On the Development of an Artifact and Design Description Language

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

This paper discusses the development of a programming language: ADDL (Artifact and Design Description Language). ADDL is designed for the implementation of Intelligent CAD (Computer Aided Design) systems. Its explication proceeds in three stages. First, we work out a model of the design process. ADDL must have constructs to represent this model. Second, we formulate the requirements ADDL must fulfill in order to describe the design process model presented. Those requirements are expressed by means of design maxims. Third, from these design maxims the language specifications are generated and an experimental ADDL compiler and interpreter are built. In this paper, we deal with the first two stages of the development of ADDL.

1. Introduction

1.1. Subject of the paper

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 special properties which ICAD systems require. Therefore, we developed a special purpose programming language: ADDL (Artifact and Design Description Language). This paper deals exclusively with the design and implementation of ADDL.

1.2. History of CAD

Early CAD systems were biased towards geometric information. A user was given a tool to generate a drawing of the artifact. A next generation was equiped with a database where product data could be stored and retrieved. However, during several stages of the design process a product's specification

CWI Quarterly 3, 219-236

needs to be verified. This can be achieved by a FEM (Finite Element Method) analysis module, or by a cost analysis module, etc. These are separate tools, forcing the CAD data to be transferred from one system to another, and back. Thus, a future CAD system needs to be an *integrated* system which contains a central design-object model, and which has several application modules connected to it, allowing the designer to analyse his product in several ways. Such a system employs a uniform language which is used by all subparts.

1.3. Programming languages for CAD

From the above we may conclude that an ICAD system puts high demands on the programming language that is used for its 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-object 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 relations between several parts of the design-object. Design process knowledge is represented by if-then rules 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.
- It must offer the means for high level interaction with a designer, i.e. good human computer interaction facilities.

We believe that a language, which is based on both objects and logic, forms a firm basis for implementing an ICAD system. In this section we shortly introduce logic programming and object-oriented programming. In the next section we present a language which is based on these paradigms.

1.3.1. Logic programming. The fundamental idea is that first order logic can be used as a programming language. Logic consists of propositions and relations among propositions [4,13]. Furthermore, there is an inference engine which can infer propositions from others, and which can validate propositions. Propositions consist of predicates and terms. Most inference engines of logic programming systems are based on resolution theorem proving [15]. Our programming language is equipped with logic for the representation of declarative knowledge about design-objects, i.e. relationships among objects. We have enriched our logic with modal operators [11] to add uncertainties and modalities to these relationships.

1.3.2. Object-oriented programming. An object-oriented programming language is based on a single universal data structure (the object), a general control structure (message passing), and a general data description structure (the class hierarchy). An object is a way to represent properties of a data structure and operations allowed on that data structure in a single location. A program obtains information from an object in answer to a message sent to that object. Message passing may also be used to give a task to an object. Objects which have a common behaviour and related properties are grouped together in a uniform description, viz. a class. Classes are defined by means of other classes, i.e., a hierarchical organization. The objects themselves are responsible for the way a message is executed. Each object has an internal state where the effect of all messages sent to it is stored [8,9]. Although we use well-established object-oriented methods, the way we have combined them with logic programming is especially designed with ICAD requirements in mind. We have implemented our programming language in a Smalltalk-80[1] programming environment, using existing constructs as much as possible.

1.4. The IIICAD system

The IIICAD (Intelligent Integrated Interactive CAD) system is a computer system under development which assists the designer in such a way that he[2] can fully employ his creative skills [3]. Such a system allows the designer to create a central model of the object to be designed. From this model the designer can derive several *aspect* models, e.g. a kinematic model, a geometric model, a finite element model, etc. Aspect models highlight a certain aspect of the design-object model. The design-object model is evaluated in stepwise manner from a rough incomplete description to a detailed complete description [1]. Routine tasks and labour intensive tasks are done by the system [18].

1.4.1. Design process model. The IIICAD architecture is derived from a model of the design process. It is a general model which describes the way a designer performs a certain design job. The design process is regarded as a mapping from the function space, where the specifications are described in terms of functions and behaviour, onto the attribute space, where the design solutions are described in terms of attributes. Each function given by the initial specification is mapped to an attribute of the resulting design-object model. Roughly speaking, we state that a designer starts with a functional specification of a design-object and ends with a manufacturable description [19].

The basic ideas behind the design process model are as follows. According to the given functional specification a candidate for the design solution is selected. This candidate has a very incomplete description. It is chosen from a library of *prototypes*. This candidate is refined in a stepwise manner until the

1. Smalltalk-80 is a registered trademark of Xerox Corp.
2. Throughout the paper the male form is used when both female and male are meant.

solution is reached. The design process is thus regarded as an evolutionary process which transfers the design-object model from one state to another. Transfer of the model is accomplished by generating aspect models on the current central model. New information is derived from these models and it is added to the central model resulting in a new state.

1.4.2. Design maxims for an ICAD language. The design process model is based on a theory of design, i.e. design process and design-object. Requirements for an Artifact and Design Description Language (ADDL) are derived from the design process model. These requirements resulted in a number of design maxims. The development of ADDL was done by collecting these design maxims and converting them into language specifications.

1.4.3. An artifact and design description language. ADDL is a special purpose programming language for implementing ICAD systems. It allows for dynamic design-object descriptions and a flexible design process representation. ADDL has it roots in both object-oriented and logic programming. Furthermore, it is enriched with special constructs to meet the requirements posed by the above mentioned design maxims. These extensions include: scenarios to describe the design process, multiple worlds to model several aspects of the design-object simultaneously, and default and assumed values to handle uncertainties [24].

1.4.4. ADDL vs. IDDL. ADDL is being developed at CWI. Originally it was called IDDL (Integrated Data Description Language), but since there are now two versions of IDDL in development (one at CWI, and one at the University of Tokyo), we decided to change the name to ADDL [21,22]. ADDL and IDDL denote in principle the same language, they only differ in certain aspects.

2. Design process model

In this section we give a formalization of the design process. In the first subsection we subdivide the entire design process into several design phases from the initial specification to a manufacturable description of the product. In §2.2 we present a model which gives a representation of the design process. This model describes design as a stepwise refinement process where a central description of the design-object (hereafter called: *meta-model*) is transferred from one state to another. In §2.3 we show how a single step in that model is performed by the meta-model evolution scheme.

2.1. Design theory

There are three distinguishable successive stages in the design process: *conceptual design, fundamental design,* and *detailed design.* Each of these stages has its own demands on design-object representation. The design process representation, however, can be captured by the meta-model evolution scheme presented in §2.2. The sub-sections below treat each of the design stages separately.

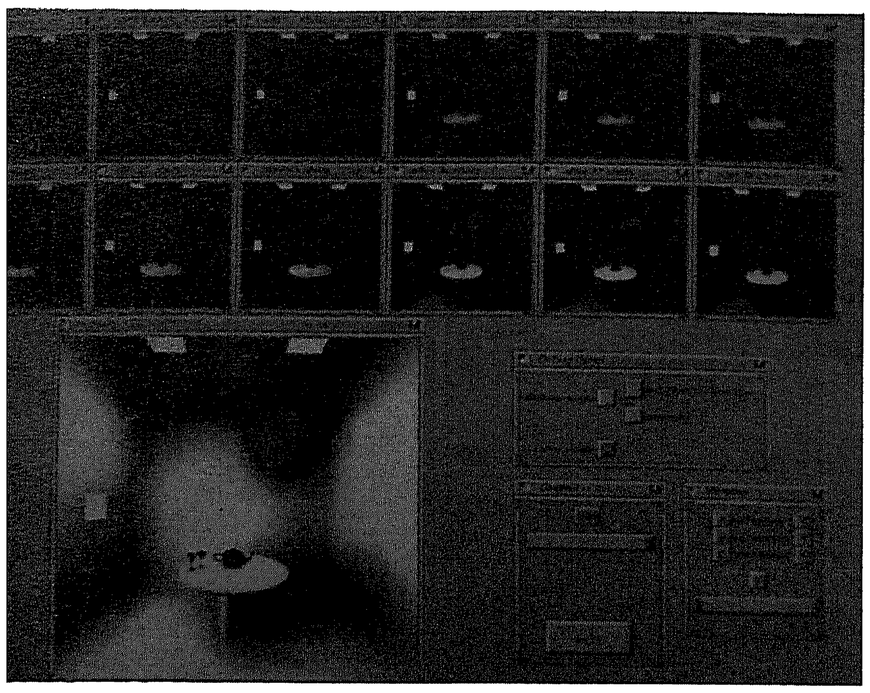

Plate VII - Radiosity at work

This plate visualizes the iterative approach of a parallel implementation of the radiosity algorithm. Radiosity is a realistic shading method based on simulation of the balanced flow of energy within a scene. The parallel implementation runs on a number of workstations that are connected via a network. The top windows show a sequence of progressively improving partial solutions of the scene. At first the flow of energy is highly unbalanced, but with each iteration step a better approximation of the equilibrium is found. The bottom window shows the scene after a number of iterations. The panels on the right hand side allows a user to control certain crucial parameters during the rendering process.

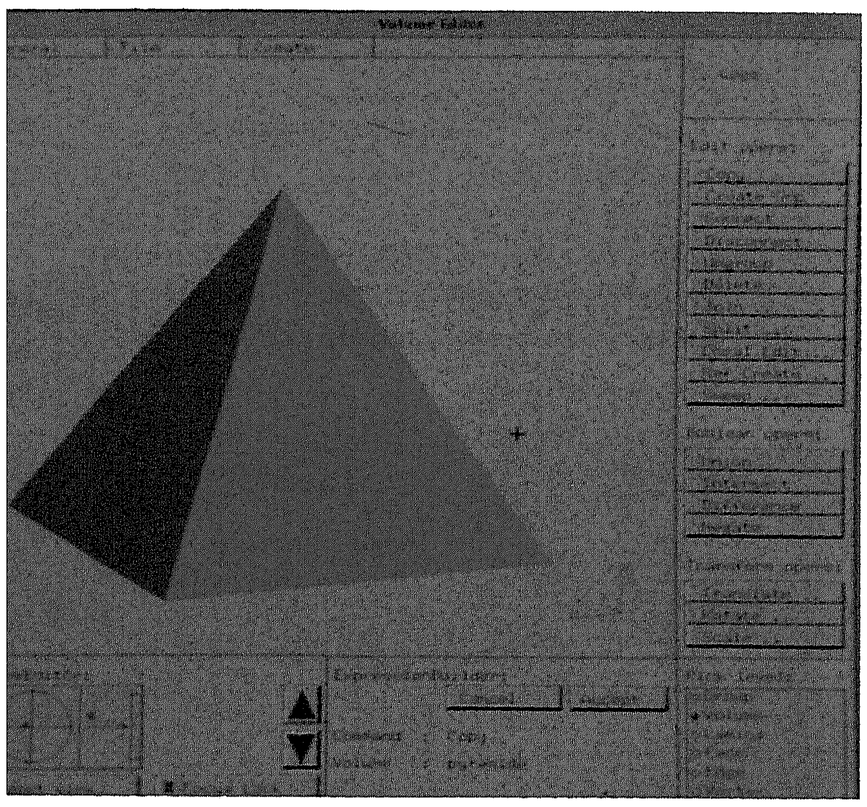

Plate VIII - DICE interface

A simple user interface for a solid modelling system programmed in the DICE user interface language showing a 3-D object, simulated dials and pushbuttons to control viewing and option selection respectively. The DICE system allows the user interface to perform all the manipulation without the necessity for the application to interfere, even when such operations are combined.

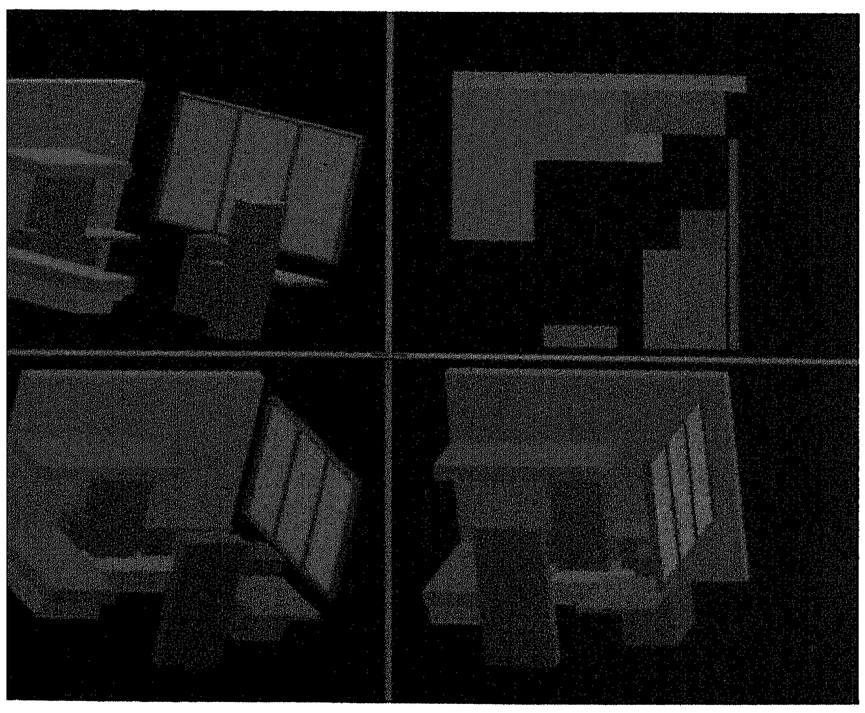

Plate IX - Design

The current IIICAD (Intelligent Integrated Interactive CAD) system is a pilot system for future design systems. Such systems act as 'intelligent apprentices' to the designers rather than as 'automated design systems'. The picture shows the configuration of a bedroom. The actual design is carried out with the aid of a prototype IIICAD system developed at CWI. The system employs a library of standard components for the selection of the parts of the room, and parts of these. The designer can freely choose components from the library and can adjust them as (s)he likes. The 3-D viewing and illumination is accomplished by a proprietary visualization tool using Silicon Graphics' GL library. (Courtesy J. Rogier and F. Kuijk)

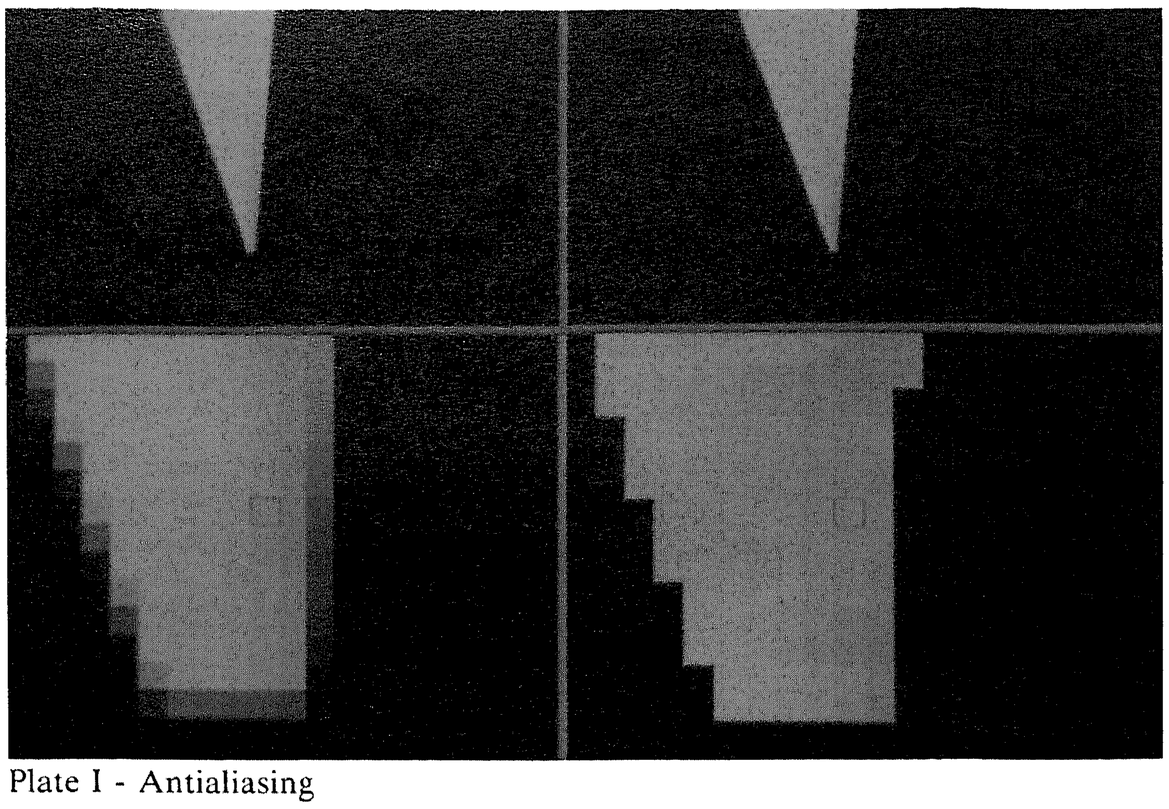

Plate I - Antialiasing

This plate shows the result of antialiasing by exact area sampling. The image on the upper right hand side is aliased. Pixels are simply taken to be either in- or outside a polygon, resulting in 'staircase' like edges. This is clearly visible in the 20-fold magnified image shown at the bottom. By weighing the pixel intensity based on the pixel-area actually covered by the polygon, we obtain a more attractive, smooth edged image. Note that the bottom-line of pixels of the polygon shown on the magnified image on the left have a reduced intensity. This illustrates the ability to handle sub-pixel positioning. As a consequence the apparent resolution is higher, even objects smaller than the size of a pixel become 'visible'.

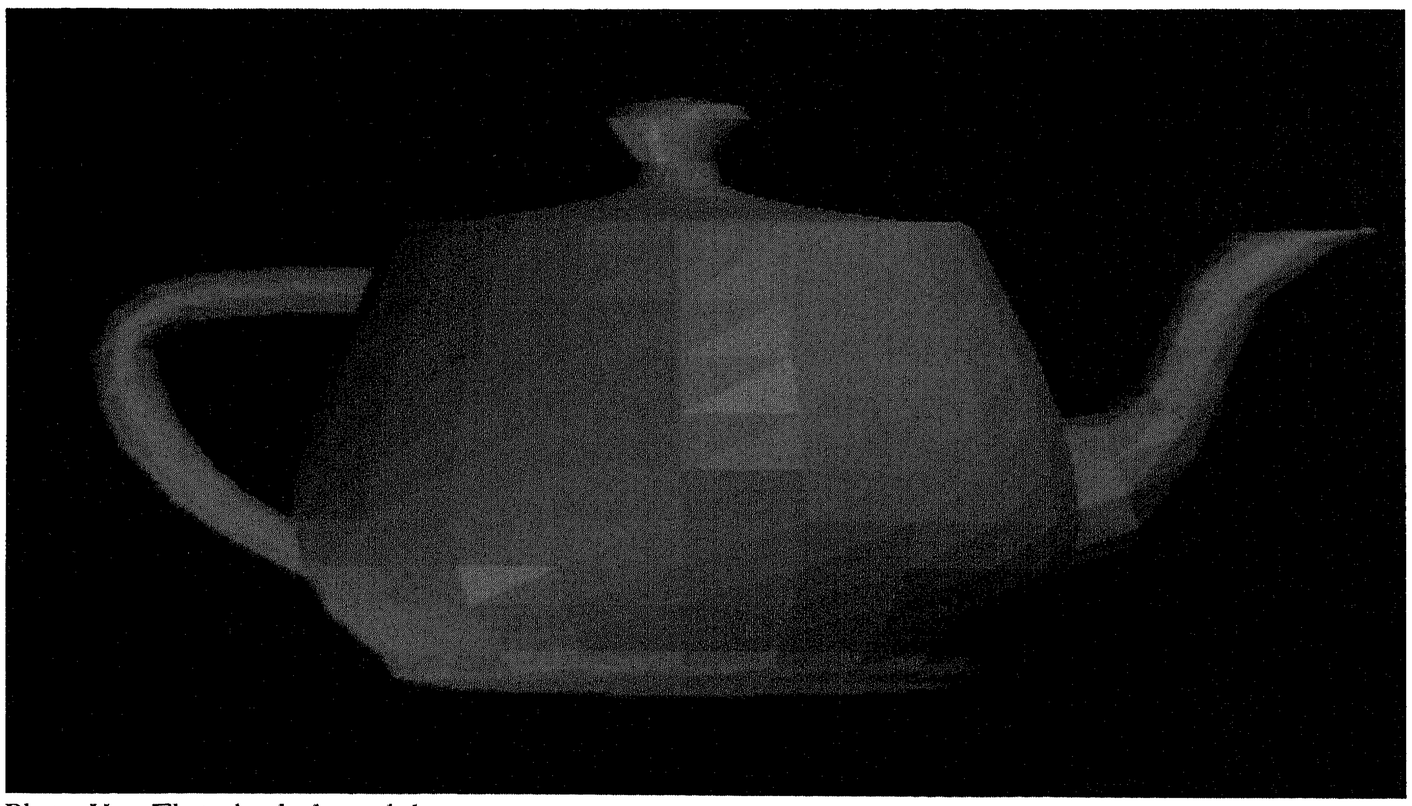

Plate II - Flat shaded model

This 3D model of a teapot has become a standard reference model to compare different shading algorithms. To be able to visualize a smoothly shaped teapot, the model is represented by a number of triangular facets. In this image each facet is simply shaded with a constant color. This color is calculated based on the orientation of the facet (i.e. its surface normal) with respect to the position of two light sources and the point of view. As a result the visualized model is far from being smooth.

Plate III - Gouraud shaded model

The improved quality of this image as compared to the previous one is obtained by performing a similar calculation of the color as in the above, but then for each individual corner of the triangle. For this, on each corner the average of the normals of all adjacent facets is taken as input for the calculation. The color of the interior points are obtained by linear interpolation of the colors found at each corner. In spite of this rather smooth shading at some points of the visualized model individual facets can still be identified.

Plate IV - Phong shaded model

A further improvement can be obtained by again applying the same color calculation but then for each individual interior point. A 'surface normal' vector needed for the calculation at each interior point is obtained by interpolating the averaged normal vectors found on each corner. The improved quality is most apparent for highlights, a peak in the intensity caused by the reflective property of the surface. In contrast with the previous method, a highlight is guaranteed to be visible even if it would fall in the middle of a facet, individual facets can no longer be distinguished and surfaces become more shiny. The computational costs however are substantially higher.

Plate V - Simulated hardware shaded model

This plate shows the result of an incremental method to calculate the color at each interior point. This method, although generally applicable, was developed at CWI to be implemented on special purpose hardware. The result is competitive with respect to the more costly Phong shading method, in fact it is even more smooth.

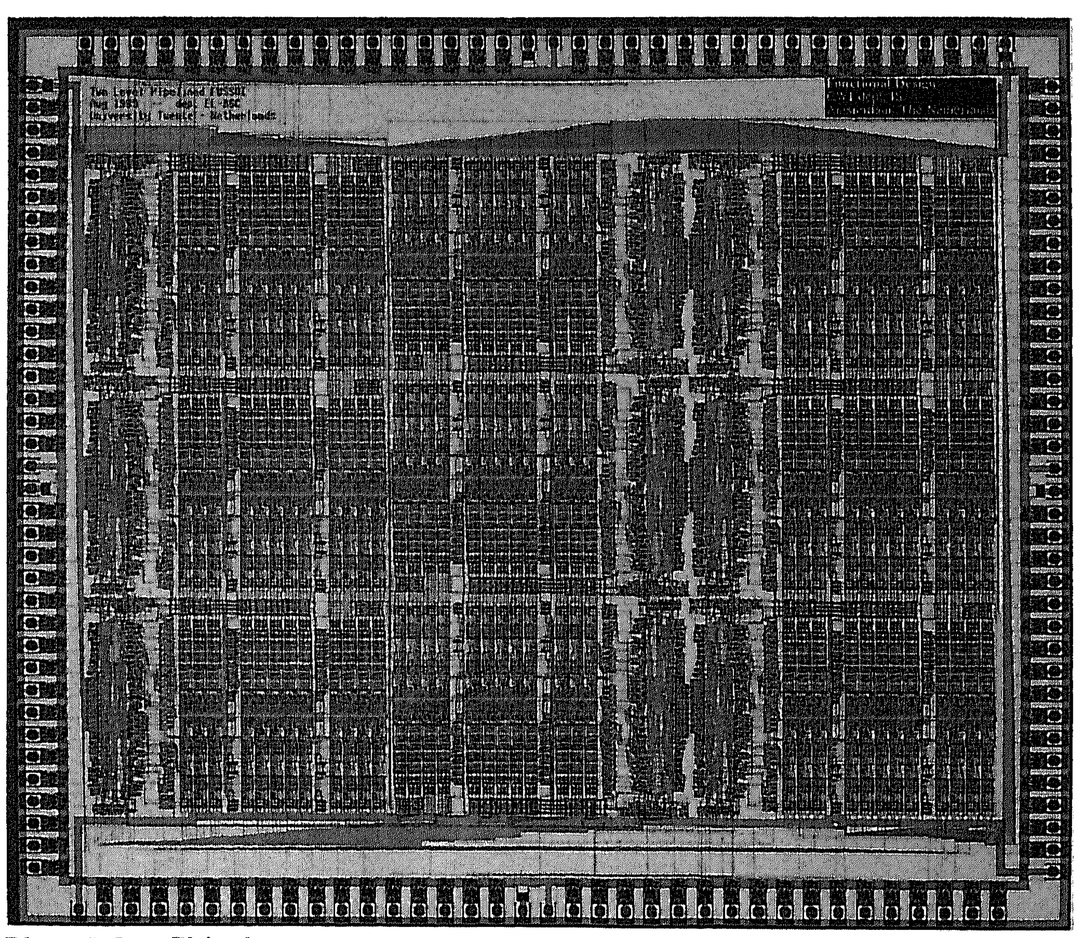

Plate VI - Chip layout

Here we see the layout of a chip designed for incremental color evaluation based on forward differencing. An array of processors (nine per chip) is able to produce a continuous pixel stream at a rate of 90 MHz, which is sufficiently fast to be able to generate the video signal of a high resolution display. As a result, pixel storage in the form of a frame buffer is not needed.

2.1.1. Conceptual design. The conceptual phase of the design process deals with the translation of the initial specifications, given to the designer, into a representation of the functions and behaviour of an artifact. The core system we present in this paper is developed to represent the second and third stage of the design process. It starts with a given functional specification of an artifact. The conceptual design is captured by a separate module which transforms —in dialogue with the designer— the initial specification to a functional specification. The functional specification is then further processed by the core system.

2.1.2. Fundamental design. During the course of the fundamental phase the functional specifications are converted into something we can actually make; a rough decomposition of the artifact is created. The principal shape and concrete structure without details is the result. The physical embodiment may require or introduce new constraints on the design-object. The design-object model is represented by a decomposition containing empty and fuzzy parts. The behaviour of the model is simulated by assuming default behaviour for the parts which are uncertain or unknown. Several possible design strategies may be applied simultaneously, resulting in either single or multiple models.

2.1.3. Detailed design. During the detailed design stage, the decomposed model is further refined. Dimensions and tolerances are now set and a complete description of the design-object is produced. All constraints are satisfied and all parts are integrated into a single coherent model. The design focuses on specific parts of the design-object model without worrying about global issues. Local optimizations are achieved which result in small changes. There is thus a need for multiple viewpoints of specific parts of a common representation, i.e. a multiple focus of attention.

2.2. Design process representation

In this section we give a general model of the design process which is applicable to the fundamental and detailed stages of the design process [23]. This model is employed in the IIICAD system to define the system architecture and develop language constructs for ADDL.[3] The model guides the design process as it is executed by the system in order to understand the designer's demands. In other words, the designer decides how to perform the design and the IIICAD system is an intelligent aid to the designer to assist him in achieving his goal [2].

3. Note that the first stage, conceptual design, is captured in the IIICAD system as well. It is defined as a front-end to the system covered by the Intelligent User Interface (IUI). The designer's specifications are transformed to functional specifications through a dialogue between the designer and the system.

2.2.1. From specification to solution. We use General Design Theory as a basis for giving a formalization of the design processes and knowledge. The theory is based on axiomatic set theory and models design as a mapping from the function space where the specifications are described in terms of functions, onto the attribute space where the design solutions are described in terms of attributes (see Fig. 2.1). Roughly speaking, one starts with a functional specification of the design-object and ends up with a manufacturable description.

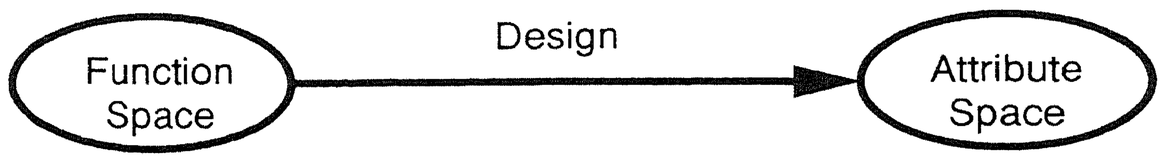

FIGURE 2.1. Design process model

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

- From the given functional specifications a candidate is selected and refined in a stepwise manner until the solution is reached, rather than by trying to get the solution directly from the specifications. The latter is not possible in a non-trivial design problem, since it involves a very complex object with a multitude of parts.
- The design process is regarded as an evolutionary process which transfers the model of the design-object from one state to another, gradually obtaining a more detailed description.
- To evaluate the current state of the design-object model, various interpretations of the design-object model need to be derived in order to see whether the object satisfies the specifications or not. We call those interpretations of the design-object model *worlds* and they can be regarded as interpretations of the design-object observed from different points of view.

2.2.2. Stepwise refinement of meta-model. Considering the general design model, the system starts from the specification S of the design-object and continues the design process until the goal G is reached (see Fig. 2.2 below).

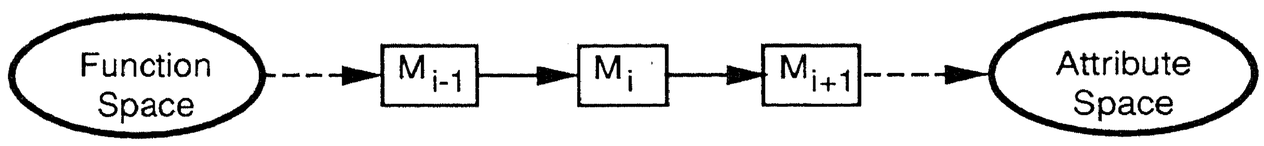

FIGURE 2.2. Stepwise refinement of the meta-model

At a certain stage of the design process the design-object model M_{i-1} is the current, incomplete description of the artifact. In order to get a more detailed description some information is added to the design-object model. After this refinement the design-object model M_{i-1} is transferred to M_i, if it is evaluated and approved. This process is continued, obtaining M_{i+1}, etc., until the

design-object model is a complete and satisfactory description of the artifact. In Fig. 2.3 an example of this process is depicted.

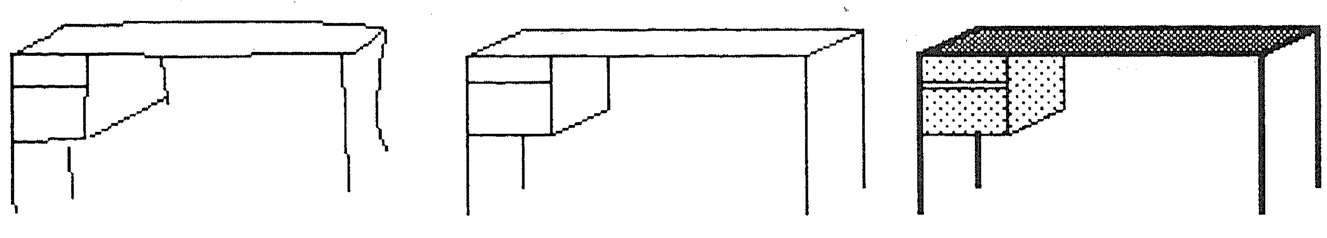

FIGURE 2.3. Example of stepwise refinement

2.2.3. Multiple models. The process as described above deals with the ideal situation in which the stepwise refinement process is a linear process. It can be regarded as a sketch of the design process in retrospect. In practice, it is a process of trial and error, rather than the straightforward process shown in Fig. 2.2. The designer might not be satisfied with a certain version and wants to redo the design from a certain point. Moreover, the designer might not be sure about the direction the design should go at a certain point and wants to model some possibilities in parallel. Therefore, we extent the stepwise refinement model with multiple models presented below.

2.2.3.1. Trial and error. During the design process an achieved subgoal might not be satisfactory and the designer might want to restart from a previous state of the design-object model M_j. In that case, the current design-object model M_j is discarded and the design process is continued from M_{i-1} taking a different direction (see Fig. 2.4).

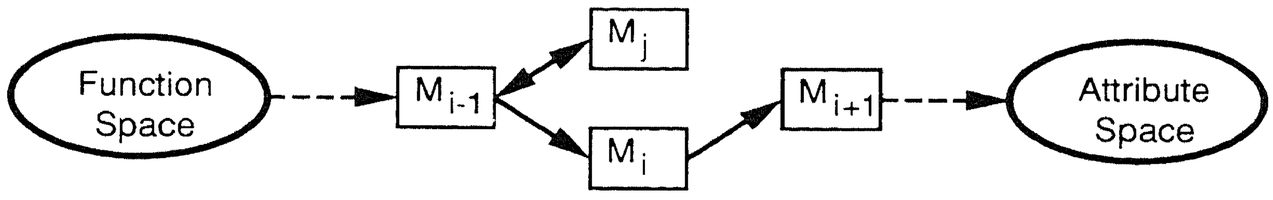

FIGURE 2.4. Backtracking to a previous meta-model

2.2.3.2. Dependent models. From the above we can conclude that at a certain state M_i more than one possible way exists to model the design-object. One of the alternatives is chosen and further refined, eventually resulting in an unsatisfactory solution. Instead of forcing a designer to take a decision at an early stage of the design process, the system must allow him to postpone such a decision and let him model more than one version of the design-object simultaneously. A distinction is made between *dependent* and *independent* models. Dependent models are alternatives which converge into one meta-model. Therefore, dependent models are only a temporary fork in the design process (see Fig. 2.5).

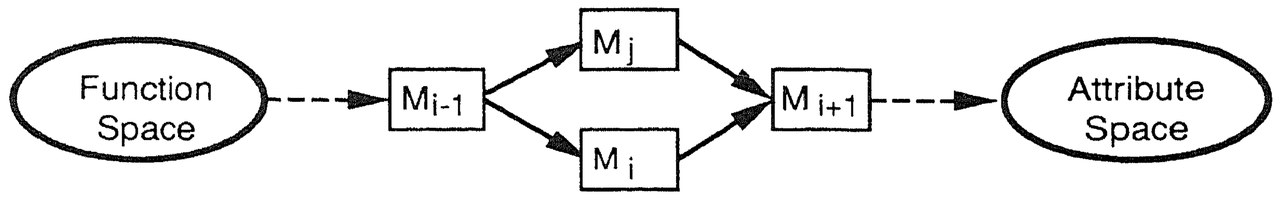

FIGURE 2.5. Dependent models

2.2.3.2. Independent models. Independent models allow the user to create alternatives which actually result in different design solutions. Each of the independent models follows its own path and has nothing in common with other models. An example is depicted in Fig. 2.6. In some cases the designer likes some of the ideas in each independent model. Then, the system has a mechanism allowing the designer to merge several independent models into a single coherent model, eliminating conflicts and unwanted properties. Such a *join* operation is executed in dialogue with the designer. Again, the decision in which way the design process should be directed is totally the responsibility of the designer. The IIICAD system provides the designer with a framework which assists him in his design activities.

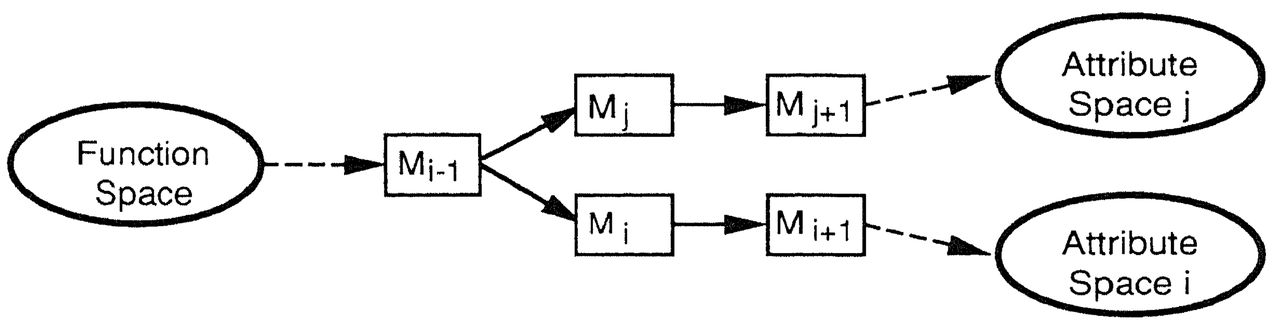

FIGURE 2.6. Independent models result in different solutions

2.3. Meta-model evolution scheme

The general design model as we have introduced it so far, is mainly concerned with the overall design process. We now focus on how to transfer from one meta-model to the next, i.e. how do we perform a design step? Here we introduce the concept of a *world.* Through the creation of a world the designer gives an interpretation of a meta-model in a certain context, i.e. he provides an aspect model. In a world new information about a design-object is obtained. The current meta-model together with the new information form the next meta-model.

2.3.1. World mechanism. The design process is regarded as a continuous manipulation of the design-object model. A certain aspect of the artifact is highlighted and some properties about the design-object are changed. This highlighting is done by worlds; the attention is focussed on a specific part of the design-object model. A world is a part of the design-object model together with a context and some design rules (an interpretation of the design-object model in a particular context). It is used to derive new properties or update

uncertain/unknown properties about the design-object model in order to get a more detailed description. The new properties about the design-object, which are derived in a world, are merged with the original design-object model when the modeling is completed.

A world consists of (a part of) the design-object description together with knowledge about the design object being valid in that particular world. A simple example of such a world is a graphical representation of the design-object. A part of the design-object description is taken and is processed in a world with graphical knowledge in order to generate an image of that part. Such a world needs to know how to map the general design-object description to a description suitable for generating such an image. The designer can now interact with this world and change its contents. After the session the contents of the world being evaluated is mapped back to a new design-object description.

2.3.2. World evolution. This mechanism is called meta-model evolution (see Fig. 2.7). From the current design-object model, M_i, a world, w_i, is taken and some action is performed in it. The world is evaluated after its termination. This evaluation checks for consistency with M_i, i.e. there are no facts that contradict each other and all constraints over the design-object model are still met. The design object model M_i is transferred to M_{i+1}, if the evaluation succeeds. In case of failure, all results of w_i are discarded and the process will restart from M_i. This backtracking is performed in dialogue with the users, so that the next attempt can be more successful.

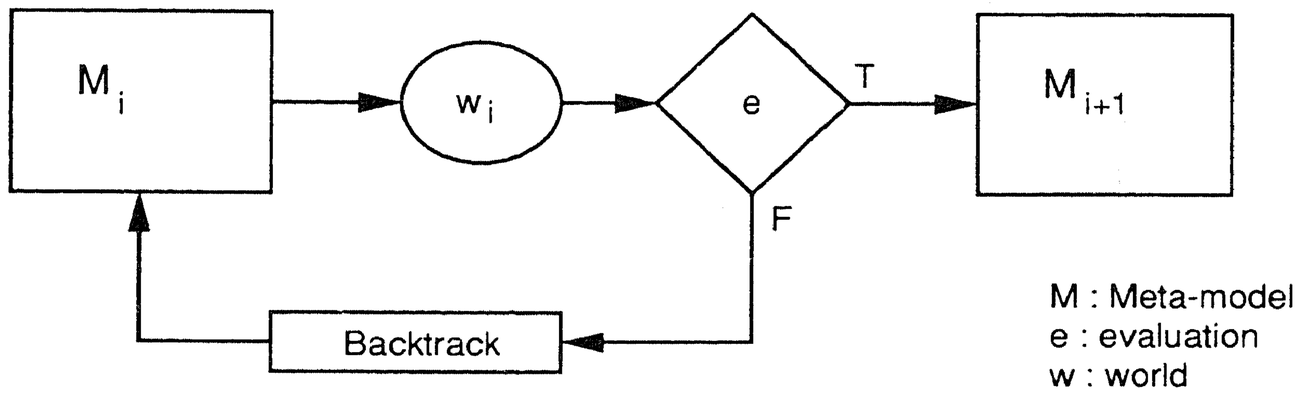

FIGURE 2.7. Meta-model evolution

2.3.3. Multiple worlds. The multi-world mechanism enables the system to create alternative descriptions of the design-object model. This leads to two or more worlds being active at the same time. Concerning this mechanism we distinguish between two types of alternatives: *dependent* and *independent* worlds.

The first option offers the possibility to regard the design-object model from different viewpoints and to model the design-object in various ways. In this case each world is an interpretation of the design-object in a certain context. Each interpretation may change some properties of the design-object model, but changes propagate to all other active worlds. Hence, the worlds are dependent on each other.

The second option allows the designer to model the design-object in different directions by following distinct paths. In this case the worlds refer to different design-object models and hence the worlds are independent. In the following two sub-sections we will expand on both concepts.

2.3.3.1. Evolution of dependent worlds. We call two or more worlds dependent if they refer to the same design-object description. They are concerned with the same set of data, but seen in a different context. They can be regarded as different views on the same model. The user can interact with these worlds separately. The dependency of the worlds comes into being when the worlds are closed. After the closure of dependent worlds these worlds are compared with each other for consistency. Note that worlds can only be compared with each other when they are all closed. If so, the contents of the worlds is merged into a single world containing all the changes. This resulting world is then checked for consistency with the design-object model, resulting in a next design-object model M_{i+1} (see Fig. 2.8). Sometimes there is a need for an intermediate check for consistency. This can be done with an explicit update call from a world. When such a call occurs, all dependent worlds, including the one that called, are checked for consistency. A permanent change in one of the worlds is transferred to the others if appropriate. A change is appropriate if it fits in the context of all dependent worlds.

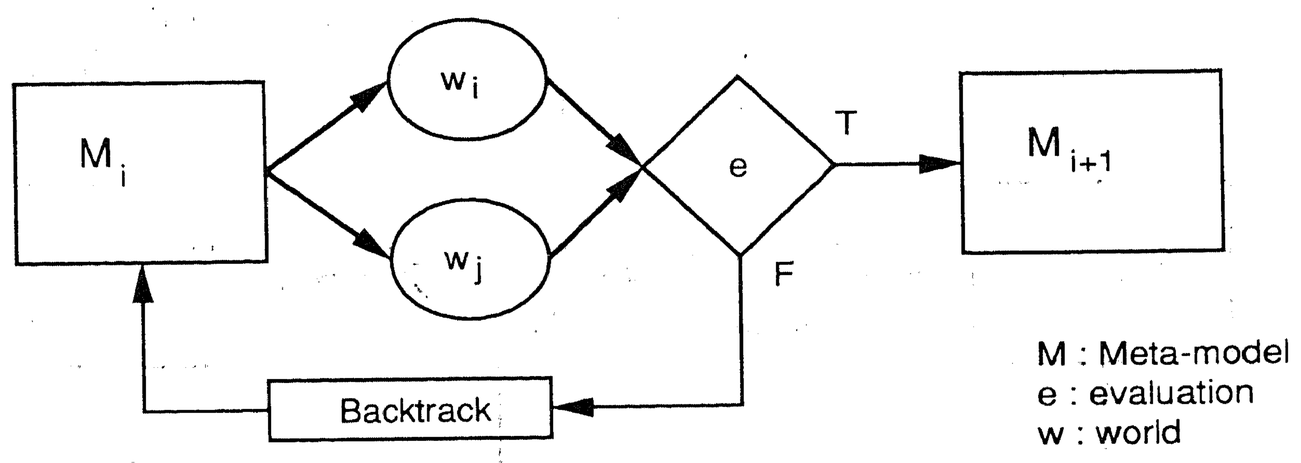

FIGURE 2.8. Dependent worlds

2.3.3.2. Evolution of independent worlds. The independent-world mechanism is used to generate alternative design solutions. Two worlds are called independent if they refer to different meta-models. The designer can then compare those worlds and choose the best design solution. After the closure of independent worlds each world is checked for consistency separately. This may result in multiple distinct design-object models, say M_{i+1} and M_{j+1} (see Fig. 2.9). Each of them can then be further processed as a candidate for the design solution.

The independent-world mechanism differs from the dependent world mechanism in the sense that the independent-world mechanism may lead to more than one design-object model being active at the same time. And hence a lot of extra administration has to be kept since more copies of the design-

object description have to be maintained. We are very well aware of the combinatorial explosion which might be caused by applying the independent-world mechanism. However, we consider it to be the responsibility of the designer to use the independent-world mechanism with care. The decision to continue with multiple design-object models in parallel is always taken by the designer. There is an intelligent user interface as part of the system, which allows the designer to take such a decision in dialogue with the system.

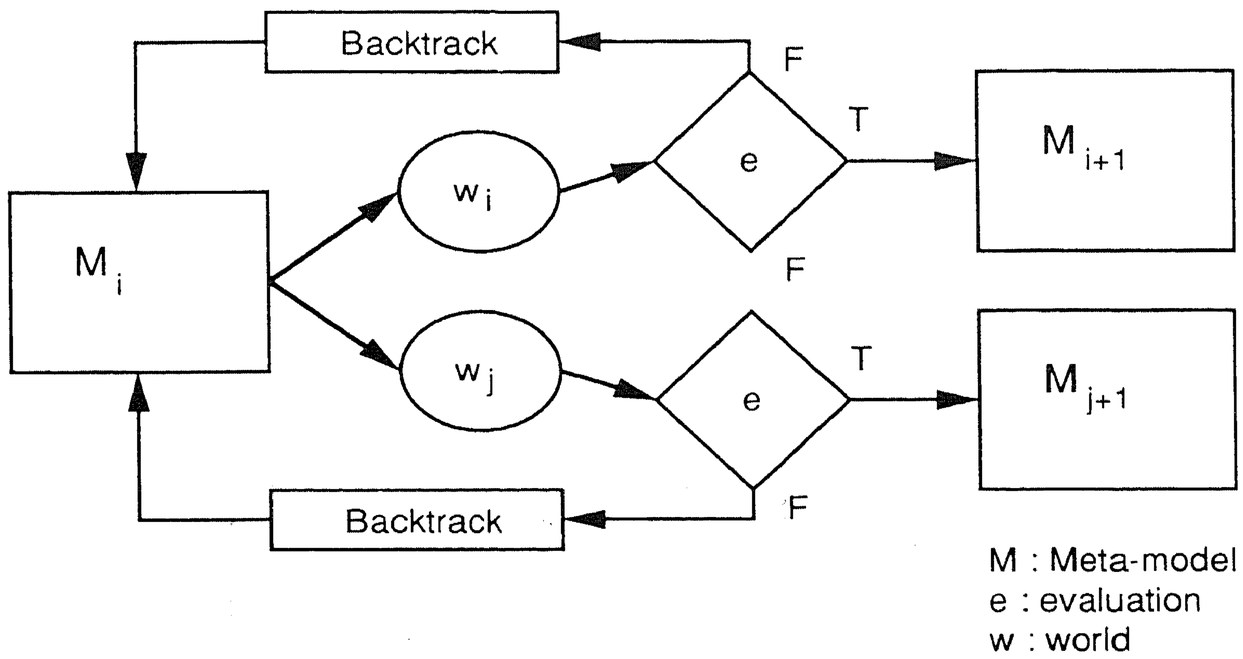

FIGURE 2.9. Independent worlds

3. REQUIREMENTS OF A DESIGN LANGUAGE

This section deals with the architecture of the IIICAD system and the requirements for a programming language to implement it. Both the system architecture and the language requirements rely heavily on the concepts presented in the previous section. In the first sub-section we give a representation of the IIICAD system. In §3.2 we specify the requirements of the design process representation imposed by the general design model. The resulting programming language constructs are presented. In §3.3 language constructs for the specification of the design-object are given. These are derived from the general design model as well. All derived language constructs are prefixed with DM (Design Maxim) [25].

3.1. Representation of the IIICAD system

We want IIICAD to be a system based on expandable ideas and a framework where designers can exercise their faculties at large. We believe that the essential thing in designing is that a designer creates his own design environment and the IIICAD system must give him the freedom to do so [17,20]. An important concept of the IIICAD system is a *scenario.* A design scenario is a piece of design knowledge which is used to perform a design step as mentioned in §2.2.2. It consists of a set of methods and rules which are applied on the design-object model. Scenarios determine the way the design process is directed. The IIICAD system consists of i) a supervisor, which drives the

design scenarios and controls the flow of information between the components of the system and the meta-model [7], ii) a knowledge-base, which contains an objects-base and a facts-base describing the meta-model, iii) an Intelligent User Interface (IUI), which controls the flow of information between the designer and the supervisor, and iv) an Application Program Interface (API), which exchanges information between external application programs and the supervisor. Fig. 3.1 shows the preceding elements of the framework in a block diagram.

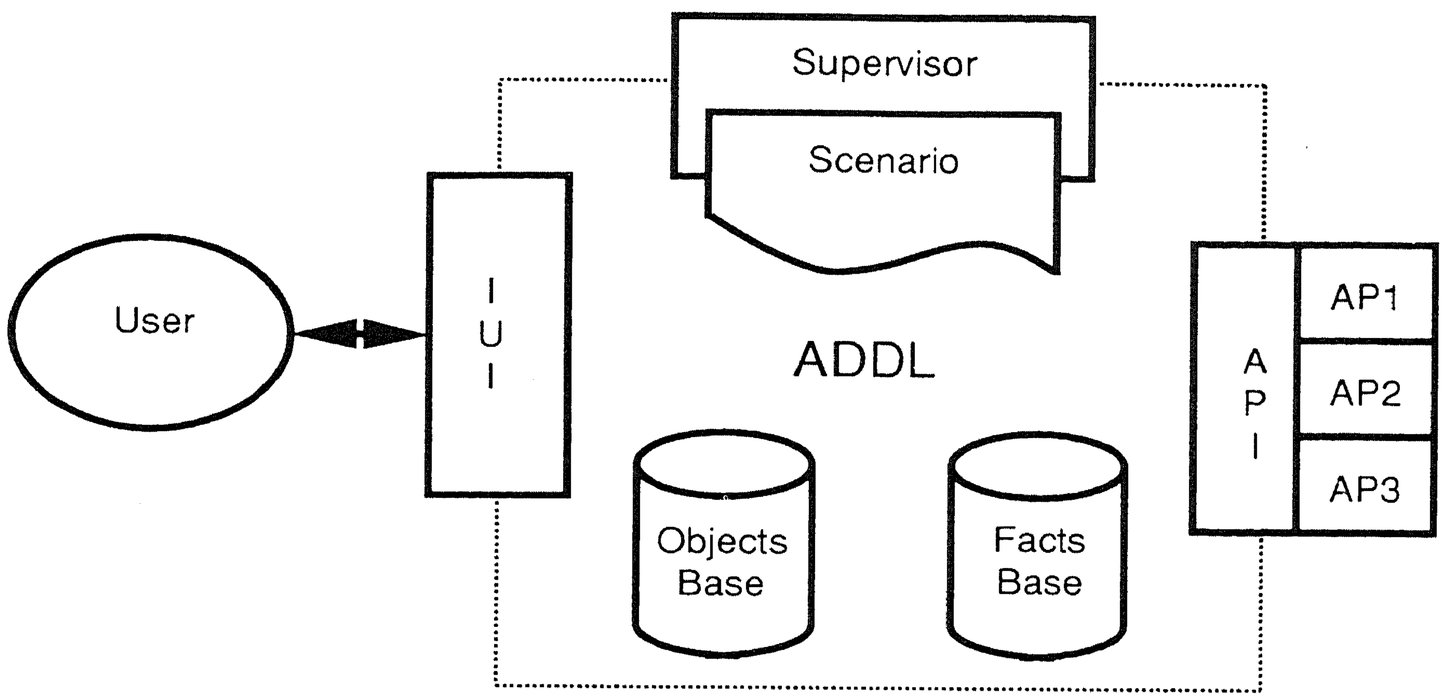

FIGURE 3.1. IIICAD architecture

3.1.1. Supervisor and scenarios. The supervisor is at the core of the IIICAD system and controls all information flow. It adds intelligence to the system by comparing user actions with *scenarios* which describe design procedures and by performing error handling when necessary. A design scenario is a piece of design knowledge employed by the system to perform a design step as described in the previous section [6,10].

DM 1. *ADDL should have a construct to describe status and control information for driving the design process.*

While the supervisor corrects the obvious designer's errors, it does not have the initiative for the design process itself because IIICAD is envisaged to be a designer's apprentice, not an automatic design environment.

3.1.2. Knowledge-base. While scenarios encode the design process representation, the knowledge-base represents the design-object model. A functional separation of the knowledge-base is to call the set of available objects an *objects-base* and the relationships among them a *facts-base.* Both the objects-base and the facts-base can only be accessed via scenarios.

DM 2. *ADDL should have constructs to build and to access both an objects-base and a facts-base.*

3.1.3. Intelligent user interface. The Intelligent User Interface (IUI) is driven by scenarios and maintains the dialogue with the designer. The IUI itself is not written in ADDL (neither are the application programs). The IUI contains knowledge about both the designer and the system. It acts as an intermediary between the designer and the system. It understands and translates the questions and commands of both the designer and the system.

DM 3. *ADDL should have constructs to instruct and query the IUI.*

3.1.4. Application programs. The Application Program Interface (API) secures the mapping between the meta-model description of the design-object and individual models used by application programs. An application program is not necessarily written in ADDL. The API is capable of translating the meta-model into code understandable by a certain modeler. Examples of modelers are: a geometric modeler, a feature modeler, a finite element analyzer, a kinematic modeler, etc.

DM 4. *ADDL should have a construct to translate ADDL code into other code.*

3.2. Specification of the design process

The design process model presented in the previous section imposes its requirements on the design process representation in ADDL. To implement the IIICAD system, which inhabits a meta-model and a design process model based on stepwise refinement, special language constructs are needed.

DM 5. *ADDL should have constructs to describe not only design-objects but also design processes.*

3.2.1. Description of the stepwise nature of the design processes. A designer, given some functional specifications, tries to select a candidate solution, and refines it in a stepwise manner, rather than trying to get the solution directly from the specifications. Therefore, a design process is regarded as an evolutionary process with a series of intermediate descriptions of a design-object.

DM 6. *ADDL should have constructs to describe the stepwise nature of the design process.*

DM 7. *ADDL should have constructs to describe knowledge to detail the design-object model, to check its feasibility, and to control the detailing process.*

3.2.2. Meta-model description. To illustrate a design process, we need to recognize three major components: *entities*, *attributes* of entities, and *relationships* among entities. A design process is thus a collection of small steps to obtain complete information about these three components. An intermediate description is used as a central model for the design solution and we call it *meta-model.*

DM 8. *ADDL should have constructs to describe a meta-model which can easily be changed and extended.*

3.2.3. World mechanism. The designer evaluates the candidate for the design solution to see whether it satisfies the specifications or not. To do so, he derives various kinds of models of the design-object from the meta-model.

DM 9. *ADDL should have constructs to describe knowledge to derive models (for evaluation) from the meta-model and knowledge to evaluate models.*

DM 10. *ADDL should have constructs to allow multiple views of a design object, which are possibly independent but still correlated.*

3.2.4. Default and uncertain facts. During the course of a design process the design-object description changes constantly. To control the stepwise refinement of the design process there is a need to express unknown, uncertain, default, and temporal information about the design-object [14, 16]. A *world* is a derived model from the meta-model. The inside of a world needs to be consistent, but we do allow temporal inconsistencies between worlds. However, when worlds are evaluated in order to derive a new meta-model they need to be consistent.

DM 11. *ADDL should have constructs to describe unknown, uncertain, default and temporal information.*

DM 12. *ADDL should have constructs to let the inconsistency between worlds be represented, but this inconsistency needs to be resolved when transferring to the next meta-model.*

3.3. Specification of the design-object

So far we have given language constructs for the design-object description in terms of design process specification. In this section we introduce ADDL constructs which are concerned with design-object specification. Since design is regarded as a mapping from function space onto attribute space, it requires ADDL to have both attributive and functional representations. There are several issues in representing attributive information. First, an attribute represents the value of a certain property of an entity. Such properties define the internal specification of an object. Second, attributive information refers to

the structure of an entity. The structure of an entity is characterized by a decomposition of sub-structures and by relationships among sub-structures. The structural decomposition defines the external specification of an object.

DM 13. *ADDL should have constructs to describe both the internal and external specification of objects.*

3.3.1. Internal state of objects. Internally an object is specified by its attributes, but an attribute is not necessarily represented by a value. An attribute may either be a value, or a relation between other attributes, or a constraint on other attributes. An object's attributes cannot be accessed directly, but only through functions. Hence, an object is an abstract data type.

DM 14. *ADDL should have constructs to specify attribute values, attribute relations, and constraints on attributes.*

DM 15. *ADDL should have data abstraction, the object's internal state can only be accessed through functions.*

3.3.2. Relations between objects. The specification of a design-object decomposition in ADDL is done through objects and relationships among objects. Each ADDL object represents a part of the entire design-object model. First order predicate logic is used to represent the relationships among the objects. Part-whole relationships, denoting the design-object decomposition, are effected by *has-part* predicates (e.g. the proposition `has-part(table, leg)` expresses that the object `leg` is a part of the object `table`). Predicates are also used to describe the facts which are known about an object.

DM 16. *ADDL should be based on first order predicate logic to specify facts about objects.*

3.3.3. Prototype definition of objects. For the construction of a design-object model a designer employs so-called 'building blocks'. Existing entities are taken from a library of building blocks and modified in such a way that they are suitable to form a new design-object structure. In ADDL such building blocks are called *prototypes* [12]. When the design-object model is created during fundamental design, it is made by copying prototypes from the prototype library. We call this copying the instantiation of an object. The object can further be modified according to the designer's wishes without affecting the original prototype.

DM 17. *ADDL should have a prototype library for the instantiation of objects.*

3.3.4. Delegation to prototypes. A non-trivial design problem easily results in a design-object model which consists of an enormous number of objects. A lot of these objects have the same prototype as their parent. Therefore, objects in ADDL which have the same prototype share the common code. Certain operations (functions) performed on objects are delegated to the object's prototype.

DM 18. *ADDL should have a delegation mechanism to achieve code sharing.*

3.3.5. Hierarchy of prototypes and inheritance. The prototype library contains a hierarchy of prototype definitions. Prototypes are defined in terms of other prototypes. Therefore, if a prototype is a sub-type of another prototype, the former inherits properties from the latter [5]. This allows the system designer to reuse previously defined code and to specialize a certain prototype.

DM 19. *ADDL should have constructs for the definition of a prototype hierarchy, and for an inheritance mechanism.*

4. Conclusions

In this paper we have presented an analysis of the design process, and a formalization according to the analysis. Although the design process consists of several distinguishable stages, it was possible to build a general design process model which covers all stages. From the model we extracted design maxims for the development of ADDL. These design maxims represent the requirements which a programming language for implementing the IIICAD system must fulfill. They serve as a basis for the formal language specifications of ADDL.

A prototype version of ADDL has now been developed, and a first experimental IIICAD is being built (see plate IX). The result of the experiments will be used to evaluate the general design model. It might be possible that either the general design model, or the language specifications should be adapted to the changed needs. For the time being we have succeeded in giving a formalization of the design process and in producing language specifications accordingly.

References

1. V. Akman, P.J.W. ten Hagen, T. Tomiyama (1987). *Design as a Formal, Knowledge Engineering Activity,* CWI Report CS-R8744.
2. V. Akman, P.J.W. ten Hagen, J. Rogier, P.J. Veerkamp (1988). Knowledge Engineering. *Design Knowledge-Based Systems 1 (2),* 67-77.
3. T. Bylander, B. Chandrasekaran (1987). Generic tasks for knowledge-based reasoning: the right level of abstraction for knowledge acquisition. *Int. J. of Man-Machine Studies 26,* 231-244.
4. W.F. Clocksin, C.S. Mellish (1981). *Programming in Prolog,* Springer-Verlag, Berlin.
5. W. Cook (1987). *Self-Referential Models of Inheritance,* Brown University Report.

6. R. DAVIES, J. KING (1977). An overview of production systems. E.W. ELCOCK, D. MICHIE (eds.). *Machine Intelligence 8*, Ellis Horwood Ltd., Chichester, 300-332.
7. J. DE KLEER (1986). Problem solving with the ATMS. *Artificial Intelligence 28*, 197-224.
8. A. GOLDBERG, D. ROBSON (1983). *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Reading, MA.
9. A. GOLDBERG (1984). *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley Reading, MA.
10. P. HAYES (1979). The logic of frames. D. METZING (ed.). *Frame Conceptions and Text Understanding*, Walter de Gruyter and Co., Berlin, 46-61.
11. G.E. HUGHES, M.J. CRESSWELL (1972). *An Introduction to Modal Logic*, Methuen and Co. Ltd., London.
12. H. LIEBERMAN (1986). Using prototypical objects to implement shared behavior in object-oriented languages. *OOPSLA '86, Special Issue of SIGPLAN Notices 21(11).*
13. J.W. LLOYD (1987). *Foundations of Logic Programming*, second, extended edition, Springer-Verlag, Berlin.
14. D. MCDERMOTT (1982). Non-monotonic logic II: Non-monotonic modal theories. *Communications of the ACM 29(1)*, 33-57.
15. M.S. PATERSON, M.N. WEGMAN (1978). Linear unification. *Journal of Computer and System Sciences 16(2)*, 158-167.
16. R. REITER (1980). A logic for default reasoning. *Artificial Intelligence 13*, 81-132.
17. J. ROGIER (1989). The BiCad system: an intelligent product modelling system for architectural design. V. AKMAN, P.J.W. TEN HAGEN, P.J. VEERKAMP (eds.). *Intelligent CAD Systems II — Implementational Issues*, Springer-Verlag, Berlin, 291-310.
18. J. ROGIER, P.J. VEERKAMP, P.J.W. TEN HAGEN (1989). An environment for knowledge representation for design. *Proceedings of Civil Engineering Expert Systems*, Madrid.
19. T. TOMIYAMA, H. YOSHIKAWA (1987). Extended general design theory. *Proceedings of the IFIP WG 5.2 Working Conference on Design Theory for CAD*, North-Holland, Amsterdam, 95-125.
20. T. TOMIYAMA, P.J.W. TEN HAGEN (1987). *The Concept of Intelligent Integrated Interactive CAD Systems*, CWI Report CS-R8717.
21. T. TOMIYAMA (1990). An experience with an integrated data description langauge. P.J.W. TEN HAGEN, P.J. VEERKAMP (eds.). *Intelligent CAD Systems III — Practical Experience and Evaluation*, Springer-Verlag, Berlin.
22. P.J. VEERKAMP, V. AKMAN, P. BERNUS, P.J.W. TEN HAGEN (1989). IDDL: a language for intelligent interactive integrated CAD systems. V. AKMAN, P.J.W. TEN HAGEN, P.J. VEERKAMP (eds.). *Intelligent CAD Systems II — Implementational Issues*, Springer-Verlag, Berlin, 58-74.
23. P.J. VEERKAMP (1989). Multiple worlds in an intelligent CAD system. H. YOSHIKAWA, D. GOSSARD (eds.). *Intelligent CAD, I*, North-Holland, Amsterdam, 77-88.

24. P.J. VEERKAMP, R.S.S. PIETERS KWIERS, P.J.W. TEN HAGEN (1990). Design process representation in ADDL. *Intelligent CAD Systems III — Practical Experience and Evaluation*, Springer-Verlag, Berlin.
25. B. VETH (1988). An integrated data description language for coding design knowledge. P.J.W. TEN HAGEN, T. TOMIYAMA (eds.). *Intelligent CAD Systems I — Theoretical and Methodological Aspects*, Springer-Verlag, Berlin, 295-313.