intent.

3. A Pseudo Language for System Description

For convenience in the further discussion, we introduce a pseudo language for describing behaviors of IIICAD as a preliminary version of IDDL (*Integrated Data Description Language*). IDDL is a language used by most of the IIICAD subsystems; for instance, SPV is driven by scenarios which are written in IDDL.

In short, IDDL is based on logic programming paradigm with a flavor of object oriented programming paradigm. It is structured in a way similar to those of a so-called forward reasoning production rule system.

However, there are many issues which must be taken into consideration to develop IDDL and which cannot be described fully here. Because IDDL is still under development, only for explanation purposes we introduce the concept of IDDL without giving the syntax. We do not justify why we need such functions, either.

3.1. Constants and Variables

Constants and variables denote entities. They are both called objects. The syntax follows the convention of well-known Prolog [ClM81]; i.e., a constant begins with a lower case letter and a variable begins with an upper case letter.

An object can be created either explicitly by scenarios or by assertion of a clause. See SECTION 3.7 for the former case and SECTION 3.5 for the latter case.

3.2. Predicates

A predicate denotes a relationship among entities and attributes which are expressed by functions. A predicate is a character string beginning with a lower case letter followed by a pair of parentheses which enclose a list of objects and functions separated by commas. (From now on, sometimes, this list of objects will be denoted by *list-of-objects*.) A predicate preceded by `~` and `%` denote negation and unknown, respectively. For example,

```
~p(A)
```

should read "not `p(A)`," and,

```
%p(A)
```

should read "it is unknown about `p(A)`," or "you cannot decide whether `p(A)` or `~p(A)`," because here we employ three-valued logic which includes *unknown* as a logical value besides usual *true* and *false*. A predicate `%p(A)` holds, when neither `p(A)` nor `~p(A)` is found in DB (database). It does not, when either `p(A)` or `~p(A)` is found.

3.3. Functions

A function represents an attribute of an entity. A function is a character string beginning with a lower case letter followed by a pair of brackets which encloses a list of objects, functions, and predicates separated by commas. Therefore,

```
f[A, b, X], g[p(x), q(y), h[A]]
```

are legal expressions of functions. Arithmetic operations, such as `+`, `-`, `*`, `/`, etc., are predefined functions.

Note that it is possible to define a function even on a set of predicates. This means a function can represent an *attribute value of a relationship*. Suppose we have two points `A` and `B` in a three dimensional Euclid space as entities. The distance, `d[A, B]`, between these two points can be defined by

TABLE 1. Truth Table

NOT

T	F
F	T
U	U

AND

∧	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR

∨	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

```
d[A, B] = abs[sqrt[(x[A]-x[B])**+(y[A]-y[B])**+(z[A]-z[B])**]],
```

which may naturally be used for defining the length of a line of which end points are `A` and `B`.

Function definition can be done by procedures. This means it is possible to write a function definition like

```
f[A, B] = { Xa := g[A], Ya := h[A],
            Xb := g[B], Yb := h[B],
            abs[sqrt[(Xa-Xb])**+(Ya-Yb)**]]}.
```

In this case, variables such as `Xa` are local variables and the value of the last statement becomes the value of the function.

3.4. Clauses

A clause is defined by a list of predicates combined by logical operators such as `&` (*and*) and `|` (*or*). Parentheses can be used to express the logical priority. The truth value of this clause will be calculated following the truth value table shown in TABLE 1.

3.5. Rules

A rule has the following syntax:

```
clause_1 -> clause_2.
clause.
clause_1 <-> clause_2.
```

The first rule is interpreted as "if `clause_1` is true, then `clause_2` must hold." This is called an assertion rule. If in DB `clause_1` is found, then `clause_2` is asserted. Assertion is done, so that the entire clause holds, which does not necessarily mean all the predicates in the clause hold. If for some reason it is impossible to assert the entire clause, the assertion fails. In general, the assertion mechanism is much the same as that of forward reasoning production rule systems and an assertion is done by atomic queries which result in set operations each of which corresponds to logical operation in the clause. The assertion is done based on rules described in TABLE 2.

For example, suppose we have a rule

```
p1(X) & p2(X) -> q(X).
```

and in DB we have facts,

```
p1(a), p1(b), p1(c), p2(a), p2(c).
```

Because the left hand side (LHS) is satisfied by instances, `a` and `c`, only facts

```
q(a), q(c)
```

are asserted (i.e., added to DB). In other words, the variable `X` in this clause is instantiated to constants `a` and `c`. When this instantiation is done, the system makes a list of a variable and instantiated constants. This

TABLE 2. Assertion Rule

Database before Assertion	Asserted Fact	Result of Assertion
p(X)	p(X)	Success and Returns True
p(X)	~p(X)	Assertion fails
p(X)	%p(X)	Success and Returns False
~p(X)	p(X)	Assertion fails
~p(X)	~p(X)	Success and Returns True
~p(X)	%p(X)	Success and Returns False
%p(X)	p(X)	Success and Adds this fact
%p(X)	~p(X)	Success and Adds this fact
%p(X)	%p(X)	Success and Returns True

list, called *instantiation pair list*, is used for controlling the deduction mechanism (see SECTION 3.7).

The second case is a special case of the first one; i.e., this clause is unconditionally asserted.

The third case is equivalent to the following two rules;

```
clause_1 -> clause_2.
clause_2 -> clause_1.
```

However, its semantics is different from that of these two rules in the sense that it works as a watch dog; i.e.,

```
p(X) <-> q(X)
```

works as a kind of constraints (as in deductive databases [Cha78]). This means, whenever this rules finds a fact which satisfies `p(X)`, this will be immediately converted to `q(X)` and *vice versa* , so that they are logically *equivalent.*

It sometimes happens that, although we want to assert a clause, we cannot find objects which satisfy that clause. In this case, a new object will be created. This object is given a system name rather than a user name as a constant. By adding this new predicate, it is possible that DB gets inconsistent. IDDS thinks it is a contradiction and the assertion fails.

Note that there is a difference in assertion of clauses between `&` (*and*) and `|` (*or*) logical operators. For instance, if a clause

```
p(X) & q(X)
```

is to be asserted and if the assertion of `p(X)` fails, the assertion of `q(X)` is not carried out and the entire assertion fails. In other words, the assertion of both `p(X)` and `q(X)` must succeed. On the other hand, in the case of

```
p(X) | q(X),
```

if the assertion of `p(X)` succeeds, the assertion of `q(X)` is not done. If the assertion of `p(X)` fails, the assertion of `q(X)` is done and the result of the entire assertion is dependent on it.

3.6. Built-In Predicates

There are so-called built-in predicates for arithmetic operations, controlling the inference, and for ultra-logic features. Arithmetic predicates include such comparison operators as =, <, and so on.

A built-in predicate `success` is to terminate the inference with information that the inference was a success. In the same way, a built-in predicate `fail` terminates the inference saying it was a failure. A built-in predicate `select` and `use` are used in the following syntax:

```
select(scenario_name, list_of_objects).
use(scenario_name, list_of_objects).
```

Here, `list_of_objects` can be omitted.

A `select` predicate changes the active scenario to `scenario_name` and restricts the active objects used in that scenario to `list_of_objects`. In the case `list_of_objects` is null, the active objects are not restricted. On the other hand, a `use` predicate adds `scenario_name` to the active scenario and restricts the active objects used to `list_of_objects`. A `use` predicate, therefore, increases available rules. These two predicates can be *true*, when the subscenario (see SECTION 3.7) is finished by the execution of a `success` built-in predicate or by *no-more-rule situation.* They can be *false*, when the subscenario is explicitly terminated by the execution of a `fail` built-in predicate.

This means a `select` predicate switches active scenarios, while a `use` predicate can show details of the presently discussed objects (FIGURE 5) using more dedicated rules. These two predicates are important to realize so-called multiworld mechanism (see SECTION 4.3).

In order to restrict active objects without changing scenarios, we can use also the following two predicates:

```
consider(list_of_objects)
unconsider(list_of_objects)
```

A `consider` predicate restricts the presently active objects to `list_of_objects` whereas an `unconsider` predicate makes objects in `list_of_objects` inactive.

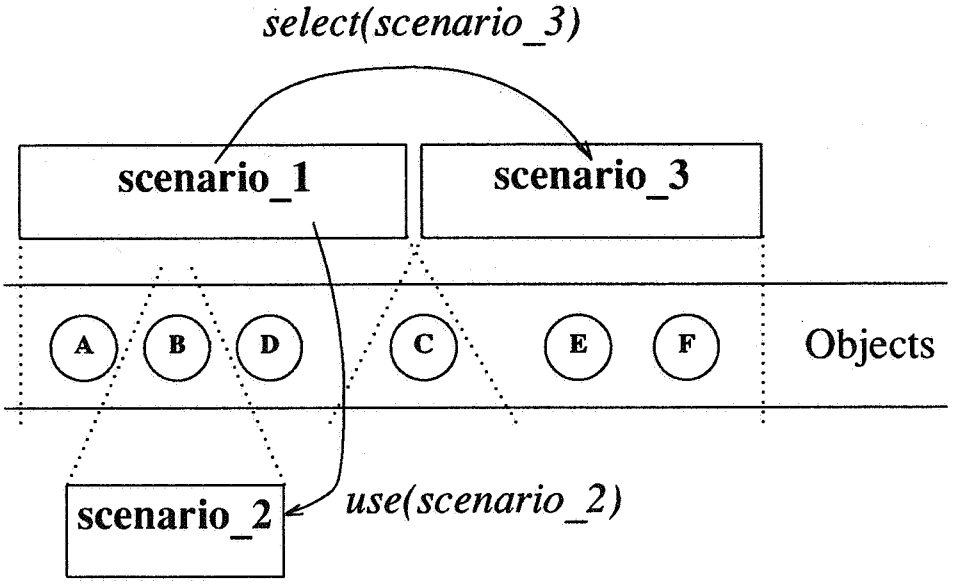


FIGURE 5. Multiworld Mechanism of IDDL

Because in IDDL input and output are described as messages to and from IUI or API, we can regard predicates for input and output as a kind of library routines. Therefore, we do not elaborate them particularly here. Note that they are in principle somewhat ultra-logic built-in predicates in the sense, for instance, we cannot erase all the side effects of those predicates.

3.7. Scenarios

3.7.1. Syntax of a Scenario

A scenario is a set of rules. A scenario set is a set of scenarios. They have the following syntax.

```
SCENARIO name (list_of_objects);
[OBJECT
    {rule;}*][1]
[FUNCTION
    {function_name = function_definition;}*]
[REPORTED
    {predicate_name BY system_element;}*]
[ASK
{predicate_name TO system_element;}*]
BEGIN
    {rule;}
SCENARIO END;

SCENARIO SET name (list_of_objects);
[OBJECT
    {rule;}*]
[FUNCTION
    {function_name = function_definition;}*]
[scenario*]
```

```
SCENARIO SET END
```

A scenario is active when control is passed to it. The argument `list_of_objects` defines active objects passed by a calling scenario. (More precisely, the *instantiation pair list* will be passed.) It may be used also for returning objects.

The object declaration in a scenario set defines *global* objects which can be used by all the scenarios in that scenario set. On the other hand, the object declaration in a scenario defines *local* objects which will never be seen from upper scenarios. At the same time, rules declared there works as constraints for restricting active objects as in deductive databases [Cha78] (see SECTION 3.5). For example, an object declaration,

```
OBJECT r(X) <-> q(X);
```

implies that, whenever `r(X)` is asserted, `q(X)` is automatically asserted. Therefore, this works as a conversion rule. When there is no restriction for objects used here, we use special predicate `ANY_OBJECT`. These features are almost corresponding to the idea of typing in conventional languages.

The difference between *local* and *global* objects is the following. Although both types of objects are stored in DB, local objects can be accessed only from the scenario in which they are defined. When the execution of that scenario terminates, those local objects are automatically removed from DB. However, global objects remain in DB until they are explicitly removed. The function declaration must be written in a pair with an object declaration.

The `REPORTED` and `ASK` sections specify from and to which system element the predicate should be reported or asked for assertion. For example,

```
REPORTED p(X) BY IUI
```

means a predicate `p(X)` can be reported by IUI at *any time* reported while this scenario is active. When `p(X)` is reported, `X` must be instantiated. On the other hand,

```
ASK q(Y) TO API
```

means a predicate `q(Y)` can be asked to API, if we need to know whether or not this predicate is true. Variables in the predicates are used for passing objects. If `q(Y)` is not instantiated before the assertion, `q(Y)` must be instantiated after assertion. If `q(Y)` is instantiated before the assertion, the constants which are the instances of this variable are passed to API.

3.7.2. Execution of a Scenario

A scenario will be executed in the following way. First, examining the rules from the top, the first rule of which LHS is satisfied is selected. Second, the right hand side (RHS) of this rule is asserted (or, executed).

The execution of a rule can terminate either successfully or in failure. If the assertion was successful, search for the next matching rule begins from the next of the previously executed rule. If the assertion failed, once again search begins and another rule will be selected. In the case the execution terminated *successfully*, all the results will be preserved. On the other hand, in the case of *failure*, all the results will be removed, which means we have a trial-and-error method.

If the search for the next applicable rule comes back to the most recently executed rule because the search is wrapped around, it is judged that there is no more rule applicable and the execution of the entire scenario stops (*no-more-rule situation*). The execution of a scenario can be also stopped by the execution of either success or fail built-in predicates.

When a rule is selected, the instantiation pair 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.

4. Organization of Design Knowledge

In CHAPTER 2, we have proposed the concept of IIICAD to which knowledge engineering is applied. The supervisor (SPV) who is controlled by scenarios written in IDDL plays the most important role in a IIICAD environment. A very interesting question is whether or not it is possible to organize and represent design

[1] The syntax { }* means iteration more than 0 times.

knowledge in IDDL which was briefly defined in CHAPTER 3; and if possible at all, how knowledge written in IDDL can contribute to realization of a IIICAD environment described in CHAPTER 2.

In SECTION 4.1, we analyze so-called design knowledge based on the model proposed in SECTION 2.2. In the succeeding sections, we see how it can be represented by IDDL through a couple of examples. In SECTION 4.2 we describe how a design task is performed by SPV based on scenarios in the IIICAD architecture and how IIICAD realizes a flexible and intelligent user interface. We will, then, see in SECTION 4.3 how simple mechanisms of IDDL materialize IIICAD features which are important for designing. By doing this, we know what we can do and what we cannot in IDDL. This will be fed back to the discussion in SECTION 4.4 about distribution of design knowledge in the IIICAD architecture. Because IUI and API have basically the same functions as SPV, except for data passing, and are also described by IDDL, they will not be discussed here.

4.1. Design Knowledge

Here, we analyze so-called design knowledge, so that later we can consider whether it is possible to describe it in IDDL. To begin with, we consider designing in mechanical engineering phenomenologically.

First of all, there are many types of design; completely new design where we must invent the mechanism; parametric design where we know already the basic mechanism; combinatory design where we know parts and the problem is combination of those parts. New design requires knowledge about basic physical phenomena. Parametric design does not require such knowledge, but it requires knowledge about how to make the mechanism fit to the specifications. Combinatory design does not require knowledge about entities in detail, but it requires knowledge on how to combine existing parts (TABLE 3). This means we need four types of knowledge as follows.

(a) Knowledge on the existing entities.

(b) Knowledge (on basic principles such as physics) to invent the structure and mechanism of new entities.

(c) Knowledge to operate entities including constraints solving, compromizing and relaxation of the specifications, etc.

(d) Knowledge to combine existing entities without changing their inner structure.

Designing a product itself contains different kinds of subprocesses where different types of knowledge are needed. There are at least three stages in mechanical design; conceptual design, basic design, detail design (FIGURE 6). In conceptual design we need knowledge on basic physical phenomena, so that we can think of a appropriate mechanism for the specifications. Basic design requires knowledge on how to combine existing parts and knowledge to predict the behavior of the candidate as the result of the combining (for example, knowledge for technological analyses). In detail design, we need to determine details of parts, just like parametric design. In summary, a design process of machine needs the following three types of knowledge besides the knowledge about the existing entities (a).

(b') Knowledge for determining the structure and the mechanism of an entity, which is typical in conceptual design.

(c') Knowledge to detail existing parts, keeping the constraints (i.e., specifications).

(d') Knowledge to construct the structure of the design solution using existing entities, keeping the constraints.

TABLE 3. Classification of Design

	Creation of Structure	Combination	Detailing of Structure
New Design	○	○	○
Combinatory Design	-	○	∇
Parametric Design	-	-	○

○: *strongly required*; ∇: *required*; -: *not required.*

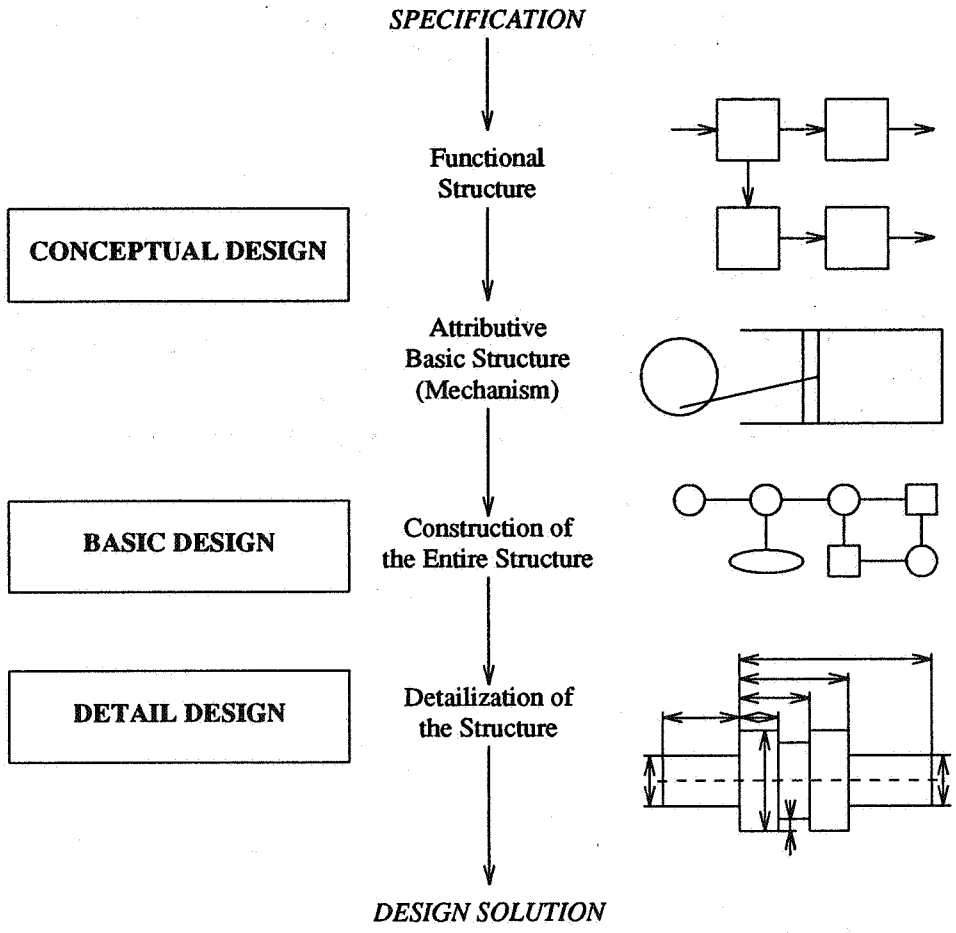


FIGURE 6. Design Process

The aims of IIICAD are to have models of the design object and to check if the design solution satisfies the specifications or not (SECTION 2.3). This idea resulted in a design process model called *metamodel evolution model*, illustrated in FIGURE 2, which requires knowledge for evaluation in addition to knowledge on design object models and their operations.

(e) Knowledge to select proper evaluations for checking the functions of the design solution.

(f) Knowledge to create a model for evaluation from the central model.

(g) Knowledge to perform evaluation.

(h) Knowledge to judge the result of evaluation. (This knowledge can be very heuristic.)

4.2. Examples of Design Knowledge in IDDL

This section and the next section deal with IDDL's possibilities to describe design knowledge for machine design. In this section, firstly, we see that procedural knowledge can be represented in IDDL. Secondly, as an example of non-deterministic knowledge, how to realize a flexible user interface is shown.

4.2.1. A Simple Case

Now, we see how SPV (supervisor) works in general, particularly, in a simple procedural style and how it controls the design process. SPV, by definition, should be able to control the design process by means of scenarios. We take an example of the designing of a winch from SECTION 2.5.2.

The main scenario shown below is used just for selecting the design subject. Note that in this scenario there is only one rule which will be unconditionally executed, because its left hand side (LHS) is null. Its right hand side (RHS) has two clauses which will be executed sequentially in this order. The variable `Subject` is local and supposed to be instantiated to one of the design subject names.

```
OBJECT
    scenario_name(Subject);
SCENARIO main;
BEGIN
    select(scenario_select, Subject) &
    select(Subject);
SCENARIO END;
```

The scenario for selecting subscenarios might be coded in a following way. By declaring `design_command` predicate at the `REPORTED` section, IUI is ready for accepting commands from the user. (The actions of IUI for this predicate must be described in detail in a scenario for IUI.) The rule holds, only if there is no user input that makes the `design_command` predicate either *true* or *false*. Because of this, unless there is an input of `design_command` it tries to get an input from the user.

```
SCENARIO scenario_select(Subject);
REPORTED
    design_command() BY IUI;
ASK
    accept_design_subject() TO IUI;
    message() TO IUI;
BEGIN
    %design_command(Subject) ->
        message("What do you want to design?") &
        accept_design_subject(Subject) &
        design_command(Subject);
SCENARIO END;
```

The next one is the main scenario for designing a winch. First, it asks the name of a winch. Then, it asks approximate load specification and it decides the winch type. After that, it proceeds to detail designing, such as designing of the wire, the gear mechanism, the safety mechanism, and the casing. Note that this rule will be executed only once, because at the end there is a `success` build-in predicate which terminates the execution of this scenario. If we think about iteration of the designing caused by unsatisfaction of the specifications, having only this rule is insufficient; we need to have either extra rules for error handling or some scenarios which are active when this scenario is used (see SECTION 4.3).

```
SCENARIO winch_design();
OBJECT
    winch(Winch);
BEGIN
    select(ask_design_object_name, "winch", Winch) &
    select(ask_approximate_load, Winch) &
    select(decide_winch_type, Winch) &
    select(wire_design, Winch) &
    select(gear_mechanism_design, Winch) &
    select(safety_mechanism_design, Winch) &
    select(casing_design, Winch) &
    success;
SCENARIO END;
```

The next scenario describes how to get a user input. Firstly, it asks the user to give a name of the design object by issuing a prompt. It is important that IUI knows only that what is required is a character string. Therefore, it can test the input syntactically. This means the semantical check for the input is left to other scenarios, although it can reject, for instance, an input of a point from the mouse. Needless to say, IUI should have scenarios for `message` and `accept_string` predicates.

```
SCENARIO ask_design_object_name(Name, Object);
OBJECT
    ANY_OBJECT(Name);
    ANY_OBJECT(Temp);
FUNCTION
    name[Object] = STRING;
ASK
    accept_string() TO IUI;
    message() TO IUI;
BEGIN
    message("What is the name of ", Name, "?") &
    accept_string(Temp) &
    name[Object] := Temp &
    success;
SCENARIO END;
```

4.2.2. Realization of Flexible User Interface

As described in SECTION 2.1, IIICAD system should realize flexible and intelligent user interface. Here, we show how it will be achieved by IDDL.

Suppose we are going to design a cube shown in FIGURE 7. It has the following attributes and constraints.

- depth ($d > 0$)
- height ($h > 0$)
- width ($w > 0$)
- volume ($v = d * h * w$)
- color ($c =$ *one of red, blue, yellow, and black*)
- material ($m =$ *one of steel, aluminum, plastic, and wood*)
- unit weight ($u =$ *an attribute of the material,* > 0)

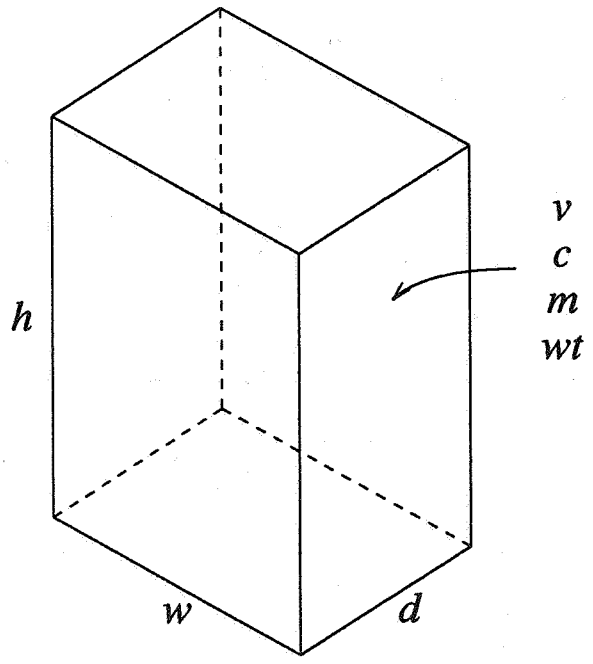


FIGURE 7. Design of a Cube

- weight ($wt = u * v$)

The scenario to design this cube can be described as follows.

```
OBJECT
    cube(Cube);
    material(steel); material(aluminum);
    material(plastic); material(wood);
    color(red); color(blue); color(yellow); color(black);
FUNCTION
    w[Cube] = REAL; h[Cube] = REAL; d[Cube] = REAL;
    v[Cube] = w[Cube] * h[Cube] * d[Cube];
    m[Cube] = SELECTED FROM steel, aluminum, plastic, wood;
    u[Cube] = unit_weight[m[Cube]];
    wt[Cube] = u[Cube] * v[Cube];

SCENARIO cube_design(Cube);
OBJECT
    real_positive_number(X);
    material(M);
    color(C);
REPORTED
    attribute_input() BY IUI;
ASK
    attribute_req() TO IUI;
BEGIN
    %fixed(depth) & attribute_input("Depth", X) ->
        fixed(depth) & d[Cube] := X;
    %fixed(height) & attribute_input("Height", X) ->
        fixed(height) & h[Cube] := X;
    %fixed(width) & attribute_input("Width", X) ->
        fixed(width) & w[Cube] := X;
    %fixed(material) & attribute_input("Material", M) ->
        fixed(material) & m[Cube] := M;
    %fixed(color) & attribute_input("Color", C) ->
        fixed(color) & c[Cube] := C;
    fixed(depth) & fixed(height) & fixed(width) ->
        fixed(volume);
    %fixed(volume) & attribute_input("Volume", X) ->
        fixed(volume) & v[Cube] := X;
    fixed(material) & fixed(volume) ->
        fixed(weight);
    %fixed(depth) & fixed(volume) ->
        attribute_req("Depth", v[Cube]/w[Cube]/h[Cube], X) &
      d[Cube] := X & fixed(depth);
    %fixed(width) & fixed(volume) ->
        attribute_req("Width", v[Cube]/d[Cube]/h[Cube], X) &
      w[Cube] := X & fixed(width);
    %fixed(height) & fixed(volume) ->
        attribute_req("Height", v[Cube]/d[Cube]/w[Cube], X) &
      h[Cube] := X & fixed(height);
    fixed(weight) & fixed(color) ->
        success;
SCENARIO END;
```

Using this scenario, now the user can input attributes of a cube, such as the depth, height, width, etc., in any order. Note that once an attribute, for example, the height, is fixed, further input concerning the height is blocked, because a fact `fixed(height)` is asserted. (However, at this moment there is no way to change

attributes which were once fixed.) The execution of this scenario is completed when both the color and the volume is fixed.

An interesting feature of this scenario is that the user is allowed to input the value of volume independently of the width, the depth, and the height in any order. This suggests that it is possible that the volume is specified without particular values for the width, the depth, and the height. The constraint among these four variables will be automatically satisfied by calling a subscenario, `attribute_req`, which is supposed to get a value of an attribute from the user showing an expected value in the second argument.

In case the designer specified a value for the depth, contradictory to the constraint, the assignment `d[Cube] := X` and hence the rule fail. This causes eventually a *no-more-rule situation* and the execution of scenario terminates. In an upper level scenario which called this scenario this situation can be detected, because there will be no `fixed(volume)` clause asserted in DB. It should be handled by error handling facility of the multiworld mechanism (see SECTION 4.3.2).

4.3. Multiworld Mechanism

In this section we discuss an important issue in IDDL, multiworld mechanism. The multiworld mechanism (see FIGURE 5 in SECTION 3.6) is useful for having multimodels derived from one central metamodel, exception handling (such as errors and user's questions), etc.

4.3.1. Realization of Multiworld Mechanism

The `use` built-in predicate passes control to a *subscenario* (FIGURE 5). The following scenario shows how it is achieved.

```
SCENARIO scn1;
OBJECT
    p(a);
    q(b);
BEGIN
    p(A) -> r(A);
    q(A) -> s(A);
    r(A) -> use(scn2, A);
SCENARIO END;

SCENARIO scn2(A);
OBJECT
    an_object(A);
BEGIN
    u(A) -> v(A);
SCENARIO END;
```

If in the subscenario `scn2` there occurs so-called *no-more-rule situation*, automatically control is given back to the parent `scn1`. Note that at this moment only objects instantiated to `A` can be used in `scn2`. In this case, only `a` is used in `scn2`. If in `scn2`, either `success` or `fail` predicate is asserted, the `use` predicate in `scn1` itself is regarded as *succeeded* or *failed*. (See FIGURE 8.)

This means by using `scn2` we can create a different world, w_2, from that of `scn1`, w_1. In w_2, we can only use an object `a`. Therefore, there is *property inheritance* of `a` from w_1 to w_2 and *vice versa*. Note that, however, any operations resulting from w_2 can be erased in w_1 by making the execution of `scn2` failed.

At the same time, this also means we can write metalevel knowledge for controlling at a global level in the parent scenario. Built-in predicates, such as select, use, success, and fail, are originally used for the controlling purpose; i.e., they are denoting the metalevel knowledge. By using those predicates, we can write reasoning control knowledge in higher level scenarios. This is one of the important features of IDDL.

Global objects can be manipulated in both parent scenarios and subscenarios, while local objects can be generated and manipulated only in subscenarios. This means changes of global objects from a subscenario are visible to the parent scenario but those of local objects are invisible. Thus, we have now complete separation of subworlds in subscenarios from the world of the parent scenario.

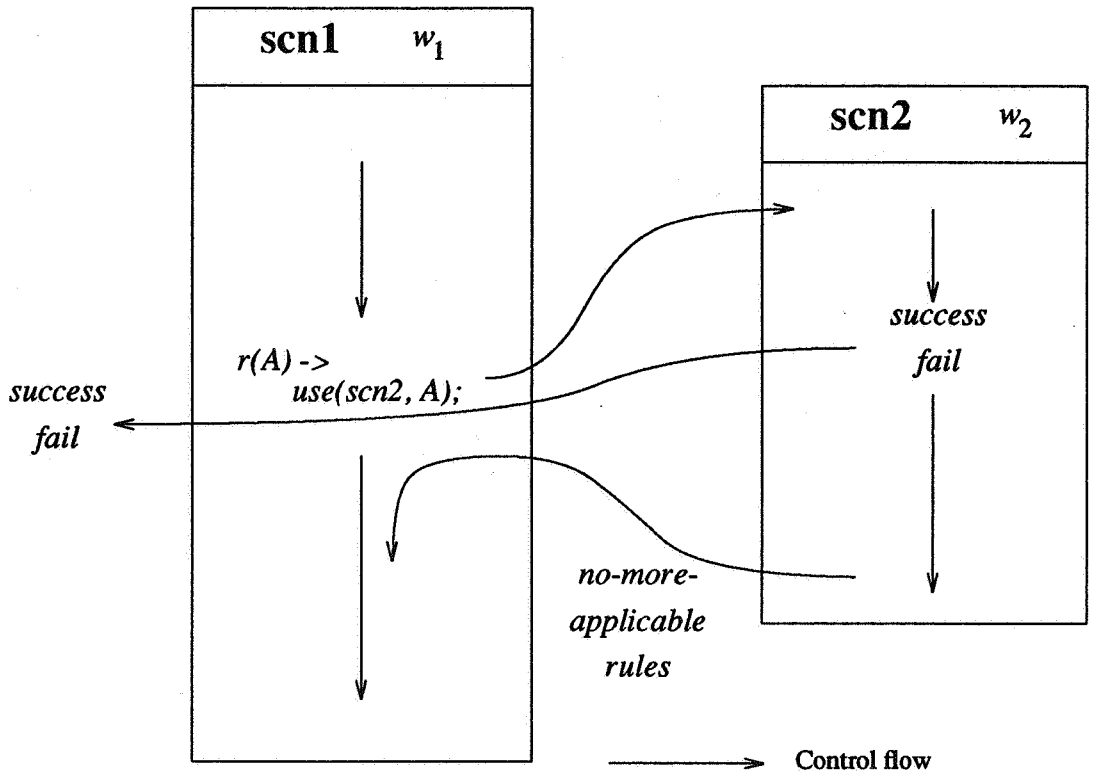


FIGURE 8. Multiworld Mechanism

4.3.2. Exception Handling

There can be several possibilities that the system faces an unexpected input from the user; i.e., the system cannot find the most appropriate rule from the current scenario. IDDL's multiworld mechanism enables also to deal flexibly with such exceptions including errors. It can be also used for handling questions from the user.

These situations can be classified as follows.

(1) The user made an error.

(2) The user is talking about something different, or the system does not know what the user is thinking about.

(3) The user does not know the proper usage of the system.

(4) The user asks questions.

By writing error handling scenarios which can be used commonly by subscenarios in the parent scenario, we do not need to write all the exception handling rules in individual scenarios. Suppose the user inputs an unexpected command. If the system cannot find the most appropriate rule, it is regarded as a *no-more-rule situation.* In this case, the control is returned to the parent scenario (see FIGURE 8).

4.3.3. Realization of Metamodel Mechanism

As pointed out in CHAPTER 2, one of the key issues of IIICAD is a mechanism for describing the central model of the design object (metamodel) and for deriving dedicated models for evaluation from it. In this section, we discuss how this mechanism is supported in IDDL by the multiworld mechanism.

Imagine we have a machine part which looks like a triangle [ToY87]. FIGURE 9 (a) is its two dimensional representation; in FIGURE 9 (b), one of its corner is chamfered, while in FIGURE 9 (c) it is rounded. From a

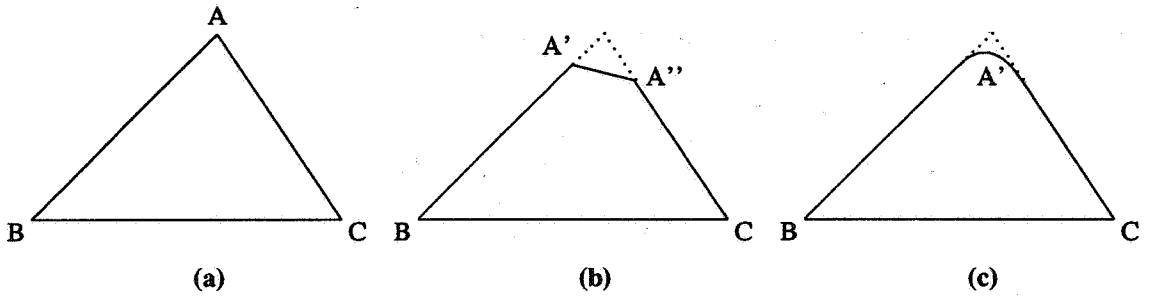


FIGURE 9. Three Similar Figures

view point of a mechanical engineer, we can say that FIGURE 9 (a) is a rough sketch of this part and that FIGURE 9 (b) and (c) are more detailed drawings for manufacturing. This means that these three are basically identical and that, if something is changed in FIGURE 9 (a), this change should propagate to other two properly and *vice versa*. In this context, FIGURE 9 (a) is the metamodel and (b) and (c) are models derived from (a).

The following example shows that IDDL allows to have different models derived from the metamodel. (To write this example, [ArW87] was a great help.)

A polygon `POLG` is generated by the following rules. (In the following scenario, `POLG` means a polygon; strings beginning with `L` and `l` indicate lines; `P` and `p` indicate points.)

```
OBJECT
    line(L) <->
        point(P1) & point(P2) &
        has(L, P1) & has(L, P2) &
        different(P1, P2) &
        endpoint(L, P1, P2);
SCENARIO create_a_triangle(T);
OBJECT
    point(P); point(P1); point(P2);
    line(L); line(L1);
    integer(M); integer(N);
    ANY_OBJECT(POLG);
BEGIN
    polygon(T, 3) & success;
    polygon(POLG, N) ->
        startpoint(P) & endpoint(L, P, P1) &
        M := N - 1 & create(POLG, L, M);
    create(POLG, L, 0) ->
        startpoint(P2) & endpoint(L, P1, P2) &
        has(POLG, L) & has(POLG, P1) & has(POLG, P2);
    create(POLG, L, N) ->
        endpoint(L, P1, P2) &
        has(POLG, L) & has(POLG, P1) & has(POLG, P2) &
        M := N - 1 & endpoint(L1, P2, P3) &
      different(L, L1) & create(POLG, L1, M);
SCENARIO END;
```

Here, a fact, `startpoint(P)`, is used for marking one of the vertices of a polygon, so that we can *draw* edges. A predicate, `endpoint(L, P1, P2)`, reads a line `L` has two end points, `P1` and `P2`. A predicate, `has(A, B)`, is representing so-called a part-assembly relationship that `A` owns `B`. A predicate, `different(X, Y)`, is true when `X` and `Y` are referring to different objects; otherwise, false.

A triangle `t` is created by `create_a_triangle` as follows;

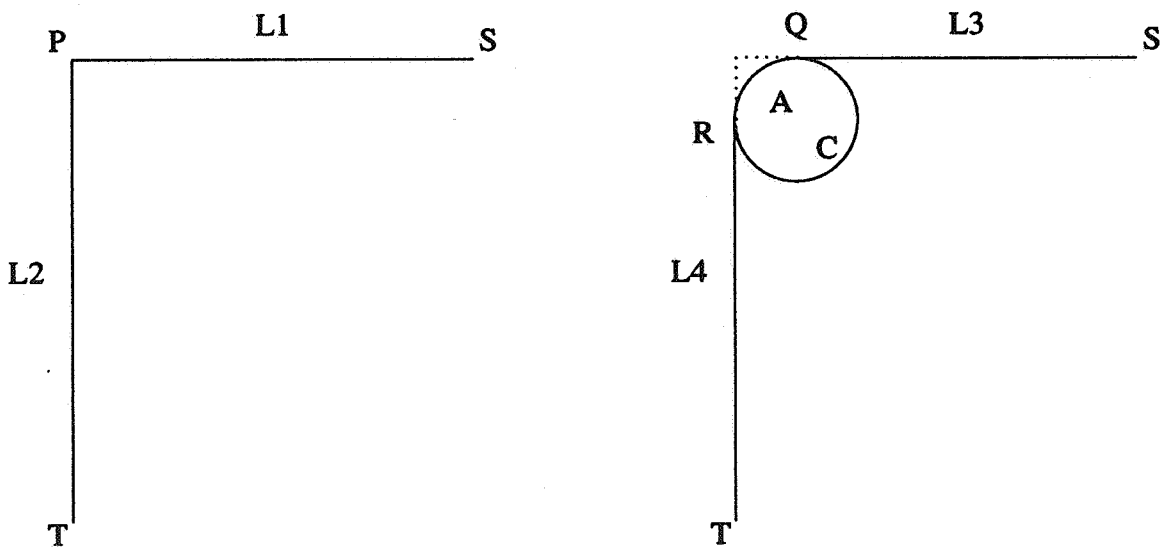
```
select(create_a_triangle, t).
```

Consequently, the following facts will be asserted and added to DB.

```
startpoint(p1).
has(t, l1).  has(t, l2).  has(t, l3).
has(t, p1).  has(t, p2).  has(t, p3).
has(l1, p1).  has(l1, p2).  has(l2, p2).
has(l2, p3).  has(l3, p3).  has(l3, p1).
point(p1).  point(p2).  point(p3).
endpoint(l1, p1, p2).
endpoint(l2, p2, p3).
endpoint(l3, p3, p1).
```

Now, let us consider rounding the corner `p2`. (Chamfering is described in the same way.) First of all, we need knowledge for rounding a corner (FIGURE 10).

```
SCENARIO round_a_corner(P, L1, L2, A, Q, R, L3, L4);
OBJECT
    point(P); point(Q); point(R);
    line(L1); line(L2); line(L3); line(L4);
    circle(C); arc(A);
BEGIN
    on(Q, L1) & different(P, Q) &
    on(R, L2) & different(P, R) &
    tangent(Q, C, L1) & tangent(R, C, L2) &
    endpoint(A, Q, R) & has(C, A) &
    has(A, Q) & has(A, R) &
    endpoint(L1, S, P) & endpoint(L3, S, Q) &
    tangent(Q, C, L3) &
    endpoint(L2, P, T) & endpoint(L4, T, R) &
    tangent(R, C, L4) &
```



(a) Before Rounding **(b) After Rounding**

FIGURE 10. Rounding a Corner

```
        convex_arc(A, L3, L4) &
        success;
    SCENARIO END;
```

A predicate, `on(P, L)`, holds, when a point `P` is on a (definite) line `L`. A predicate, `tangent(P, C, L)`, holds, when a line `L` is tangent to a circle `C` at a point `P`. A predicate, `convex_arc(A, L3, L4)`, holds, when an arc `A` form a convex arc corner of `L3` and `L4`.

Let `remove` and `add` predicates be explicit removal and addition of objects to and from DB. Then, we can actually round a corner by the following assertion.

```
select(round_a_corner, p2, l1, l2, A, R, Q, L1, L2) &
remove(p2) & remove(l1) & remove(l2) &
add(A) & add(R) & add(Q) & add(L1) & add(L2).
```

After this assertion, we have the following facts in DB.

```
tangent(q, cir, l1).  tangent(r, cir, l2).
tangent(q, cir, m1).  tangent(r, cir, m2).
circle(cir).  has(cir, a).
convex(a, l3, l4).
arc(a).
has(m1, p1).  has(m1, q).
has(m2, p3).  has(m2, r).
endpoint(m1, p1, q).  endpoint(m2, p3, r).
endpoint(a, q, r).
has(a, q).  has(a, r).
point(q).  point(r).
on(q, l1).  on(r, l2).
startpoint(p1).
has(t, l3).  has(t, p1).  has(t, p3).
has(l3, p3).  has(l3, p1).
point(p1).  point(p3).  endpoint(l3, p3, p1).
```

Comparing the state of DB before the rounding and after, we find out the following properties of our metamodel mechanism.

(1) We have obtained complete separation of the states of DB before the rounding and after. This can be formalized as follows. Before the rounding operation we had a central model M (i.e., the first state of DB), and after the operation it became M_r (i.e., the second state of DB). The transition from M to M_r was carried out by rounding operations, R. We can write it as;

$$M_r = R(M).$$

(2) The fact that M and M_r are separately established in different worlds means we have complete separation of models as well as inheritance of attributes and their changes. Changes in M_r can be, for instance, propagated to M. Therefore, if we regard M as the central model which is necessary for the integration of CAD systems, we can easily obtain different models by having a scenario to derive them from M, such as R (see FIGURE 11). (Which objects or predicates can be seen from the parent scenario is totally dependent on the declaration of objects, i.e., local or global.)

(3) The multiworld mechanism provides a try-and-error method. Suppose for some reason the procedure to round a corner failed. All the effects of this procedure will be removed, if it has failed.

(4) There were predicates, like `on(P, L)`, which cannot be directly described only by predicate logic. This problem is solved by introduction of functions. For instance, a predicate `on(P, L)` is defined by

```
on(P, L) = { point(P);
             line(L);
             a[L]*x[P]+b[L]*y[P]+c[L] = 0;}
```

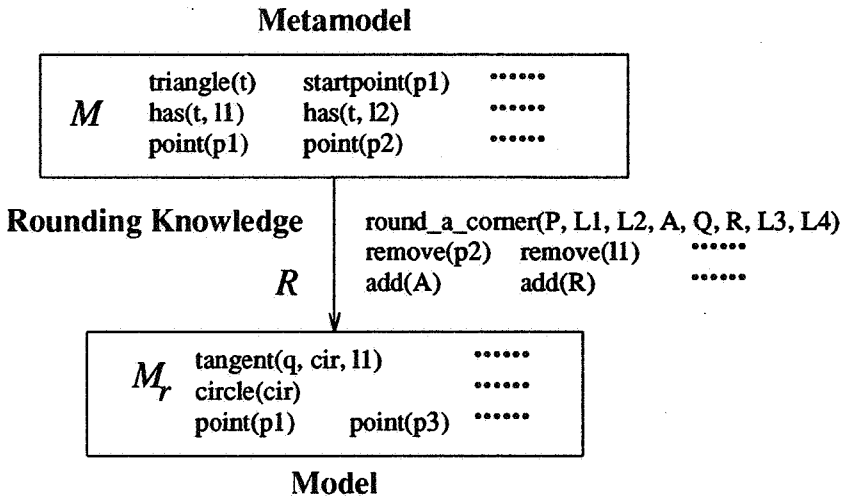


FIGURE 11. Metamodel and Model

where `x[P]` and `y[P]` stand for the coordinates of the point `P`, and `a[L]`, `b[L]`, and `c[L]` stand for the coefficients of the line equation of `L`.

4.4. Distributing Knowledge in the IIICAD Architecture

4.4.1. Limitation of the Pseudo Language (IDDL)

In SECTION 4.1 we analyzed design knowledge. We counted up eight categories of design knowledge which are shown below.

(A) Knowledge about existing entities.

(B) Knowledge to determine the inner structure based on fundamental principles (such as physics) and to verify the feasibility of the existence of entities.

(C) Knowledge to manipulate entities for detailing without changing the inner structure of an entity drastically.

(D) Knowledge to combine existing entities without changing their inner structure.

(E) Knowledge to select proper evaluation to check the functions of the design solution.

(F) Knowledge to derive models from the metamodel.

(G) Knowledge for evaluation of models.

(H) Knowledge to judge the result of evaluation.

Although our pseudo language, or IDDL, is a very simple language, it has several powerful features (see CHAPTER 3).

(1) Representing design objects by means of features to describe objects, predicates, and functions. An object corresponds to an entity; a function to an attribute of an entity; a predicate to a relationship between entities.

(2) Representing design processes as scenarios, i.e., sets of rules, based on deductive reasoning. If we regard IDDL as a deductive reasoning system, objects, functions, and predicates are axioms and scenarios are rules.

(3) The multiworld mechanism to allow different kinds of views (or models) for various purposes; for instance, derivation of different models from the metamodel, implementation of backtracking mechanism, etc.

However, these features do not necessarily mean that IDDL has capability to describe all the knowledge from (A) to (H). Thus, there arises an interesting question,

"How can we incorporate those different types of knowledge in the IIICAD architecture?"

For instance, the analogy in (2) makes us realize that by using IDDL (or similar language) we are not able to create what will not be deduced from the axiom and rules defined beforehand. Furthermore, empirical knowledge, such as knowledge (H), might be very difficult to describe in a predicate logic based language such as IDDL, because that type of knowledge requires dedicated knowledge representation languages if possible at all. This suggests, although IDDL is powerful, still we cannot write down everything in it and we need to distribute the knowledge in the architecture of IIICAD. Let us examine this issue in more detail here.

(A) *Knowledge about existing entities.* An existing entity can be represented as an object in IDDL. Its attributes may be described as functions. Predicates denote its relationships between other entities. These descriptions are stored in DB through IDDS. Note this type of knowledge can be declarative and, in machine design, can be sometimes given even in a form of catalogues.

(B) *Knowledge to determine the inner structure based on fundamental principles and to verify the feasibility of the existence of entities.* This type of knowledge is rather difficult to describe in IDDL, because it may include modification and generation of new objects and predicates, which means operations to modify and add descriptions about new entities to knowledge (A). Therefore, it will be embedded outside IDDS, for instance, in an external and independent knowledge based system, rather than in scenarios.

(C) *Knowledge to manipulate entities for detailing without changing the inner structure of an entity drastically.* This can be described in IDDL together with knowledge (A), since it is typical design process knowledge to which the concept of scenarios especially is appropriate. However, this knowledge may include very heuristic or empirical knowledge.

(D) *Knowledge to combine existing entities without changing their inner structure.* This type of knowledge typically appears in so-called combination design and its considerable part can be reduced into constraints solving, optimization, linear programming, etc. Therefore, in case it can be represented as routine design process knowledge like knowledge (C), IDDL can be used; otherwise, it will be incorporated in external applications.

(E) *Knowledge to select proper evaluation to check the functions of the design solution.* In IIICAD it is written in IDDL and stored in DB as scenarios for SPV and API. Note that to perform such evaluations we might need knowledge about the fundamental principle such as physics like knowledge (B).

(F) *Knowledge to derive models from the metamodel.* This knowledge can be also written in IDDL and stored in DB as scenarios like knowledge (C), since we are able to know about evaluation methods beforehand.

(G) *Knowledge for evaluation of models.* Since application programs for evaluation, such as FEM programs, are usually very huge and can be only utilized as external packages, this will be placed outside IDDL.

(H) *Knowledge to judge the result of evaluation.* This is typically heuristic or empirical knowledge in machine design. It might be even inappropriate to leave it with CAD and should be judged by the designer.

Although IDDL is a simple but yet powerful language, it is obviously difficult to write down all the design knowledge. There are two reasons for this.

1. IDDL's scenario mechanism might be suitable to describe procedural knowledge which can be nevertheless non-deterministic rather than heuristic one which is not only non-deterministic but also hard to formalize. Heuristic knowledge can be implemented, heavily depending on backtracking, unification, etc. Although IDDL can support such features, practically its performance cannot be so good as that of dedicated reasoning systems tuned for a particular problem.

2. In SECTION 4.3.3 we have seen a predicate `on(P, L)` is implemented arithmetically rather than logically. If we have a predicate that requires much more complicated calculations, the

performance of the logical implementation is almost hopeless. We should use applications written in conventional languages through API.

Therefore, these two types of knowledge will be placed outside IDDL in IIICAD as external applications. This creates, however, another problem, i.e., how to realize communication between knowledge distributed in the IIICAD architecture from scenarios to applications. The next section deals with this problem.

4.4.2. Mechanism for Distributing Knowledge

In SECTION 2.3, we did not particularly specify what kind of applications were necessary for IIICAD. We assumed, as application programs for machine design, three dimensional modeling system, characteristics analysis programs (FEM, etc.), and so-called knowledge based systems for tasks which requires heuristic knowledge, etc. Together with knowledge in these applications, knowledge written in scenarios plays an important role in IIICAD. In other words, all those subsystems should form special distributed *knowledge libraries* which realize the integration of knowledge.

The logical connection between applications and IDDS is done by API and an application program is regarded as a predicate assertion mechanism. FIGURE 12 illustrates this idea. (Note that this feature does not contradict to the definition of the current version of IDDL defined in CHAPTER 3. Even if the REPORTED or ASK section of a scenario specifies the origin or destination of predicate assertion, this mechanism can be incorporated.)

Suppose in a scenario for SPV we need to know whether `p(c)` holds or not. Either *true*, *false*, or *unknown* is returned to such an inquiry. If *true* or *false* is returned, variables in a clause are instantiated. This is first asked to DB (arrow 1 in FIGURE 12). DB keeps all the asserted clause, until objects in the clause are changed. If for some reason DB does not know it, i.e., if *unknown* is returned, SPV asks API (arrow 2). API, given its own scenario, knows to which application it should ask and invokes that particular application, in this case AP1 (arrow 3). If AP1 either does not know or fails to answer, SPV asks IUI, i.e., the designer (arrow 4).

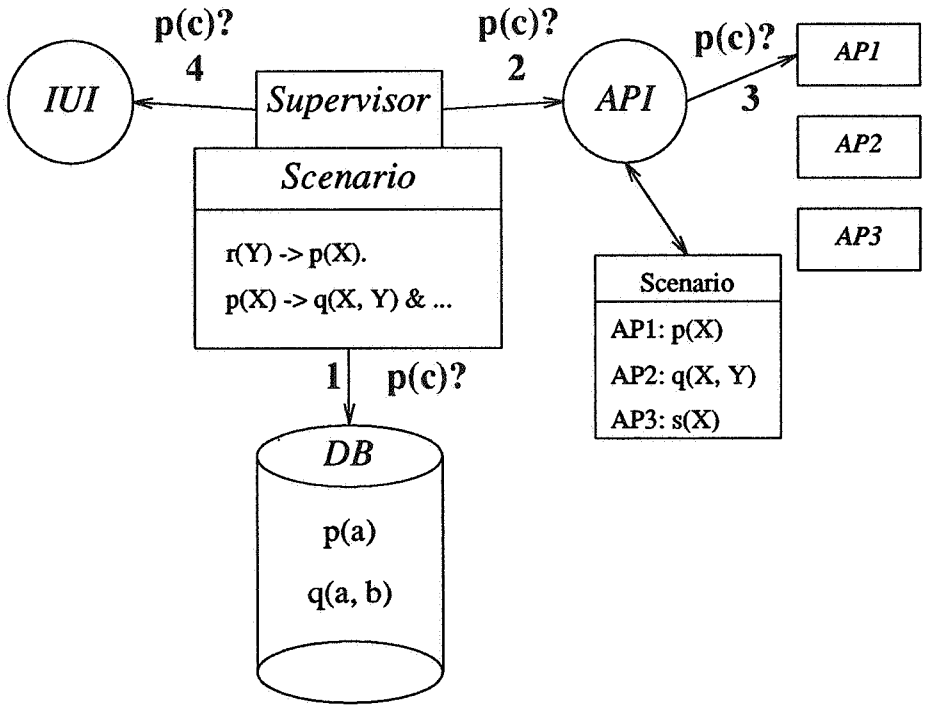


FIGURE 12. Knowledge Distribution and Organization by Predicate Assertion Mechanism

An important issue here is that, from a viewpoint of SPV, invoking applications, asking DB, and asking IUI can be treated equally; they are equal in the sense of being able to confirm a predicate. This usage of applications suggests how to develop a IIICAD system; it will not be developed on top of existing CAD systems. We must first decompose an existing CAD system into small modules according to their functions. Each module becomes a predicate assertion mechanism for a IIICAD system. In this context, such small modules form distributed knowledge libraries.

5. Concluding Remarks

(1) The concept of IIICAD was proposed. The IIICAD system has intelligent components, such as Supervisor (SPV), Application Program Interface (API), and Intelligent User Interface (IUI), and Integrated Data Description Scheme (IDDS) for the integration of the design knowledge. It is based on a design process model called metamodel evolution model.

(2) The system is driven by scenarios written in a language called IDDL based on predicate logic and object oriented programming paradigm.

(3) The SPV mechanism was discussed based on scenarios written in IDDL. We could show that IIICAD would provide a flexible user interface and an integrated environment for describing the design knowledge.

(4) The multiworld mechanism of scenarios was illustrated. This gives capability of global design process control, metamodel description and derivation of models from the metamodel, exception handling, and so on.

(5) Design knowledge was analyzed. There are eight kinds of knowledge. Because it is rather difficult to represent all in IDDL, distributing the design knowledge to applications is considered to be optimal.

(6) A mechanism for organizing distributed design knowledge was illustrated. It will be actualized by a simple facility of IDDL.

Concerning future work, we mention two issues. First, the specifications for IDDL should be further elaborated and implemented. We are in the process of developing an experimental version on Smalltalk-80 [GoR83].

Second, much more efforts should be spent on capturing task specific knowledge. In case of machine design, there are several studies to represent heuristic knowledge in a more rational way, such as design methodology [Rod71]. We need formalization and computerization of that kind of methodology. It is pointed out that we need somewhat *deeper knowledge* [Mac86] to make the system understand the common sense knowledge of machine design and to make the system innovative. Qualitative physics [Bob84] is regarded as the starting point for this approach. These two views should be taken into consideration when we extend the IDDL specifications.

Reference

[ArW87] F. ARBAB and J. M. WING, "Geometric Reasoning: A New Paradigm for Processing Geometric Information," in *Design Theory for CAD, Proceedings of the IFIP W.G. 5.2 Working Conference 1985 (Tokyo)*, H. YOSHIKAWA and E. A. WARMAN (eds.), North-Holland, Amsterdam, (1987), pp. 145.

[Bob84] D. G. BOBROW (ed.), *Qualitative Reasoning about Physical Systems*, North-Holland, Amsterdam, (1984).

[BKt82] H. G. BORUFKA, H. W. KUHLMANN and P. J. W. TEN HAGEN, "Dialogue Cells: A Method for Defining Interactions," *IEEE Computer Graphics and Applications*, 2(7), 1982, pp. 25-33.

[Cha78] C. L. CHANG, "DEDUCE 2: Further Investigations of Deduction in Relational Data Bases," in *Logic and Data Bases*, H. GALLAIRE and J. MINKER (eds.), Plenum Press, New York and London, (1978), pp. 201.

[ClM81] W. F. CLOCKSIN and C. S. MELLISH, *Programming in Prolog*, Springer, Berlin, Heidelberg, New York, (1981).

[For81] C. L. FORGY, "OPS5 User's Manual," Technical Report No. CMU, Carnegie-Mellon University, Pittsburgh, (1981).

[Ger85] J. S. GERO (ed.), *Knowledge Engineering in Computer-Aided Design, Proceedings of IFIP WG5.2 Working Conference in 1984 (Budapest)*, North-Holland, Amsterdam, (1985).

[GoR83] A. GOLDBERG and D. ROBSON, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, New York, (1983).

[KSH83] F. KIMURA, T. SATA and M. HOSAKA, "Integration of Design and Manufacturing Activities Based on Object Modelling," in *Advances in CAD/CAM*, T. M. R. ELLIS and O. I. SEMENKOV (eds.), North-Holland, Amsterdam, (1983), pp. 375.

[Mac86] K. J. MACCALLUM, "Knowledge-Based Systems for CAD," *Proceedings of CAPE '86, Second International Conference on Computer Applications in Production and Engineering*, Copenhagen, (1986), pp. 903.

[RHS85] D. R. REHAK, H. C. HOWARD and D. SRIRAM, "Architecture of an Integrated Knowledge Based Environment for Structural Engineering Applications," in *Knowledge Engineering in Computer-Aided Design, Proceedings of IFIP WG5.2 Working Conference in 1984 (Budapest)*, J. S. GERO (ed.), North-Holland, Amsterdam, (1985), pp. 89.

[Rod71] W. RODENACKER, *Methodisches Konstruieren*, Springer, Berlin, Heidelberg, New York, (1971).

[SaW81] T. SATA and E. WARMAN (eds.), *Man-Machine Communication in CAD/CAM: Proceedings of IFIP WG5.2/5.3 Working Conference in 1980 (Tokyo)*, North-Holland, Amsterdam, (1981).

[SrA86] D. SRIRAM and R. ADEY (eds.), *Applications of Artificial Intelligence in Engineering Problems, Proceedings of 1st International Conference (1986), Southampton University, UK*, Springer, Berlin, Heidelberg, New York, Tokyo, (1986).

[teD84] P. J. W. TEN HAGEN and J. DERKSEN, "Parallel input and feedback in dialogue cells," CWI Report No. CS-R8413, Centre for Mathematics and Computer Science, Amsterdam, (July 1984).

[ToY85] T. TOMIYAMA and H. YOSHIKAWA, "Requirements and Principles for Intelligent CAD Systems," in *Knowledge Engineering in Computer-Aided Design, Proceedings of IFIP W.G. 5.2 Working Conference 1984 (Budapest)*, J. S. GERO (ed.), North-Holland, Amsterdam, (1985), pp. 1-23.

[ToY87] T. TOMIYAMA and H. YOSHIKAWA, "Extended General Design Theory," in *Design Theory for CAD, Proceedings of the IFIP W.G. 5.2 Working Conference 1985 (Tokyo)*, H. YOSHIKAWA and E. A. WARMAN (eds.), North-Holland, Amsterdam, (1987), pp. 95.

[vat87] R. VAN LIERE and P. J. W. TEN HAGEN, "Introduction to Dialogue Cells," CWI Report No. CS-R8703, Centre for Mathematics and Computer Science, Amsterdam, (January 1987).

[Yos81] H. YOSHIKAWA, "General Design Theory and a CAD System," in *Man-Machine Communication in CAD/CAM: Proceedings of IFIP WG5.2/5.3 Working Conference in 1980 (Tokyo)*, T. SATA and E. WARMAN (eds.), North-Holland, Amsterdam, (1981), pp. 35.