Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

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

IDDL: A language for intelligent interactive integrated CAD systems

Computer Science/Department of Interactive Systems Report CS-R9056 October

IDDL: A Language for Intelligent Interactive Integrated CAD Systems

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

V. Akman
Department of Computer Engineering
Bilkent University
P.O.Box 8, 06572 Maltepe, Ankara, Turkey

In this paper, we present IDDL (Integrated Data Description Language), a language designed for Intelligent CAD systems. IDDL consists of two main parts. (1) it provides a database integrating object-oriented programming and logic programming paradigms for a design object description. These two separate parts of the database are connected through the means of functions. Representational constructs of the language are selected according to special needs of data representation in CAD. These include several non-classical logic features of the language. (2) IDDL provides rule-based scenarios for the design process description to manipulate the database in a forward reasoning manner by executing these scenarios. The scenarios provide also a mechanism to maintain multiple worlds of the design database. This reflects the evolutionary aspect of the design development. We present the application of these concepts to design process representation.

CR Categories and Subject Descriptors: D.3, F.4.1, J.6.
Key Words & Phrases: intelligent CAD, object-oriented programming, logic programming, forward reasoning, multiple worlds.

1. Introduction

The programming language IDDL (Integrated Data Description Language) described in this paper is aimed at specifying Computer Aided Design systems (CAD systems) [16]. Every human controlled production process for some artifact is containing numerous design tasks. Only very few of those are supported by computerised design tools. IDDL contains new constructs which supports writing CAD systems for a wider range of design tasks.

Designing is a very complex activity. Consequently, the motives for improving design support can be quite different, e.g. making designing more economical, improving the quality of designs, making designing more independent of expertise, etc. In a number of preliminary studies [13, 14] such general objectives have been analyzed in order to distill a number of problem areas for which adequate scientific and technological treatment would provide better support.

In this paper, the problems to be dealt with are classified in such a way that language constructs needed for formulating design support methods can be gradually introduced. This is

Report CS-R9056
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

achieved by making use of ideas originated from AI research [1]. Hence, with each new facility added, both language and application domain will grow. This will allow us to first attempt to solve some fundamental problems associated with what we understand about the essentials of a design process.

All CAD systems, written to support a design task, have three major components in common: a design object component, a design process component and a user interface component. An improvement of the design object representation is a necessity for more integration of design tasks: results can be shared among tasks. A better design process representation is necessary for enlarging the scope of design tasks: it allows for more subtle treatment of the design object, e.g. when the object definition is still incomplete. A better user interface is necessary to allow the user to directly specify in the domain specific terminology. This requires semantic processing of user transactions. The user interface must be directly coupled to the object and process representations.

In order to understand these three aspects of a CAD system we will, in the next chapter, elaborate the relation between design object and design process. Then a model for user interfacing will present itself. After this the most basic semantic features of IDDL, which directly reflect these relations, will be introduced.

2. Object and Design Process Representation

A design process representation is a list of actions which when executed in a meaningful order may specify a design object. Each of the actions is preceded by a condition which expresses when the action is possible. In that case execution will contribute something to the design object. Actions can be structured in design scenarios. A scenario exists for each design task. The status of the design process can be explicitly maintained in a facts-base. In a similar way, the status of the design object is maintained in the same facts-base. Both types of status information and the user inputs together form the operands of the conditions that precede the action rules. The conditions therefore represent the control structure of a scenario. The control structure of the entire design process is obtained by adding to this the scenario structures which consists of a sub-scenario hierarchy and the option to activate several scenarios concurrently.

The design object is represented by atomic objects and object relations, which can be used to create composite objects. Hence, all object structuring is obtained explicitly by asserting relations. These relations together make up the object status mentioned above. Equally, the design process status can be represented as object relations. This makes it possible to experimentally take a very close look at how a design object and its generating process are interrelated.

The part of a design object and the design process status which is relevant for a given scenario is called a world. Much of the potential of the system will depend on how flexible a world mechanism can be built. A world can create a particular, restricted view on the design object. In a world one can more clearly and efficiently, execute the actions of the corresponding scenario.

All automated parts of the design process now follow the same scheme: if a condition defined over process status and object status holds, then action is taken as specified in the consequence of the condition. This action may lead to a new status for both object and process, for which another rule may apply, and so on. This process may come to a halt. This is the point where new input from the user is needed to continue.

3. Design Object Representation in IDDL

The total knowledge base consists of the collection of all objects which are created and the collection of all relationships that hold among these objects. In addition, every object has its current status — e.g. values of its attributes, etc. The elementary object is just a constant name. Some objects, however, can have attributes. These are represented by functions and map objects onto attribute values. A set of such functions forms the inside structure of an object. The former is called a *primitive* object, the latter a *composite* object.

Due to the control structures of the language (to be described later when scenarios are introduced) one can identify a particular configuration of objects and their relationships. Such configurations are called *worlds.* Worlds as a configuration of objects and their relationships partition the knowledge base in a straightforward manner. A world is also regarded as a third kind of object. At any point in time a functional separation of a knowledge base (knowledge base partition) is to call the set of available objects an *objects-base* and the relationships among them a *facts-base.*

3.1. The objects-base

In their modelling task CAD systems need to establish structures. Objects have particular properties which make it possible to treat them as either manipulable entities (constants) or to extract information from them. Every object has (1) a *constant name* and (2) an *object type.* The former serves as a reference to the objects-base, the latter is a reference to the object's prototype definition. Such a prototype serves as a template to build up the object's structure in the objects-base. There exist two kinds of objects:1 viz. *primitive objects* and *composite objects.* The former have nothing but a name and a value whilst the latter have an internal structure, i.e. attributes, functions and constraints. In this sense, we may compare IDDL with Smalltalk-80.† IDDL functions are like 'methods' in Smalltalk and attributes are like instance variables [8, 9].

3.1.1. Primitive objects. Primitive objects are the building blocks of the IDDL objects-base. They have a *name,* a *type,* a *value,* and a set of *functions.* The following primitive object definitions are available: Integer, Real, Character, String, Symbol, List, and Collection. There are two ways to instantiate a primitive object. (1) by assigning a value to an uninstantiated variable, e.g. gets(X,8), X becomes a primitive object of type Integer with value 8, or gets(Y,'foo'), Y becomes a string with value 'foo' or (2) with the built-in predicate new. The first argument of new becomes a newly instantiated object, the second argument is a primitive object type, e.g. new(X,Integer).

1 The third kind of object, a world, is regarded as a different type not being part of the objects-base. It is dealt with in §4.3.

† Smalltalk-80 is a trademark of Xerox Corporation.

A primitive object's name represents its value, i.e. it is an evaluable term. For instance, suppose we have an object a of type Integer. We can test its value with the built-in predicate equals. A valid formula is thus equals(a,10), evaluating to either true or false. A primitive object recognises a set of functions belonging to the object's 'type'. A function is represented by a colon followed by the function name and optional argument(s) between square brackets, e.g. :name[anObject]. The object to which the function is 'sent' precedes the function. For the type Integer the available functions are: +, –, *, /, :abs, :negated, :gcd[]. (Note that for convenience the colon is omitted for the well-known arithmetic operators.) E.g. the function call 9 :gcd[12] answers the greatest common divisor of 9 and 12.

3.1.2. Composite objects. Objects which have more than a name and value own an internal structure. Such objects are called composite objects. The structure of such objects depends on the specific *prototype* to which its type refers. The structure of such a prototype is made of:

- *attributes;* These denote a certain property of the object. An attribute itself is a primitive object, therefore, having a name, type and value. An attribute value can be accessed by a 0-ary function with the same name as the attribute but preceded by a colon. Attributes denote the internal structure of objects (e.g. an object tableLeg has attributes length, diameter, and material, accessed by :length, :diameter, and :material).
- *functions;* These define a property of the object just like an attribute except that it cannot or it is not useful to represent that property as a simple value. A function has a function body which contains some procedural code which can be executed. This body may have function calls to other objects which are part of the entire structure of the design artifact.
- *constraints;* These describe consistency criteria of the given object. They consist of clauses. In principle, constraints define internal relationships between attributes and functions. These constraints are so called watch dogs to prevent attributes and functions from receiving a faulty value. Constraints have the same syntax as clauses described in the next section.

Composite objects are instantiated by asserting a 1-ary predicate to a facts-base. The predicate symbol refers to an object type, its ground term denotes an object name, e.g. table(aTable). When this predicate is actually asserted, the prototype for table (see Fig. 1) is looked up in the objects-base and a copy of it is made to be used further during the course of the design process. Upon instantiation attributes may have a default value which is used until the 'real' value has been determined. In the example the attribute numberOfLegs has the default value 4.

When somewhere in the prototype definition the keyword *self* is used, it refers to the object itself. Hence, the term self :length is evaluated by calling the object's own function :length. However, there may be references to other objects as well. In the facts-base an object hierarchy is defined, describing the decomposition of the design artifact. This hierarchy is described by a built-in predicate hasPart. It is a 2-ary predicate symbol, the second argument denotes an object which is a sub-part of the first, e.g. hasPart(aTable,aTop). A hasPart predicate defines the links between objects in the objects-base. Consequently, in an object prototype definition there may be a reference to another object in a function. This object is supposed to be a sub-part of the prototype defined by an object hierarchy. In Fig. 1 such a reference is made by top :length. When the object hierarchy is not (yet) defined a function call in which such a reference appears returns nil.

```
object table
attributes
    numberOfLegs = Integer (4)
functions
    :height = { leg :length + top :thickness }
    :length = { top :length }
    :width = { top :width }
    :size = { self :length * self :width}
constraints
    greater(numberOfLegs,2) &
    greater(self :length,self :height) &
     X,Y ( leg(X) & hasPart(self,X) &
           leg(Y) & hasPart(self,Y) &
           equals(X :length,Y :length) )
end
```

Fig. 1. A prototype for tables

3.2. The facts-base

The facts-base is used for representing relationships between objects. The facts-base is built of definite program clauses and unit clauses [3, 10]. If we query the facts-base, the answer is either directly available through a unit clause, or it is derived from a (series of) program clause(s). To define definite program clauses we first introduce an alphabet, terms, predicates and literals. The alphabet consists of six classes of symbols:

1. *variables,* symbols beginning with an upper case letter (e.g. X, Y, AContainer).
2. *constants,* symbols beginning with a lower case letter (e.g. a, b, aContainer). Number constants such as integer, real etc., as well as string constants are defined as usual (e.g. 123, 0, –456, –3.45E3, 'aString', 'tyuhaj').
3. *function symbols* beginning with a colon followed by a lower case letter (e.g. :f, :h, :volume). We use the skolem function, '_', to denote the unknown[2]. Some function symbols are predefined.
4. *predicate symbols* beginning with a lower case letter (e.g. p, q, container). Some predicate symbols are predefined. These include the unary predicate symbols: ![3] (cut), fail, true, false, etc. and the n-ary predicates: use, equals, gets, etc.

[2] Note that '_' is equivalent to Prolog's 'don't care' symbol.

[3] Cut is a non-logical annotation to convey a certain control facility, which can be applied in systems using a leftmost search algorithm. Although it is used in the position of a predicate symbol, it is not a predicate and it has no logical significance at all. However, it is convenient to regard it as a predicate which succeeds immediately.

5. *connectives.* They are limited to ←, & and ~. They are understood in the standard logic programming sense[4].
6. *punctuation symbols,* ‘(’, ‘)’, ‘[’, ‘]’, ‘,’ and ‘.’

Over this alphabet we can define terms as follows:

1. A variable is a term.
2. A constant is a term.
3. If t is a term, then (t) is a term.
4. If :h is a 0-ary function and t_0 is a term, then t_0 :h is a term.
5. If :h is an n-ary function (n>0) and $t_0, t_1, \ldots, t_n$ are terms, then t_0 :h$[t_1, \ldots, t_n]$ is a term.

A *ground term* is a term not containing variables. If p is an n-ary predicate symbol (arity may be zero) and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a *predicate.* A *literal* is a predicate or the negation of a predicate. The facts-base is built up from definite program clauses. A definite program clause is a clause of the form

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

A is called the head of a clause and $B_1 \ \& \ \cdots \ \& \ B_n$ is the body. A is a predicate and B_1, ..., B_n are literals. A clause with an empty body is called a unit clause, viz. A ←. An example of a facts-base with some definite program clauses is given in Fig. 2.

```
material(concrete) ←
material(steel) ←
material(wood) ←
colour(X) ← colours(Cs) &
            element(X, Cs).
colours([white, black, green, red, brown, yellow, blue]) ←
element(E, [␣] :cons[E]) ←
element(E, Tl :cons[Hd]) ← ~equals(E, Hd) &
                           element(E, Tl).
hasColour(concrete, white) ←
hasColour(steel, black) ←
hasColour(wood, brown) ←
```

Fig. 2. A facts-base concerning materials and colours

In this example, we define three kinds of predicates: material, colour and colours. Furthermore, we define the relationships element and hasColour. (Note that equals is a built-in predicate; :cons is the built-in list constructor function.) With this example we show two ways to specify an object type. One way is to enumerate all individual objects explicitly (like material) using unit clauses, or another way is to specify them implicitly (like colour) using definite program clauses. The scope of variables is restricted to the clauses in which they appear. All

[4] A different set of connectives is used for query formulae and another for assertive formulae.

variables in a clause are universally quantified over that clause.

The *facts-base* contains *clauses* but differs from a standard Prolog database. There are **no** built-in predicates in this part of IDDL, which could modify the facts-base (e.g. assert, retract) while proving a goal. Assertions and retractions are always performed from outside by a separate interpreter called *supervisor* which executes *scenarios* (see §4.1). Consequently, the only purpose of the facts-base is to store relationships and to answer questions about these relationships. Together with the *function definitions* a facts-base behaves like a database — thus can be queried by the query processor (QP). The QP can be accessed from:

- The if-part of scenario rules (see §4.2).
- Clauses in the facts-base or from adjacent function definitions.

There are built-in predicates (and built-in functions as well) to issue queries. The single query p(t) finds one set of ground terms for the free variables in t for which p(t) can be proven.

When a query is addressed to the facts-base it can either give the answer immediately (since the atom is there as a unit clause) or it can derive it via definite program clauses. When the query contains uninstantiated variables, a unification algorithm is used to find the instantiation pair list. By definition, a list of the form [t, S] is called an *instantiation pair list,* where elements of S are ground instances of t.

Example: if we query material(concrete) to the facts-base described above, the answer will be 'true'. If we ask material(M), the answer will be 'true' and M will be instantiated to one of [concrete, steel, wood]. If we ask material(M) & (hasColour(M, white) | hasColour(M, black)), the instantiation pair list will be one of [M, steel], or [M, concrete]. In case of the query colour(green) the answer is derived rather than found directly. Note that the way an answer has been obtained is not visible to the issuer of the query.

We can say that the facts-base is goal-oriented. In other words when we pose a query to a facts-base, it tries to match this goal with the head of a definite program clause. The predicates of the body of this clause are considered to be the new (sub)goals. This procedure is continued until all (sub)goals are matched with unit clauses. This kind of top-down search strategy is commonly called backward chaining. When an answer to a query is already an element of the instantiation pair list of the poser of the query another binding is collected through backtracking. This will require some careful design to avoid pitfalls of inefficiency [11] and unfairness or falling into infinite loops. To this end the search in the SLD tree will have to divert from a purely depth first mode (see [10] page 59).

The result of the queries is used by *scenarios*. Special functions can be defined for instance for initialisation of objects to build their inside at instantiation time. Technically this amounts to adding values to the object's attributes. These values can be default values which can be altered or become certain later.

The entire mechanism is controlled in scenarios. Scenarios can also manipulate objects in other ways, e.g. by assigning values to object attributes. By this functionality a scenario may change the inside of an object.

3.3. Evaluating the terms

The main issue is how to mold the unification mechanism of the logic programming part with the binding process which takes place when functions are evaluated. In the literature several approaches are proposed to this purpose. (Several modified forms of unification allow defined functions into logic programming — see [5] for an extensive study of the question.) Basically, we would like to use SLD resolution techniques but add a co-processor which evaluates functions (the evaluator). Since SLD resolution amounts to subsequent application of substitution and unification processes, the problem is really how to unify with evaluable terms.

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

- should the subterms be evaluated first and then passed to the body of the function, or
- should the subterms be substituted without evaluation into the body of the function definition and evaluation only take place when the need arises?

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

It is disputable whether a resulting term may contain functions with unbound variables. In case we admit this, the evaluator must allow symbolic computation or the evaluation must be suspended in the given goal and using and-parallelism the next subgoal should be attempted. If a function as a subterm can not be evaluated to a non-evaluable term, and there is a chance to prove the corresponding subgoal in which it appeared, we can temporarily assume that the subgoal is true and proceed. Later subgoals may bind such variables and thus the strict left-to-right evaluation of subgoals is not enforced any more.

We expect that not allowing evaluable terms into the heads of function definitions will simplify the problems involved, because we have made the evaluation process one-way.

4. Design Process Representation in IDDL

IDDL is basically a very simple language: it consists of a facts-base and a set of scenarios. The facts-base contains definite program clauses; they denote the facts that are currently known about a design artifact. A scenario contains IF-THEN rules, and upon activation, performs (forward) reasoning on those facts. Rules denote the somewhat invariant knowledge about designing; i.e. how designing takes place [15]. Clauses express the more variant knowledge about the design objects themselves. Consequently, rules will not change during the design process, whilst the number of clauses will grow significantly.

We may conclude that functionally IDDL acts simultaneously as a deductive database language concerning the facts-base and as a modular production system with regard to scenarios. Hence, the two parts of IDDL have their own syntax and interpreter called the supervisor. In the next sections we describe the scenarios, rules, worlds, and rule selection control.

4.1. Scenarios

A scenario is a procedure-like structure with a world declaration part as argument, a function definition part and a body with rules (see Fig. 3).

```
scenario name(World_1, ..., World_n)
functions
   :f[] = { function body } ;
   .
   .
   :h[] = { function body } ;
begin
   if condition then action ;
   .
   .
   if condition then action ;
end
```

Fig. 3. Scenario structure

The set of objects, which are declared in a scenario, and their relationships embody a *world.* A world is a partition of the facts-base and it consists of objects and the clauses which were asserted in connection with these objects. A world denotes the current state of the design object description (see §4.3).

The function definition part defines a function which can be used locally during the execution of the scenario (see example in Fig. 4).

```
:dist[P] = { Dx := self :x – P :x ;
             Dy := self :y – P :y ;
             (Dx*Dx + Dy*Dy) :sqrt } ;
```

Fig. 4. Example of function definition

The keyword self refers to the object to which the function is sent. The functions :x and :y in the example are defined in the objects self and P returning a 'real' object. The variables Dx and Dy are local variables and the value of the last statement becomes the value of the function. The functions denote the more procedural knowledge whilst the rules denote the more declarative knowledge. The object declarations and the function definitions are optional.

The rules are so called 'IF-THEN' rules. They are similar to those found in production rule systems [4]. Rules have no logical significance at all. They are used to encode the design process, and they therefore denote the design process knowledge.

4.2. Rules

In this section, we describe IDDL rules in more detail. A rule consists of two components, a left and right hand side. The left hand side (LHS) is evaluated with reference to the facts-base and if this succeeds, the action specified on the right hand side (RHS) is executed.

4.2.1. Syntax of rules. The LHS and RHS of a rule are both called formulae although they have a somewhat different syntax. A LHS-formula is defined as follows:

1. A literal (a predicate or the negation of a predicate) is a LHS-formula.

2a. If F and G are LHS-formulae, then so are F & G and F | G.

The definition of a RHS-formula is analogous. We have 1. defining an (atomic) RHS-formula and:

2b. If F and G are RHS-formulae, then so are F & G, F | G, F ∧ G, and F ∨ G.

A rule is defined as follows: **if** <LHS-formula> **then** <RHS-formula> ;

The connectives ∧ and ∨ are understood in the standard logical sense. The connectives & and | have respectively the same meaning except that the operands of the former are evaluated in parallel whilst those of the latter are evaluated sequentially. This parallelism is explained below.

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

4.2.2. Evaluation of the LHS. For the evaluation of a LHS formula we make a distinction between predicates which are built-in predicates and predicates which can be found as predicates in the facts-base (FB). The former can be categorised as operators for comparing objects, user interface calls, control predicates and other predicates which are system routines, the latter can be seen as queries to the FB and they are evaluated to be true if they are successfully unified with the FB.

A rule is fired if the condition in the LHS is fulfilled; i.e. the built-in predicate or the unification algorithm succeeds. The rule tries to perform the action at the RHS using the variables instantiated in the LHS. Below we give some examples of such rules.

```
if p(a) & q(a,X) then action ;
if p(X) | q(X) then action ;
if askUI(length,L) & greater(L,10) then action ;
```

The rule in the first example reads: if both p(a) and q(a,X) are derived successfully from the FB with X instantiated to some value, then the action at the RHS is performed. In the second rule the second operand, q(X), is only queried in case the first, p(X), fails. The action is performed if either of them succeeds.

The third rule shows the use of built-in predicates. The built-in predicate askUI starts a dialogue with the user asking for a certain length. It returns the value in L. The action is then performed if this value is greater than ten.

4.2.3. Evaluation of the RHS. Similar to what we have shown above, the predicates on the RHS are either built-in predicates or predicates which are asserted to the scenario's world. The former are assignments, scenario calls, modifications of the knowledge-base (assert, retract or change) or reports to the user interface. The latter are predicates which stay within the scenario's world as long as the scenario is active. These assertions are only temporary and local, i.e. they do not affect the parent scenario's world after termination.

On the RHS of a rule basically only two logical connectives are used: *and* and *or*. However, for both connectives there exists a sequential (& and |) and a parallel (∧ and ∨) version. The operands of the former are executed sequential, whilst those of the latter are executed in parallel. The parallel connectives are used to invoke the multi-world mechanism. This mechanism is described in detail in the next section.

A rule is fired if the LHS is successfully unified with the facts-base. The rule tries to perform the action at the RHS, which is either an assertion to or a retraction from the scenario's world or it is the invocation of an other scenario. A scenario is invoked by a built-in predicate use. Its first argument is a scenario name and the remaining arguments are names of the worlds which are associated with that particular scenario. This mechanism creates a new sub-world on top of the former, the parent world, and the invoked scenario becomes active. We call a scenario active if control is passed to it. An example is given below:

```
if p(a) & p(b) then use(sc1, w) ;
```

This rule reads: if both p(a) and q(b) are derived successfully from the facts-base, then use the scenario called 'sc1' with world w associated with it.

During the execution of a scenario the rules are fired in some order which is determined by a *rule selection method.* This mechanism is described in §3.4. This process is continued until the scenario has either succeeded or failed. A rule succeeds if an assertion or an invoked scenario succeeds. There are two ways to make a scenario succeed. The first possibility is when a *noMoreRule* situation arises. If there are no rules which can be applied, control is given back to the parent scenario and the child scenario succeeds. The second possibility is via the built-in predicate success. When this is encountered control is given back to the parent scenario. A built-in predicate, fail, makes a scenario terminate unsuccessfully.

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

4.3. Worlds

A world is a partition of the facts-base and it therefore consists of clauses. These clauses denote object definitions, relationships between these objects, and also other world definitions which are sub-worlds. A world itself is yet another kind of object. An example of a world is given in Fig. 5. A world is given as an argument to a scenario call and it is associated with that scenario. A world may be augmented through the rules during the execution of the scenario. This either happens when an object's inside is changed or when new object declarations are asserted or when new relations are asserted. When the scenario succeeds, the (modified) world is returned to the parent world which is associated with the parent scenario. If the scenario fails, the parent world will not be affected. We distinguish between local and global assertions. The former will not appear in the parent scenario's world, while the latter will.

```
world(table)
begin
   table(aTable) ←
   hasPart(aTable,aLeg1) ←
   hasPart(aTable,aLeg2) ←
   hasPart(aTable,aLeg3) ←
   hasPart(aTable,aLeg4) ←
   hasPart(table1,aTop) ←
   world(legs) ←
   world(top) ←
end
```

Fig. 5. Example of the world table

In the example, the worlds legs and top may be opened by the built-in predicate open. In that case, the clauses which appear in those worlds will become accessible to the scenario. A world given as an argument to a scenario call is automatically opened with the call.

4.4. Multiple worlds

Using parallel connectives in the assertive formulae we can invoke two or more scenarios at the same time and thus we can create multiple worlds. In creating multiple worlds we make a distinction between *and-* and *or-parallelism* of the forward reasoning process. The and-mechanism is used to create sub-worlds that depend on each other. They use the same design object description and they serve to model it in various ways. The or-mechanism creates sub-worlds that are independent of each other. The sub-worlds denote distinct design object descriptions possibly generating different design solutions.

The and-mechanism is invoked by using the operator: $\wedge$. So, with the statement: use(scenario1, w_1) $\wedge$ use(scenario2, w_2), two scenarios are activated each with its own world attached to it. The former contains w_1, the latter w_2. Each scenario acts upon its own world, and there is no direct dependency between them. Both scenarios may augment their worlds, i.e. they may add new facts to their worlds, or they may assign values to *unknown, believed,* or *default* facts. Known, unknown, believed, and default facts respectively express facts that are fixed for the remaining time of the design process, facts that are not yet determined at this time of the design process, facts that are true in a certain world but not certain for the whole universe, and facts that are assumed to be true, but may change during the design process. In the following two sub-sections we elaborate on the and- and or-mechanism.

4.4.1. The and-mechanism. Let us consider the following rule:

if <condition> **then** use(sc1, w_1, w_2) $\wedge$ use(sc2, w_1, w_3) ;

The two scenarios sc1and sc2 are activated. (The names of currently active scenarios are not necessarily unique.) The rule might have invoked the same scenario twice with different worlds. Two sub-worlds are created, the former containing the worlds: w_1 and w_2 the latter containing: w_1 and w_3. Each scenario acts upon its own world.

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

4.4.2. The or-mechanism. The or-mechanism creates independent worlds. Suppose we have the following rule:

if <condition> **then** use(sc1, w_1, w_2) $\vee$ use(sc2, w_1, w_3) ;

This rule also activates two scenarios. These scenarios are independent and the rule, therefore, only fails when both scenarios fail. If one scenario succeeds and the other fails, only the succeeding scenario affects the parent world. However, if both scenarios succeed two copies of the parent world are generated, one with the results of sc1's world included and the other with those of sc2's world. This separation continues as long as the parallel worlds differentiate.

It is clear that this multi-world mechanism may be applied to more than two cases. It then behaves in a similar way. Note that $\vee$ has higher precedence than $\wedge$. Suppose we have the rule:

if condition **then** use(q, w) $\wedge$ use(r, w_1) $\vee$ use(r, w_2) ;

The rule succeeds if the scenario q(w) is successfully called together with the successful call of r(w_1) or r(w_2) or both. In the latter case, once again two copies of the parent world are made, one including the contents of both w and w_1, and the other that of both w and w_2.

The control of the and- and or-mechanism is taken care of by the *supervisor.* The supervisor controls all the facts-base traffic and determines when and how to generate new copies of the facts-base. It may discard, in interaction with the user, certain worlds when they do not look promising or they may be suspended for some time. This suspension means that the worlds are maintained and that they may become active after a while when the scenarios which were processed first were not successful.

The copying of worlds does not mean physical copying: instead we would like to use techniques similar to assumption based truth maintenance [6], where opening a world is an assumption with which subsequent assertions in that world are labeled.

4.5. Rule selection control

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

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

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

Another mechanism is that all applicable rules are collected and from those the most appropriate rule is selected, satisfying some criteria. Several criteria have been suggested by Davis and King [4] and de Kleer [7]:

- *Data order*. Objects in the facts-base are ordered, and the rule that matches object(s) in the facts-base with highest priority is selected.
- *Generality order*. The most specific rule is chosen. A rule is more specific if it has more conditions to be satisfied in its LHS.
- *Recency order*. This means choosing either the most recently executed rule (of course with a different instantiation-pair list), or the rule referring the most recently created (modified) object(s) in the world.
- *User's decision*. The user determines what kind of design should be performed. The rule which suits best to this decision is chosen. This rule selection method is different from the other methods in the sense that it examines the action part of a rule rather then the condition.
- *Meta rules*. These are rules about rules and may embody noncontingent heuristic strategies for selecting rules.

The rule-control mechanism used in IDDL is a combination of all these. It is possible to select a new mechanism dynamically at run-time. To achieve this we introduce a construct in IDDL called directive, which is yet another built-in predicate. When directive(aRuleSelMeth) is encountered, the rule selection method aRuleSelMeth is picked by the interpreter. This method will then be used until another directive is encountered. The above set of methods may be extended at any time by the user, thereby providing all the possible freedom to influence the behaviour of the system. When a new rule selection method is introduced, the only thing to be done is to add enough ''logic'' to the interpreter so that it can execute it.

Another feature of IDDL is the possibility to group rules together in so-called blocks. A block starts with a curly bracket **{** and ends with another **}** (thus it is a list of rules). Blocks are evaluated sequentially in a circular fashion. The rules inside a block are evaluated using the current rule selection method. A built-in predicate break is used to leave a block and continue with the next block. This mechanism can easily be used for implementing dynamic constraints [7]. (as opposed to static constraints in object definitions). Moreover, when a noMoreRule situation occurs inside a block, that block is also exited and the next block's evaluation is started. In Fig. 5 we give an example of blocks and the use of the built-in predicates directive and break.

```
scenario name(...)
begin
{  directive(dataOrder)
   if <> then <> ;
   .
   .
   if <> then directive(generalityOrder) & break ;
}
{
   if <> then <> ;
   if <> then directive(askUser) ;
   .
   .
   if <> then fail ;
}
end
```

Fig. 5. Example of rule selection methods and blocks

5. The User Interface

The user interface follows the border line between the information in the computer, which is the formalised part and the rest of the information which remains outside the system in the user's mind. For instance, for some design activities only the final result will be represented as a user decision.

The user interface provides access to the information represented internally. In the scenario-world-scheme it is possible to represent the user input events as instantiated objects existing in the world. They can then be treated just like any other object. Output can be provided through object functions. Object functions can be part of predicates in the facts-base. Hence, a query may yield a reply essentially containing the user input.

The object interface to the user makes it possible to define an object configuration for the user interface on a per scenario basis. The user interface object configuration simply is a subset of the scenario's world definition which contains the entire object configuration. Hence, for the first initial phase of IDDL no extra construct need to be introduced to serve the user interface other than exchanging object instantiations.

6. Conclusions

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

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

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

The dynamic part of IDDL is mainly concerned with two problems. One is, how can be provided an appropriate control for the forward reasoning process used in scenarios. As many have argued one of the impediments in expressing design knowledge in any computer language is the monocultural nature of the languages available [2]. Our approach is to provide the user of the language with metalevel control facilities. This pluralistic, permissive approach allows control strategies tailored to the problem and is achieved by a set of control primitives to influence the rule selection method of the forward reasoning process. Also pure procedural programming is possible in the functional part of the language while backward chaining algorithms can be best described using IDDL's subset for the facts-base.

The already well-known requirement of maintaining multiple worlds influenced the IDDL design into defining specific constructs for starting parallel worlds. As in assumption based truth maintenance [6], default logic [12] is only one of the reasons why assumptions are created. It is the problem solvers duty to judge whether and when to create an assumption — on account of the use of a default fact in the reasoning process (i.e. at the left hand side of a rule) or because of an explicit parallel world branch.

Acknowledgements

We are all thankful to T. Tomiyama who started the research and to our visiting researchers F. Arbab and Zs. Ruttkay who contributed with their criticisms to the results presented here.

References

1. V. Akman, P.J.W. ten Hagen, J. Rogier, and P.J. Veerkamp, ''Knowledge engineering in design,'' *Knowledge-Based Systems*, vol. 1, no. 2, pp. 67-77, 1988.
2. T. Bylander and B. Chandrasekaran, ''Generic Tasks for Knowledge-Based Reasoning: the Right Level of Abstraction for Knowledge Acquisition,'' *Int. J. of Man-Machine Studies*, no. 26, pp. 231-244, 1987.

3. W.F. Clocksin and C. S. Mellish, *Programming in Prolog,* Springer-Verlag, Berlin, 1981.
4. R. Davis and J. King, ''An Overview of Production Systems,'' in *Machine Intelligence*, ed. E.W. Elcock and Donald Michie, pp. 300-332, Ellis Horwood Ltd., Chichester, 1977.
5. D. De Groot and G. Lindstrom, *Logic Programming — Functions, Relations, and Equations,* Prentice Hall, Englewood Cliffs, N.J., 1986.
6. J. De Kleer, ''An Assumption Based TMS,'' *Artificial Intelligence*, vol. 28, pp. 127-162, 1986.
7. J. De Kleer, ''Problem Solving with the ATMS,'' *Artificial Intelligence*, vol. 28, pp. 197-224, 1986.
8. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its implementation,* Addison-Wesley, Reading, MA, 1983.
9. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley, Reading, Mass., 1984.
10. J.W. Lloyd, *Foundations of Logic Programming,* Second, Extended Edition. Springer-Verlag, Berlin, 1987.
11. M.S. Paterson and M.N. Wegman, ''Linear Unification,'' *Journal of Computer and System Sciences*, vol. 16, no. 2, pp. 158-167, 1978.
12. R. Reiter, ''A Logic for Default Reasoning,'' *Artificial Intelligence*, vol. 13, pp. 81-132, 1980.
13. T. Tomiyama and P.J.W. ten Hagen, ''Organization of Design Knowledge in an Intelligent CAD Environment,'' CWI Report CS-R8720, Amsterdam, April 1987.
14. T. Tomiyama and P.J.W. ten Hagen, ''Representing Knowledge in Two Distinct Descriptions: Extensional vs. Intensional,'' Report CS-R8728, Centre for Mathematics and Computer Science, Amsterdam, 1987.
15. P.J. Veerkamp, ''Multiple Worlds in an Intelligent CAD system,'' in *Intelligent CAD: Record of IFIP 5.2 Workshop on Intelligent CAD*, ed. H. Yoshikawa and D. Gossard, North Holland, Amsterdam, 1988. to appear
16. Bart Veth, ''An Integrated Data Description Language for Coding Design Knowledge,'' in *Intelligent CAD Systems 1 — Theoretical and Methodological Aspects*, ed. P.J.W ten Hagen and T. Tomiyama, pp. 295-313, Springer-Verlag, Berlin, 1987.