Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

T. Kiriyama, D. Xue, T. Tomiyama, P.J. Veerkamp

Representation and implementation of design knowledge for intelligent CAD
Implementational aspects

Computer Science/Department of Interactive Systems Report CS-R9071 December

---

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 Research (N.W.O.).

Representation and Implementation of Design Knowledge for Intelligent CAD
Implementational Aspects

Deyi Xue, Takashi Kiriyama, and Tetsuo Tomiyama
Department of Precision Machinery Engineering
Faculty of Engineering, The University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan

Paul Veerkamp
Department of Interactive Systems
Centre for Mathematics and Computer Science
P.O.Box 4079 1009 AB Amsterdam, The Netherlands

Abstract: In this paper we present the metamodel mechanism, one of the features of IDDL (Integrated Data Description Language). IDDL is a programming language with features of object-oriented programming and logic programming and it is currently being developed at the University of Tokyo and CWI (The CWI version is called ADDL, Artifact and Design Description Language). It is a knowledge representation language especially designed for implementing intelligent CAD (Computer Aided Design) systems. It incorporates a metamodel mechanism, which provides a means to represent the design object model in a context independent way. From a metamodel aspect-models can be derived. Aspect models are interpretations of the metamodel in a certain context. Examples of an aspect model are a kinematic model and a geometric model. The paper is organized as follows, first we give an informal introduction to IDDL, thereafter we give an extensive example of the usage of the metamodel. Note that this paper is a follow-up of another paper in this volume, which provides a theoretical introduction to the subject [11].

CR Categories and Subject Descriptors: D.3, F.4.1, J.6.
Key Words & Phrases: intelligent CAD, knowledge representation, metamodel, aspect-model, logic programming, object-oriented programming

1. Introduction

Although CAD (Computer Aided Design) systems have become an essential tool for designers in various disciplines, it is also recognized that they are still inflexible and task dependent. The purpose of a CAD system is to support a designer in performing the design task. Certain routine tasks are delegated to the system. However, the majority of existing CAD systems are merely sophisticated workbenches for engineering drawing. As the application domain becomes more complex, designing becomes unmanageable with only this type of support. Therefore, we need a more sophisticated system which can assist a designer in an *intelligent* way, hence ICAD (Intelligent Computer Aided Design). Furthermore, to obtain a good system it must be highly *interactive* using the best human computer interaction techniques.

Existing programming languages do not have the properties which ICAD systems require, an ICAD system makes high demands on the programming language that is used for its

Report CS-R9071
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

implementation. Such a language must have the following features:

- It must provide a flexible design object description, allowing for incomplete and temporarily inconsistent descriptions. Incomplete in this context means that certain attributes and parts of the design objects are not yet determined. Inconsistent means that certain parts of the design object contain information which is in contradiction with other parts. This is only temporary since the final design object description needs to be complete and consistent.
- It must allow for design knowledge representation, both the design object knowledge and the design process knowledge. Design object knowledge is denoted by functions on and relationships among several parts of the design object. Design process knowledge is represented by scenarios and they describe how to create a design object and how to model it in order to obtain a complete design object description.
- It must provide a mechanism to integrate several sub-modules into the main system. These are used for the evaluation of the central design object description.

We believe that a language, which is based on both objects and logic, forms a firm basis for implementing an ICAD system. In the next section of the paper we give an informal introduction to IDDL.[1] In §3 we show how a framework for creative design can be established by means of a solution independent metamodel. Here we give an example of the application of the metamodel mechanism in IDDL.

2. The Metamodel Mechanism in IDDL

Current CAD systems focus their attention on the representation of a design object. However as discussed in [11], the representation of the design process plays an important role in intelligent CAD systems. Thus, a language which can describe both design objects and design processes is needed. IDDL (Integrated Data Description Language) is a language especially developed for design knowledge representation. It is to be used as the kernel language of the IIICAD (Intelligent Integrated Interactive CAD) system being developed at CWI (Centre for Mathematics and Computer Science) in Amsterdam [7].

The ideas of IDDL were initially presented in [13] by the IIICAD project group at CWI. Currently IDDL is developed at both CWI (called ADDL, Artifact and Design Description Language) [10, 12] and the University of Tokyo (for convenience we will refer to the language by IDDL) [9]. Although these two versions of IDDL are different in implementational details, they originate from the same basic specifications and philosophical ideas. In the next section, we give a brief description of IDDL. Since, the example in §3 is implemented in ADDL, the syntax used in this paper is that of ADDL. In §2.2 we present the metamodel mechanism in IDDL.

[1] In the previous Eurographics Workshops there has been an extended discussion on IDDL [9, 10, 12, 13]. We, therefore, think that yet another elaborated introduction is overdone.

2.1. Introduction to IDDL

IDDL is a programming language with features of both object-oriented and logic programming languages. It is basically a very simple language: it consists of *objects, facts,* and a set of *scenarios*. Like other object-oriented programming languages, IDDL has a single universal data-structure, an *object*. Each object has an internal structure, *attributes, functions* and *references* to other objects. The objects are stored in the *objectbase*. Facts describe relationships among objects, and properties of an object. Facts are definite program clauses, they are stored in the *factbase*. The objectbase and the factbase together denote the facts that are currently known about the artifact, i.e. the design object knowledge.

Scenarios contain IF-THEN rules and functions. The rules perform the (forward) reasoning about the facts, and the functions perform calculations on the objects. They denote the somewhat invariant knowledge about the designing; i.e. the design process knowledge. The facts express the more variant knowledge about the design objects themselves. Consequently, the rules will not change during the design process, while the number of facts will grow significantly and the inside of the objects will change drastically.

Resulting from this separation, we can conclude that IDDL acts as deductive database language concerning the factbase, it behaves like an object-oriented programming language in respect with the objects, and it acts as a modular production system in case of the scenarios. In the following sections we will describe the factbase, the objectbase, and scenarios.

2.1.1. The factbase. The factbase is used for defining properties of objects and relationships among objects. It is a deductive database whose language is built up from definite program clauses [1, 5]. So far, only unit clauses are available in IDDL. A unit clause consists of a predicate symbol, and a list of terms. Only constant terms are allowed, the are names of objects. For convenience we call a unit clause a *predicate*.

The factbase differs from a standard Prolog database. Clauses are not allowed to have side effects. Therefore, assertions and retractions are always performed from outside (i.e. via the scenarios). Consequently, the only purpose of the factbase is to store relations among objects, and to answer questions about these objects. A query may be posed to the factbase from the scenarios by asking for a predicate. The factbase gives the answer by matching the predicate symbol and the list of the predicate's terms. When the queried predicate contains uninstantiated variables, a unification algorithm is used to find the instantiation-pair list (i.e. the variables are mapped to all possible constants).

2.1.2. The objectbase. The objectbase consists of all instances of objects. Each object has a unique *name* and a *type,* and it has an internal structure consisting of attributes, functions, and object references. They can be compared with Minsky's frames [6]. Attributes contain the data of an object, functions describe the operations which can be performed on an object, and object references are the names of objects which are a a part of an object. E.g. an object of type slot and with the name aSlot has the attributes length, and width, it has the function :surface, and it has the object references face1, face2, face3, and face4. The object names are used as references to the factbase, where they are found as constant terms.

To generate new instances IDDL does not have the *Class* construct like in Smalltalk-80 [3],† instead it uses *prototypes* [4]. Prototypes serve as templates for creating objects, they define the type of a new object. In order to generate a new object, an object definition predicate is asserted to the factbase. It is a one-placed predicate, whose predicate symbol denotes the type of the object, and whose predicate term gives its name, e.g. by asserting slot(aSlot) we instantiate a new object of type slot with the name aSlot. IDDL has a prototype definition for each type providing the internal state of a new object. The result of an instantiation is that a new object is asserted to the objectbase together with the attributes and functions that belong to the prototype. The object's name is used as a reference from the factbase, where it is found as a constant term.

Initially there are no object references in an object definition. They are added to the object's internal structure when a hasPart predicate is asserted. E.g. in case of the assertion of hasPart(aSlot, face1), face1 is added to aSlot's objects list. In this way we define the part-of hierarchy. A prototype may give default values to certain attributes of an objects, which may be changed during the design process. The attributes and functions of an object are accessed from a scenario by means of function calls to the object.

2.1.3. Scenarios. A scenario is a procedure-like structure with a *function* definition part (optional) and a body with *rules*. The function definition part contains the definition of functions which can be used locally during the execution of the scenario. A function is defined by its name and a function body with a list of statements and a return value:

:name[*argument-list*] = { *statements ; return-value* }

A function is applied to an object by a function call: anObject :functionName[list–of–arguments]. The list of arguments is optional.

The rule part of a scenario consists of so called IF-THEN rules as they are found in production rule systems [2], they have no logical significance at all. They are used to encode the design process, and they therefore denote the design knowledge. Rules consist of two components, a *condition* part and an *action* part. The condition part is evaluated with reference to the factbase. The action specified by the action part is executed, if the condition succeeds. Both the condition and action parts include a list of predicates connected with & or | (logical-and and -or). The predicates may be preceded by a negator, ~, or a modal operator, % and #D (representing an unknown fact and a default fact respectively). The % operator is not allowed in the action part. The logical connectives are used in the standard logical programming sense, the predicates are evaluated from left to right consecutively. The terms of a predicate are either constants (starting with a lower case letter), variables (starting with a capital letter), or function calls. An example of an scenario is depicted in Fig. 1.

For the evaluation of a condition we make a distinction between predicates which are built-in predicates and predicates which can be found as predicates in the factbase. The former can be categorized as operators for comparing objects, user interface calls, control predicates and other predicates which are system routines, the latter are queries to the factbase and they are

† Smalltalk-80 is a registered trademark of Xerox Corporation.

```
refineGuide
    "compare the user-defined and kinematic stroke and adjust it accordingly"
FUNCTIONS
  "position"  :adjustX[ Dir, F ] = { self :x + (Dir :x * F) }
  "position"  :adjustY[ Dir, F ] = { self :y + (Dir :y * F) }

RULES

IF linearMotion( GD, P1, P2 ) & stroke( UserStroke )
    & equals( KinStroke, P1 :distance[ P2 ] )
THEN strokeFault( UserStroke - KinStroke )

IF strokeFault( X ) & smaller( X,0.001 ) & greater( X,0.001 :negated )
THEN succeed

IF strokeFault( F ) & direction( D ) & endPosition( P )
THEN gets( P :x, P :adjustX[ D, F ] )
    & gets( P :y, P :adjustY[ D, F ] )
    & positionChanged( P ) & use( adjustGeometryOfGuide )
```

Fig. 1. Example of a scenario

evaluated to true if they are successfully unified with the factbase.

A rule is fired if the condition is fulfilled, i.e. the sequence of predicates, either built-ins or queries, evaluates to true. Variables occurring as arguments of predicates are unified to constants and functions are evaluated, when the condition is evaluated. The rule tries to perform the action part using values for the variables found by the unification algorithm in the condition. The action part consists of built-in predicates or assertions of facts to the factbase. The former are either assignments of values to attributes of objects (e.g. gets(aFace :x, 8)), or activations of scenarios (e.g. use(geometryOfGuide)), or scenario termination predicates, succeed, or fail. The latter are used to extend the factbase.

2.2. Metamodel mechanism

In the previous section it was shown how the IDDL syntax looks like. Now, we concentrate on semantical aspects of the language. We present the metamodel mechanism of IDDL. Another paper by the same authors in this volume [11], introduced that the design process is considered as a mapping from function space to attribute space. The mapping cannot be accomplished directly. Instead, the design object model is refined in a step-wise manner. This mechanism is called *metamodel evolution*. We call the central description of the design object a *metamodel*.

The metamodel is the representation of the design object found in the factbase, i.e. the facts represent a metamodel. During the design process the factbase is extended, and hence the metamodel evolves. This evolution is achieved by executing scenarios. Scenarios are used to evaluate the metamodel in a certain context, i.e. they create an *aspect* model. An aspect model is used to evaluate a design object on various aspects, e.g. geometric aspects, or kinematic aspects, etc. First we give a short introduction to aspect models, thereafter we present the metamodel's role to keep consistency between several aspect models.

2.2.1. Aspect models. Aspect models are used to highlight a certain aspect of the design object. It is created by executing a scenario. The rules of the scenario represent the declarative knowledge about the aspect, the functions of the scenario denote the procedural knowledge. The rules evaluate the metamodel and add new facts or give values to objects' attributes. Therefore, an aspect model needs to have access to the internal structure of an object. A scenario has access to an object's attributes and functions by function calls. E.g. when the aspect model is a geometric model, these are geometric attributes and geometric functions.

An aspect model has knowledge to interpret a certain state of the metamodel, i.e. the factbase. It derives new information from the metamodel in terms of new facts, or new values for objects' attributes. But, since an aspect model has only knowledge about its own domain of expertise, viz. the aspect model's specialization, it does not know about the manipulation of objects' attributes by other aspect models.

2.2.2. Metamodel. In the metamodel a description of the design object which is independent of any aspect is found. It is the metamodel's task to keep track of all changes of objects achieved by aspect models. It organizes and controls all facts that are known about the design object description. Therefore, if an aspect model has changed a certain attribute of an object, the metamodel is notified. The metamodel keeps record of all changes caused by aspect models. A top level (meta) scenario examines the changes of the metamodel and takes proper actions. This top level scenario controls the metamodel mechanism. It creates aspect models by activating scenarios. It must recognize inconsistencies obtained by various aspect models.

Summarizing, we say that the metamodel is described in terms of objects and facts and aspect models are described in terms of objects, functions and attributes. The metamodel mechanism is driven by scenarios containing rules and functions. Fig. 2 shows the role of the scenario, linearMotionDesign activating two other scenarios, refineGuide and adjustGeometryOfSlot.

From the above we can conclude that there are two types of scenarios:

1. Scenarios which examine the current state of the metamodel, and which depending on that state activate a certain aspect model. Scenarios belonging to this category effect the metamodel mechanism. They control the evolution of the metamodel; they describe the global design process. Scenarios belonging to this category are called *metamodel scenarios.*
2. Scenarios which model the current state of the metamodel by creating aspect models on the metamodel. Their function is to add new information to the metamodel. They describe specific features of the design process. Scenarios belonging to this category are called *aspect scenarios.*

It is the metamodel scenario's task to activate proper aspect scenarios at a certain stage of the design process. Another task is to maintain the consistency between the data obtained by several aspect models. In the next section we give an elaborated example about how the metamodel mechanism works in practice.

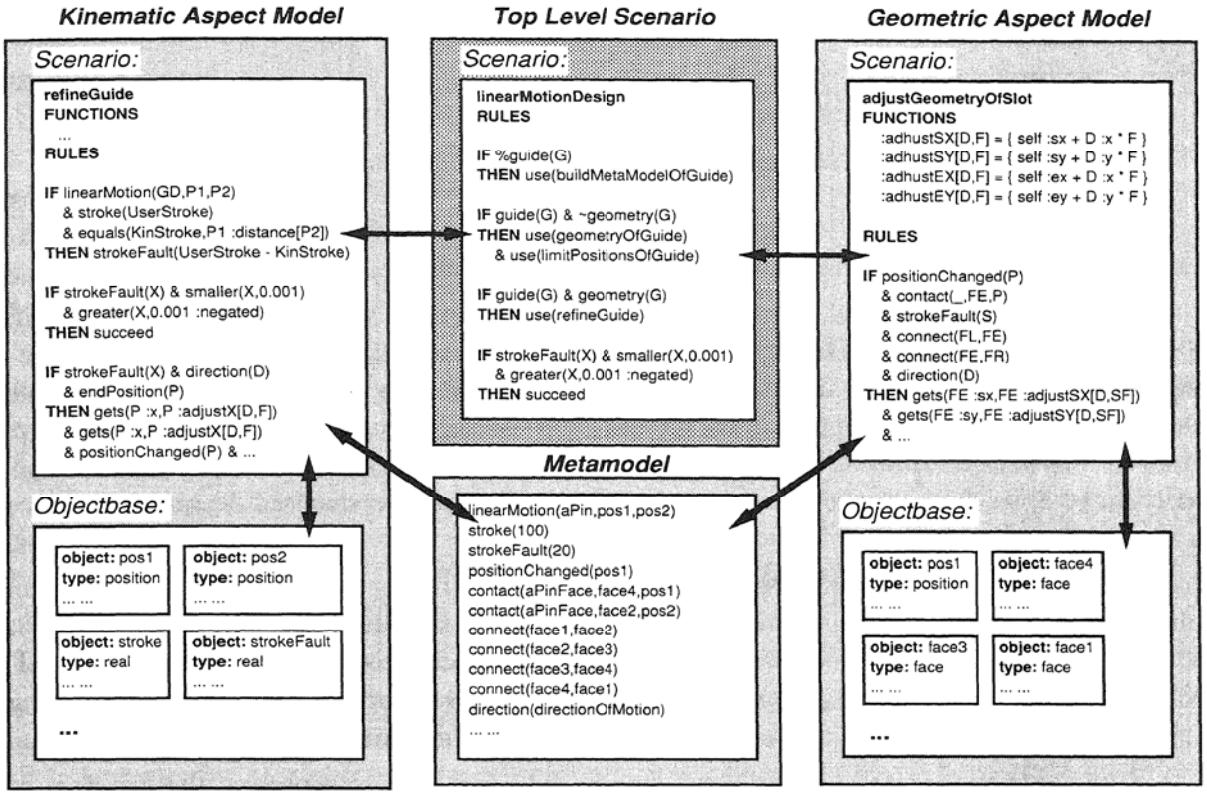

Fig. 2. Metamodel mechanism

3. Implementation of a Design Problem

In this section we describe the usage of the metamodel [11] in IDDL by showing the implementation of the example design problem. The problem involves the design of a linear motion mechanism. We show that it is feasible in IDDL to build a metamodel, which represents the solution for a certain category of design problems in a general way. In order to include new designs, the only thing a designer has to do is to add aspect dependent scenarios. The metamodel mechanism increases the possibilities to use the system for creative design, since scenarios for a new type of solution can easily be added. A restriction is that the type of the design problem stays within a known category for which there exists a metamodel description.

For designing a linear motion mechanism there are at least three different approaches. A first approach uses a slot and a pin, another uses a shaft and a slider, and a last approach uses a rail and a table to construct a linear motion mechanism (see Fig. 3). All three approaches employ the same metamodel description in IDDL. The aspect models which are created on the metamodel differ, e.g. each type of solution has its own geometric and kinematic models.

To aid the design of a linear motion mechanism, a number of scenarios are implemented in IDDL. We subdivide the linear motion design scenarios into two categories, the first category of scenarios are generally applicable to a linear motion design problem. The second category contains the scenarios which are applicable for a certain type of design solution, i.e. slot-pin, shaft-slider, or rail-table. The former are called *metamodel scenarios* while the latter are called *aspect*

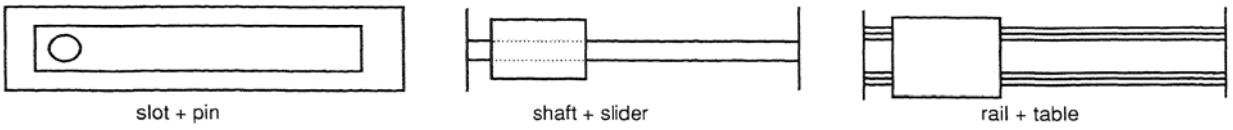

Fig. 3. Three possible approaches to construct a linear motion mechanism

scenarios. Each stage of the design process (i.e. conceptual, fundamental, and detailed) has its own sub-set of both categories of scenarios associated with it.

This section is subdivided into five parts. In the next section we give a precise definition of the linear motion design problem, and we present the metamodel associated with a range of design solutions. The scenario which controls the overall design process of a linear motion mechanism is presented in §3.2. In §3.3 we describe the conceptual design of a slot and a pin, chosen as the design solution. The fundamental design of this solution is presented in §3.4. Finally, in §3.5 we illustrate the last stage of the design process, i.e. detailed design.

3.1. Precise definition of the design problem

In the example used in this paper we concentrate on the metamodel representation and we omit other aspects. The design problem deals with a bounded linear motion between two points. The functional specification consists of the following facts: there is a direction vector $\vec{d}$ and a length of stroke $|s|$. Basically there are three types of solutions to this problem (see Fig. 3).

Let us commence with building a metamodel which is able to represent the problem in a solution independent way. We, therefore, state the following general facts. There is an entity, which we call objectInMotion, that moves between two positions (linearMotion(objectInMotion, pos1, pos2)). There is another entity, guide which supports the object in motion. At each of the positions there is a limit arrangement between the object in motion and the guide (limitArrangement(objectInMotion, guide, pos1) and limitArrangement(objectInMotion, guide, pos2)). The distance between the two positions is defined to be the desired stroke. A picture of the metamodel is shown in Fig. 4. It shows the state of the metamodel after conceptual design.

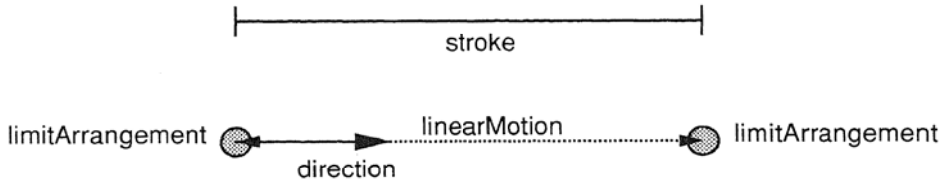

Fig. 4. Metamodel for a linear motion mechanism

At a later stage of the design process, when the geometry of the object in motion and the guide is defined, the metamodel is extended. One of the two positions in a limit arrangement is defined to be the start position of the linear motion. At this position we define a contact between a face of the object in motion and a face of the guide (startPosition(pos1) and contact(pinface, guideFace1, pos1)). The face of the guide is defined by the angle of its normal vector, which must be the same of that of the direction of the motion. By the same token, we define the end position of the motion. At this position the angle of the normal of the guide's face equals the

inverted angle of the direction vector (endPosition(pos2) and contact(pinface, guideFace2, pos2)).

At this point of the design process, the start position and the end position serve a dual purpose. They are defined by the geometric model as a contact between two faces, and in the kinematic model they define the stroke of the motion being the distance between the two positions. These two models can be inconsistent with each other. The metamodel is aware of this inconsistency because it has knowledge about the two models. Such an inconsistency is determined during the last stage of the design process, detailed design. The stroke fault is defined to be the difference between the desired stroke and the actual value achieved by the geometric model.

When the stroke fault is greater than a certain tolerance, the metamodel either creates a kinematic or a geometric aspect model to remove the inconsistency. In the former case we say that the geometrical properties are fixed and we change the desired stroke. In the latter case the geometry of the guide is changed in order to achieve a stroke fault smaller than the tolerance. The remainder of this paper is concerned with demonstrating the use of IDDL to implement the metamodel mechanism.

3.2. General solution to the design problem

In the previous section, we introduced a design problem and three types of solutions to the problem. Here, we show how the problem is actually solved using the pin-slot solution. We implemented two categories of scenarios, *metamodel* scenarios and *aspect* scenarios. The former work independently of a chosen type of solution. They construct the general metamodel, and guide the overall design process. The latter are specific to a particular type of solution. They are applied when a certain aspect of the design object needs to be examined and changed. In our example we apply geometric and a kinematic aspect models to derive new information about the metamodel.

The overall design process is controlled by the scenario linearMotionDesign depicted in Fig. 5. It contains four rules, the first three rules denote three consecutive stages of the design process, i.e. conceptual, fundamental, and detailed design. The fourth rule contains the stop condition for a successful completion of the design of a linear motion mechanism, viz. the absolute value of the stroke fault is smaller than a certain tolerance. The scenario and hence the design of a linear motion mechanism succeeds, if this condition is met. Below, we treat the first three rules in more detail.

The first rules depicted in Fig. 5 reads as follows: if there is not an object of type guide in the scenario's context, then activate a scenario which is called buildMetaModel. If use(buildMetaModel) succeeds, then the rule succeeds as well. BuildmetaModel creates an initial abstract anatomical model of the design object. It decides in dialogue with the designer which type of solution is chosen.

Next, the second rule is applied if there exists an object of type guide and the geometry of that object is not defined. Note that the query guide(G) succeeds if either a guide or an object which is defined as a subsort of a guide is found. The rule executes the scenarios geometryOfGuide and limitPositionsOfGuide in order. It builds a concrete anatomical model of the design object by defining the objects' geometrical structures and by defining kinematic properties.

```
linearMotionDesign
    "design a linear motion mechanism"
RULES

IF %guide( G )
THEN use( buildMetaModel )                    "Conceptual design"

IF guide( G ) & ~geometry( G )                "Fundamental design"
THEN use( geometryOfGuide )
     & use( limitPositionsOfGuide )

IF guide( G ) & geometry( G )                 "Detailed
design"
THEN use( refineGuide )

IF strokeFault( X ) & smaller( X, 0.001 ) & greater( X, 0.001 :negated )
THEN succeed
```

Fig. 5. Top-level scenario

The third rule executes the scenario refineGuide, if the geometry of a guide is defined. RefineGuide checks the metamodel for discrepancies between the geometric and kinematic models and adjusts the metamodel accordingly. These discrepancies are represented by the stroke fault. If the fault of the stroke is within a predefined limit, then the design is completed.

Each of the first three rules of linearMotionDesign represents a certain stage of the design process. The first rule performs conceptual design, the second rule fundamental design, and the third rule stands for detailed design. The rules are executed in order in a circular fashion. Backtracking over these rules proceeds as follows. In this scenario a condition of a rule can only be met if the previous rule succeeded. For example, the first rule is executed as long as guide(G) is unknown. In other words, an object of type guide cannot be found in the metamodel. By the same token, the third rule is applied as long as the fault of the stroke is not within a certain margin. In this case, the system can force backtracking over the previous rule by retracting the fact that the geometry of a guide is defined. In that case the second rule is applied again. In the following three sections we explain each of these three design stages, and we show the state of the metamodel at the end of each stage.

3.3. Conceptual design of link and pin

In this section we show how the conceptual design of a linear motion mechanism is performed. A single scenario is responsible for establishing the metamodel structure; it defines an abstract anatomical representation of the design object. The scenario is shown in Fig. 6.

The scenario buildMetaModel composes the metamodel structure in a very straightforward way. The first rules asks the designer to provide the length of the stroke which the linear motion mechanism must reach, the value must be greater than zero. Secondly, it asks for the direction of the stroke. The direction is stored in the metamodel as a normalized vector. Rule two through four determine what kind of solution is chosen. The scenario distinguishes between three solutions. The first solution is cheap but less reliable (viz. slot and pin). A quite reliable but more

```
buildMetaModel
        "create the metamodel for a linear motion mechanism"

IF %stroke( S ) & uiNumber( lengthOfStroke, S ) & greater( S, 0 )
        & uiNumber( directionOfMotionX, X ) & uiNumber( directionOfMotionY, Y )
THEN stroke( S ) & direction( directionOfMotion )
        & gets( directionOfMotion :x, X :normalize[ Y ] )
        & gets( directionOfMotion :y, Y :normalize[ X ] )

IF stroke( S ) & ui( cheapButLessReliable )
THEN pin( aPin ) & slot( aSlot )

IF stroke( S ) & %guide( GD ) & ui( reliableButMoreExpensive )
THEN shaft( aShaft ) & slider( aSlider )

IF stroke( S ) & %guide( GD ) & ui( veryReliableAndExpensive )
THEN rail( aRail ) & table( aTable )

IF objectInMotion( OM ) & guide( GD )
THEN position( pos1 ) & position( pos2 ) & linearMotion( OM, pos1, pos2 )
        & limitArrangement( OM, GD, pos1 ) & limitArrangement( OM, GD, pos2 )
        & succeed
```

Fig. 6. Scenario for conceptual design of a linear motion mechanism

expensive solution is provided by the second choice (viz. shaft and slider). A last solution is very reliable but also very expensive (viz. rail and table). These three choices are depicted in Fig. 6.

The last rule of buildMetaModel stores the properties of a linear motion mechanism in the metamodel. It behaves as follows: if there are objects of type objectInMotion (OM) and guide (GD), then assert that there are two positions, pos1 and pos2, there is linear motion of OM between these positions, and there are two limit arrangements for OM and GD, one at pos1 and one at pos2. When the scenario buildMetaModel is completed, the metamodel contains The following facts (the stroke is assigned to the value 100 by the designer):

```
pin(aPin)           direction(directionOfMotion)
slot(aSlot)         limitArrangement(aPin,aSlot,pos1)
stroke(100)         limitArrangement(aPin,aSlot,pos2)
position(pos1)      linearMotion(aPin,pos1,pos2)
position(pos2)
```

The metamodel contains nine facts, six of these are object definitions. Note that the properties which describe the function of the mechanism, limitArrangement and linearMotion are asserted independent of the chosen mechanism (slot and pin). The metamodel only assumes that there are objects of type objectInMotion and guide. Once we have built the metamodel we can continue the design process with fundamental design.

3.4. Fundamental design of link and pin

At the beginning of this stage of the design process the metamodel consists of abstract anatomical description of the design object. During the course of fundamental design the metamodel is transferred to a concrete anatomical structure. At this stage the geometrical properties of the design object are defined by a geometric aspect model, and the requirements for the desired stroke length are determined by a kinematic aspect model. The aspect models are created by the scenarios geometryOfGuide and limitPositionsOfGuide respectively. These scenarios are activated by the top-level scenario introduced in §3.2. We start with explaining the scenarios for creating a geometric aspect model.

The scenario geometryOfGuide activates a geometric aspect model of the metamodel. It contains three rules, and each of these rules activates an *aspect* scenario belonging to a particular solution. If that scenario succeeds, then the fact stating that the geometry of the chosen solution is defined is asserted to the metamodel. In case of the design of a slot and pin solution, the scenario geometryOfSlot is selected. By doing so, an aspect model for a specific solution is created on the metamodel. When the execution of the scenario is completed, we assert to the metamodel that the geometry of the slot is defined (geometry(aSlot)).

The scenario geometryOfSlot contains both functions and rules denoting procedural and declarative knowledge respectively. The rules perform the geometric reasoning about a slot and pin, the functions denote geometric procedures. The first rule creates a number of faces, one face (pinFace), which is a part of a pin, and four faces (face1, face2, face3, and face4), which are part of a slot. The four faces are connected with each other in a circular way (see Fig. 7). The second rule activates two scenarios which initialize some attribute values of a slot and a pin. A pin receives a value for its diameter, and a slot receives a value for its length and width. These values are requested from the designer, they are treated as default values. These values can be made certain, or can be changed at a later stage of the design process.

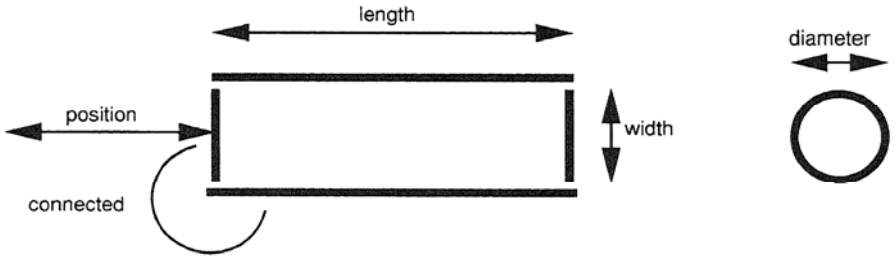

Fig. 7. Geometry of a slot and a pin

The third rule determines start- and end-points of the four faces of a slot. A face contains four attributes, which denote the x- and y-coordinates of the start- and end-point (sx, sy, ex, and ey). These coordinates are set as follows. The start-point of a arbitrary face is set to the origin (0,0), we call this point the left bottom point. The orientation of the faces is anti-clockwise. Hence, the end-point of the first face is the right bottom point. The scenario has a function which calculates the x- and y-coordinate of the right bottom point by using the length of the slot and the normalized direction vector of the stroke. If two faces are connected, then the end-point of one face is equal to the start-point of the other. For each corner of the slot there exists a function

to calculate its corner. So, this process is repeated until the coordinates of the end-point of the fourth face are set to (0,0). The scenario succeeds when this is done.

Once the geometries of the slot and the pin are defined, the scenario limitPositionsOfGuide is activated. It builds a kinematic model, which is able to calculate the stroke of the achieved geometry (see Fig. 8). This model determines the limit positions of the object in motion.

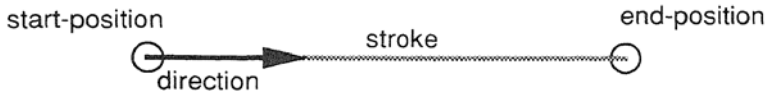

Fig. 8. Kinematic model of an object in motion

The distance between these limit positions determines the stroke of the linear motion mechanism. The first rule of limitPositionsOfGuide (see Fig. 9) queries the metamodel for a limit arrangement between the object in motion and the guide at a certain position. This position is defined as the start position of the motion. Furthermore, the rule asserts that at this start position there is a contact between the face of the object in motion and a face of the guide, whose normal vector has the same angle as the direction vector. We defined the faces of the guide in a anti-clockwise direction, so the normal vector is uniquely defined.

```
limitPositionsOfGuide
        "determine limit positions of two faces"
RULES

IF limitArrangement( OM, GD, Pos ) & face( F1 ) & face( F2 )
        & hasPart( OM, F1 ) & hasPart( GD, F2 )
        & direction( D ) & equals( D :tangent, F2 :normal )
THEN startPosition( Pos ) & contact( F1, F2, Pos )

IF limitArrangement( OM, GD, Pos ) & face( F1 ) & face( F2 )
        & hasPart( OM, F1 ) & hasPart( GD, F2 )
        & direction( D ) & equals( D :inverted, F2 :normal )
        & ~contact( Skolem1, Skolem2, Pos )
THEN endPosition( Pos ) & contact( F1, F2, Pos )

IF slot( S )
THEN use( limitPositionsOfSlot ) & succeed

IF shaft( S )
THEN use( limitPositionsOfShaft ) & succeed

IF rail( R )
THEN use( limitPositionsOfRail ) & succeed
```

Fig. 9. This scenario determines the limit positions of the mechanism

Since we defined two limit arrangements, the end position of the motion is determined by the second rule in Fig. 9. The rule finds a face of the guide, whose normal has the same angle as the inverted direction vector. The fact that there is a contact between this face and the face of the object in motion at this end position is asserted to the metamodel as well as the end position

itself. Rule three through five determine the actual coordinates of the end and start positions in a geometric model which depends on the chosen design solution.

When the geometric model, activated by one of these rules, has performed its task, the kinematic model is complete and limitPositionsOfGuide succeeds. Control is given back to the top level scenario mentioned in §3.3. The metamodel is extended with a number of facts. These facts are (the hasPart predicates are omitted from the list):

```
connect(face1,face2)          face(face1)
connect(face2,face3)          face(face1)
connect(face3,face4)          face(face2)
connect(face4,face1)          face(face3)
contact(pinFace,face2,pos2)   face(face4)
contact(pinFace,face4,pos1)   face(pinFace)
endPosition(pos1)             geometry(aSlot)
startPosition(pos2)
```

3.5. Detailed design of link and pin

The final stage of the design of a linear motion mechanism is now reached. Both a geometric and kinematic model of the design object have been obtained. However, between these models there might be some inconsistency due to imprecise specifications by the user. The obtained geometry might result in an incorrect stroke. The metamodel is able to recognize such an inconsistency, since it has knowledge about both models. In this section we show how the metamodel detects an inconsistency by means of a kinematic model and repairs it by changing the geometric model of the design object (see Fig. 10).

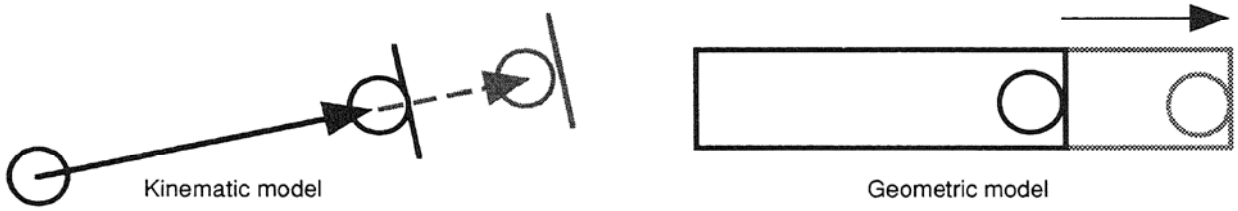

Fig. 10. Stroke-fault detected in the kinematic model affects the geometric model

The scenario refineGuide (see Fig. 1.) calculates the stroke fault in its first rule. The stroke fault is defined as the difference between the stroke given by the user and the stroke determined by the kinematic model. The kinematic model applies its knowledge that the stroke of a linear motion is defined as the distance between the start position and the end position of the motion. The third rule of the scenario adjusts the x- and y-coordinate of the end position according to the calculated stroke fault. It moves the end position along the direction vector. Furthermore, the rule asserts that the position of the end position is changed and it activates a scenario which adjusts the geometry of the guide in accordance with the changed end position.

The scenario adjustGeometryOfGuide activates, depending on of the chosen solution, a scenario which changes the geometry of the slot. It determines which face of the slot is in contact with the pin in the changed position. The scenario adjusts the coordinates of this face and

those of the two faces which are connected with it. It also updates the length of the slot. Control is give back to refineGuide which calculates a new stroke fault and succeeds if the stroke fault is smaller than a certain tolerance.

The scenario refineGuide detects the proper face to be adjusted, since the metamodel has a description of the behaviour of an object in motion guided by a slot. This knowledge can be represented in neither the geometric model nor the kinematic model. Therefore, without a metamodel the system would not have been able to create a geometric model independent of certain properties which are determined by a kinematic model. The metamodel avoids this inflexibility by introducing a general model of the design object (e.g. the limit arrangements) independent of a certain context. The fact representing a contact between a face of the slot and a face of the pin at a certain position, is found because the metamodel has ontological knowledge about the relation between kinematic motion and geometry.

4. Conclusions

In this paper we showed, how IDDL can be used to represent a design object model, which is independent of a certain context. The metamodel mechanism is used to provide such a representation. In a previous paper we have presented three roles of the metamodel [8]:

1. a central modeling mechanism to integrate aspect-models
2. a mechanism for modeling physical phenomena
3. a tool for describing evolving objects

In the two combined papers in this volume we have showed, how the first and third role of the metamodel can be represented in IDDL. Besides, we have introduced ontological knowledge representation as a technique to fulfill the second role of the metamodel. Creative design can be modeled by allowing the designer to create new scenarios which are applicable to a new design. These scenario operate on a metamodel which describes a solution to a design problem in a general way. In order to aid the designer in writing new scenarios the system must have knowledge about the physical properties of design objects. Ontological knowledge in the system provides such knowledge.

References

1. W.F. Clocksin and C.S. Mellish, *Programming in Prolog,* Springer-Verlag, Berlin, 1981.
2. R. Davis and J. King, ''An Overview of Production Systems,'' in *Machine Intelligence 8*, ed. E.W. Elcock, and D. Michie, pp. 300-332, Ellis Horwood Ltd., Chichester, 1977.
3. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, Reading, MA, 1983.
4. H. Lieberman, ''Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Languages,'' *OOPSLA '86, Special Issue of SIGPLAN Notices*, vol. 21, no. 11, 1986.
5. J.W. Lloyd, *Foundations of Logic Programming,* Second, Extended edition, Springer-Verlag, Berlin, 1987.

6. M. Minsky, ''A Framework for Representing Knowledge,'' in *The Psychology of Computer Vision*, ed. P.H. Winston, pp. 211-277, McGraw-Hill, New York, 1975.

7. T. Tomiyama and P.J.W. ten Hagen, ''The Concept of Intelligent Integrated Interactive CAD Systems,'' CWI-Report CS-R8717, 1987.

8. T. Tomiyama, T. Kiriyama, H. Takeda, D. Xue, and H. Yoshikawa, ''Metamodel: A Key to Intelligent CAD Systems,'' *Research in Engineering Design*, vol. 1, no. 1, pp. 19-34, 1989.

9. T. Tomiyama, D. Xue, and Y. Ishida, ''An Experience with Developing a Design Knowledge Representation Language,'' in *Intelligent CAD Systems III – Practical Experience and Evaluation*, ed. P.J.W. ten Hagen, and P.J. Veerkamp, Springer-Verlag, Berlin, 1990. Forthcoming.

10. P.J. Veerkamp, V. Akman, P. Bernus, and P.J.W. ten Hagen, ''IDDL: A Language for Intelligent Interactive Integrated CAD Systems,'' in *Intelligent CAD Systems II – Implementational Issues*, ed. V. Akman, P.J.W ten Hagen, and P.J. Veerkamp, pp. 58-74, Springer-Verlag, Berlin, 1989.

11. P.J. Veerkamp, T. Kiriyama, D. Xue, and T. Tomiyama, ''Representation and Implementation of Design Knowledge for Intelligent CAD – Theoretical Aspects,'' *in this volume*, Compiegne, 1990.

12. P.J. Veerkamp, R.S.S. Pieters Kwiers, and P.J.W. ten Hagen, ''Design Process Representation in ADDL,'' in *Intelligent CAD Systems III – Practical Experience and Evaluation*, ed. P.J.W. ten Hagen, and P.J. Veerkamp, Springer-Verlag, Berlin, 1990. Forthcoming.

13. B. Veth, ''An Integrated Data Description Language for Coding Design Knowledge,'' in *Intelligent CAD Systems I – Theoretical and Methodological Aspects*, ed. P.J.W. ten Hagen, and T. Tomiyama, pp. 295-313, Springer-Verlag, Berlin, 1987.