VRIJE UNIVERSITEIT

On the Development of an Artifact and Design Description Language

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
dr. C. Datema
hoogleraar aan de faculteit der letteren,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der wiskunde en informatica
op dinsdag 15 september 1992 te 13.30 uur
in het hoofdgebouw van de universiteit, De Boelelaan 1105

door

Paulus Jan Veerkamp

geboren te Amsterdam

CWI, Amsterdam
1992

Promotor:	prof.dr. J. Treur
Copromotor:	F. Arbab Ph.D.
Referenten:	prof.dr.ir. N.J.I. Mars
	prof.dr. R.P. van de Riet

Voor Sabine
en mijn ouders

Table of Contents

Acknowledgements

The work described in this dissertation is the result of research I performed while employed at the CWI in Amsterdam. Many people have aided this effort amongst whom I would like to mention a few especially.

First of all, I would like to thank my promotor Jan Treur and my copromotor Farhad Arbab for their supervision. Although Jan Treur became my promotor only during the last phase of my research, he convinced me to change the focus of my research and therefore the overall thrust of the thesis advanced in this dissertation. The many stimulating discussions and useful suggestions have contributed a great deal to the final version of ADDL.

I am very grateful for the continual support of Paul ten Hagen, head of the Department of Interactive Systems at CWI. He provided me with a pleasant research environment and he gave me great latitude in choosing the scope of my research. I owe a great deal to Tetsuo Tomiyama, Associate Professor at the University of Tokyo. He not only started the IIICAD project and stimulated me to become researcher, he also became a good friend with whom I shared a lot of time in pubs and restaurants. I would especially like to thank 'Tomi' for allowing me to be a visiting researcher to his laboratory in Japan. I would also like to thank Varol Akman who succeeded Professor Tomiyama as project leader. We had many fruitful discussions about ADDL and qualitative physics.

My roommates Jan Rogier and Daan Otten provided a great deal of help during the past five years. They created a pleasant and inspiring atmosphere that led to many stimulating discussions.

My thanks to you all, and to the other members of the department of Interactive Systems at CWI. Amongst whom I thank Edwin Blake for improving my

English and Anco Smit for improving my Dutch. I am also grateful to Gusz Eiben from the Vrije Universiteit Amsterdam for proofreading chapter 5 till Chapter 7.

My parents were always my greatest supporters in every way. They created the possibility and motivated me to study from the very beginning and they taught me perseverance. But most of all I would like to thank my wife Sabine who insisted in reading every chapter of my dissertation even when she couldn't follow all of it. Every time I felt dispirited, she encouraged me to continue and challenged me to finish my work.

1

Introduction

1.1 Subject of the dissertation

Although CAD (Computer Aided Design) systems have become an essential tool for designers in many disciplines, it is also recognized that they are still inflexible and too task specific. Supporting a designer in performing the entire design task is the purpose of a CAD system. Routine tasks are delegated to the system. However, the majority of existing CAD systems are merely sophisticated workbenches for engineering drawings. As the application domain becomes more complex, designing becomes unmanageable with only this type of support. Therefore, designers need a more sophisticated system that can assist them 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 that ICAD systems require. The lack thereof necessitated the development of a special purpose programming language: ADDL (Artifact and Design Description Language). This dissertation deals exclusively with the design and implementation of ADDL.

The work described in this dissertation was carried out at the Centre for Mathematics and Computer Science (CWI) as part of the IICAD (Intelligent Interactive Integrated CAD) project. The project started in 1987 with 75% funding from NFI (Nationale Faciliteit Informatica, project NF 51/62-514: 1986–1992) plus 5% funding from TNO (Netherlands Organization for Applied Scientific Research). Part of the research activity within the project was carried out by the Artificial Intelligence group of the Department of Mathematics and Computer Science at the Free University in Amsterdam. Their effort is complementary to the work at CWI. It

is focussed on control of the design process and handling of incomplete information.

The goal of the project is to study the issues involved in building CAD systems that:

- contain more complete knowledge about the activity of design, as well as domain dependent design knowledge;
- are better integrated to provide a consistent set of functionalities which can be combined to cover a broad range of activities involved in a production cycle;
- have high-quality, high functionality, intelligent user interfaces.

The emphasis is in particular on the use of AI techniques.

The project started with the development of ADDL. The language has special dedicated features to encode existing and newly acquired knowledge about the design object, about the design process and their relations. The encoding and treatment of design knowledge is studied in the context of geometric modeling, object-oriented databases, user interfaces and geometric reasoning.

1.2 Programming languages for 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 equipped with a database where product data could be stored and retrieved. However, during several stages of the design process a product's specification needed to be verified. This could be achieved by external modules, such as a FEM (Finite Element Method) analysis module, or a cost analysis module, etc. These were separate tools, forcing the CAD data to be transferred from one system to another, and back. A future CAD system needs to be an *integrated* system which contains a central design object model, and which has several application modules connected with it, allowing the designer to analyze his product in several ways. Such a system employs a uniform language which is used by all subparts.

An ICAD system makes high demands on the programming language that is used for its implementation. Such a language must have the following characteristics:

- It must provide a means to represent a design object as a collection of entities. Each entity represents a structural component of the design object. Entities must contain a set of attributes that represent dimensions of a component. They must respond to a set of operations that can be applied to their attributes.
- It must allow for representing relations among entities and properties of entities in a flexible way. The complete set of relations and properties describe the function and behaviour of the design object. It is called the

qualitative model of the design object. The qualitative model must also describe a decomposition of the design object into parts.

- It must have a mechanism to represent knowledge of how to create, evaluate and modify the above design object model. This knowledge is also called *object* knowledge because it represents the designer's knowledge about the objects. Designers use different kinds of expertise during various stages of the design process. The object knowledge must therefore be encoded in a modular fashion.
- It must also have a mechanism to represent knowledge of when and where to apply the object knowledge. This knowledge is called *process* knowledge because it represents knowledge about the design process. It controls the application of the object knowledge. The process knowledge must also be encoded in a modular fashion.
- 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, i.e., the qualitative model.
- It must offer the means for high level interaction with a designer i.e., good human computer interaction facilities.

A language that is based on both objects and logic, forms a firm basis for implementing an ICAD system. Furthermore, if the language has a strict separation between object and process knowledge, it is easier to use, debug and modify. The following sections shortly introduce object-oriented programming languages, logical programming languages and reflective architectures.

1.2.1 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 of representing properties of a data structure and operations allowed on that data structure in a single location. A program obtains information from an object by means of an answer to a message sent to that object. Message passing may also be used to give a task to an object. Objects that have a common behaviour and related properties are grouped together in a uniform description, viz. a class. Classes are defined in terms 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 [Wegner, 1990; Rumbaugh *et al.*, 1991].

The three most popular object-oriented languages are Smalltalk-80 [Goldberg and Robson, 1983], Eiffel [Meyer, 1988], and C++ [Stroustrup, 1986]. They range from a purely object-oriented language such as Smalltalk towards an object-

oriented extension of an existing language (C) such as C++. Smalltalk was the first popular object-oriented language. The Simula language served as a basis for Smalltalk when it was developed at the Xerox Palo Alto Research Center. The Smalltalk-80 System is not only a language but also a programming environment that includes the functionality usually attributed to a computer operating system: a file system, display handling, text and picture editing, a debugger, processor scheduling, compilation and decompilation. Although the implementation of ADDL is done in Smalltalk, the implementation does not depend on it. ADDL can also be ported onto Eiffel, C++ or another programming language. The object-oriented part of ADDL uses existing Smalltalk constructs as much as possible. For instance, ADDL objects use Smalltalk Class definitions and for message passing in ADDL the Smalltalk message passing mechanism is used.

1.2.2 Logic programming

The fundamental idea of logic programming is that first order logic can be used as a programming language. The need to use logic as a programming language stems from natural language processing. It was argued that logic is a precise and formal language as opposed to natural language, which is imprecise and ambiguous. Logic was thus an appropriate means for representing natural language in a computer. The popularity of logic programming became more widespread after the success of the first logic programming language Prolog (*Pro*gramming in *log*ic [Clocksin and Mellish, 1981]). It is now one of the most popular programming languages used in Artificial Intelligence applications. The advantage of a logic programming language is the combination of having a *declarative* semantics as well as a *procedural* interpretation. The former gives the meaning of a program is while the latter is concerned with how such a meaning is obtained.

Logic consists of *propositions* and *relations* among propositions [Clark and Tärnlund, 1982; Lloyd, 1987; Apt, 1990]. Propositions of *sorted first order logic* consist of typed predicate symbols and terms. Furthermore, there is an *inference engine* that can infer propositions from others, and which can validate propositions. Most inference engines of logic programming systems are based on resolution theorem proving [Paterson and Wegman, 1978]. The basic idea of logic programming can best be described by an example. Suppose I want to express that 'Socrates is a man'. This can be done by the proposition

```
man(socrates).
```

Furthermore, I have the knowledge that all men are mortal. This can be expressed by the implication

```
man(X) → mortal(X)
```

where `X` is variable that ranges over all constants in the language. In other words, *if* `X` is a man *then* he is mortal. Using these two expressions an inference engine

can infer the proposition that socrates is mortal, i.e.,

```
mortal(socrates).
```

1.2.3 Reflective architectures

A common property of *reflective* or *meta-level* architectures is that they consist of two levels. The object-level at which reasoning is performed about the application domain and the meta-level at which the object-level reasoning is controlled. The main advantage of such architectures is the separation between what the system knows (the object-level) and how this knowledge is applied (the meta-level). A system with explicitly and separately represented process knowledge is more modular, and thus easier to develop, debug and modify (see [van Harmelen, 1989] pp. 14).

In ADDL both the object-level and the meta-level knowledge-base are partitioned into *scenarios*. A scenario consists of a set of rules that are applicable to an information state. An information state consists of a description of either a design object or a design process, depending on the level of reasoning. The application of the rules of an object-level scenario results in an extended *object information state*. An object information state embodies a (partial) model of the design object. It consists of the entities that describe parts of the design object decomposition and it consists of relationships among these entities. Therefore, the application of an object-level scenario causes an expansion of the entities and relations in the object information state. Object-level scenarios represent knowledge about a certain aspect of design. The reasoning involved is a forwardly directed activity. Initially the design object model consists of a minimal description which is gradually extended as the design proceeds. Both the meta-level and the object-level language of the architectures presented in this paper are based on a subset of sorted first order logic. Scenarios consist of a set of rules that are equivalent to logical implications. The object-level language is based on three-valued logic. It describes the world as partial models in which every statement is either *true* or *false* or *unknown*. The meta-level language only uses the classical truth values and is based on the closed world assumption [Reiter, 1978].

1.3 Outline of the dissertation

The ordering of the chapters of this dissertation gives a chronological reflection of the development of ADDL. Except for Chapter 3 which should actually be placed between Chapter 8 and Chapter 9. I moved it forward to provide the reader with an example. The following sections briefly summarize the chapters of the dissertation.

1.3.1 Summary of chapter 2

Chapter 2 starts with the presentation of a *prescriptive* model of the design process, which explains how design should be performed. It discusses the representational issues that a designer meets during the design process. It also serves as a tutorial for readers unacquainted with design. The next topic of that chapter is a *descriptive* design process model (DPM) that describes design as a cognitive process. The model is based on the General Design Theory of Tomiyama and Yoshikawa [Tomiyama and Yoshikawa, 1987]. It models the design process 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, a designer starts with a functional specification of a design object and ends with a manufacturable description [Tomiyama and Yoshikawa, 1987].

The basic ideas behind DPM 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 refined in a stepwise manner until the solution is reached. The design process is thus regarded as an evolutionary process that transfers the object model through a sequence of design steps from one state to another. DPM is certainly not the one and only descriptive model of the design process As in cognitive psychology, many approaches by different researchers have led to various models. The last part of this chapter presents some alternative models of the design process and compares them with DPM.

1.3.2 Summary of chapter 3

Chapter 3 introduces an example design problem. It shows how a designer solves the problem in accordance with DPM. The problem deals with a bounded linear motion mechanism. The chapter gives apart from a general solution to the problem, also a worked out example of a real design using an experimental ICAD system.

1.3.3 Summary of chapter 4

Chapter 4 lists a set of of criteria that must be met by a design environment consisting of the IIICAD system implemented in ADDL (Artifact and Design Description Language). I used DPM as the initial inspiration for the derivation of these criteria. They are presented in the form of twenty-nine *design maxims*. The development of ADDL was done by collecting these design maxims and deriving language constructs from them. The design maxims have led to the specifications given in the subsequent three chapters. Note however, that the specifications are not limited to DPM. The alternative models of the design process are also covered by ADDL.

1.3.4 Summary of chapter 5

Chapter 5 represents the object-oriented aspect of ADDL. It introduces two categories of ADDL objects, namely *primitive* and *composite* objects. Four different types of objects are distinguished within the category of primitive objects. These types are *number*, *symbol*, *string*, and *array*. Each type defines its own set of operations by which an object of the type can respond to a function call. A primitive object is represented by its value. The operations and the value together embody the behaviour of a primitive object. Whereas the set of primitive object types is limited, the set of composite object types can be extended by the application programmer. The type definition of composite objects does not only give a set of operations but also a set of named slots that is used as a set of attribute values. Thus, composite objects have an internal structure that consists of these attribute values. Attributes are properties of objects such as length, height, colour etc. Both the attribute values and the operations are accessed through functions. The instantiation and modification of both primitive and composite objects is accomplished through *object-level expressions*.

Each composite object and its set of associated attribute values is stored in an *object-base*. Properties of objects that cannot be described by attribute values are stored as a set of *literal facts* in a *fact-base*. Unary literal facts describe a specific 'quality' of an object. Literals with an arity greater than one describe a relationship among two or more objects. The entire set of literal facts represents a qualitative model of the current state of the design object. The fact-base is also manipulated by object-level expressions. The joint states of the object-base and the fact-base are called an *object information state*.

1.3.5 Summary of chapter 6

Chapter 6 describes the class of object-level languages that are part of ADDL. Since ADDL has a meta-level architecture, it consists of both a meta-level language and an object-level language. The purpose of the object-level language is to represent knowledge concerning the design object. This knowledge is distributed over (object-level) scenarios. Each scenario is an autonomous module that knows how to perform a design step. It contains a set of rules that evaluate the object information state and add information to that state. Scenarios aim at satisfying goals stated at the meta-level. A scenario remains active as long as its intended goal has not (yet) been satisfied. After termination of a scenario, control is given back to the meta-level interpreter. It merges the information derived from the scenario's knowledge (i.e., its rules) with the object information state resulting in a new state. The application of an object-level scenario is equivalent to a design step as introduced in Chapter 2.

1.3.6 Summary of chapter 7

Chapter 7 presents the meta-level language. Meta-level scenarios represent knowledge about the design process. They state design goals that are either satisfied by object-level scenarios or meta-level scenarios. In other words, the meta-level interpreter controls the application of either kind of scenarios. Other conclusions derived from meta-level scenarios are concerned with the process state of the design object model represented by process parameters. These conclusions are stored in a *process information state*. Thus, the process information state consists of i) asserted design goals, ii) satisfied design goals, and iii) parameters concerned with the process state of the design object model. Meta-level scenarios can also evaluate the object information state through a reflection principle. Upward reflection maps object-level information onto the meta-level information state. Using this mechanism, the meta-level interpreter can reason about the truth value of information in the object information state.

1.3.7 Summary of chapter 8

Chapter 8 deals with the implementational aspects of ADDL. It consecutively presents the ADDL compiler, its interpreter and a run-time environment. The compiler consists of a lexical analyzer, a parser and a code generator. The parser reads a token stream scanned by the lexical analyzer. The parser is written with the use of Yacc. The code generator traverses the parse tree and creates a Smalltalk method for each ADDL rule. The interpreter executes these methods when a scenario is activated. Furthermore, it registers all information obtained by the execution. Both the compiler and the interpreter consist of a meta-level and an object-level part taking care of meta-level and object-level scenarios respectively. The meta-level interpreter maintains control of the design process. It dictates the object-level interpreter.

The run-time environment is on the one hand the programmer's workbench where scenarios and prototype definitions can be edited. Its user interface is similar to the Smalltalk-80 programming environment. On the other hand, it is an experimental CAD system where scenarios can be executed. Two example design systems have been implemented in the experimental system. Chapter 9 presents one of these, the other is presented in [Veerkamp and ten Hagen, 1991; Treur and Veerkamp, 1992].

1.3.8 Summary of chapter 9

Chapter 9 describes an example design system that has been implemented for the class of design problems introduced in Chapter 3. The system consists of five meta-level scenarios and nine object-level scenarios. The chapter presents and discusses each of them. Appendix 3 gives a signature for each scenario. It contains the types, constant symbols, function symbols and predicate symbols being used.

The chapter also gives a trace of the process information state in simulating a run of the example design system. It uses the specifications of the working-out of a real design presented in Chapter 3.

1.3.9 Summary of chapter 10

Chapter 10 gives a discussion on the topics of this dissertation. It evaluates the results that have been obtained. It gives some topics for future research and it compares ADDL with competing languages and systems.

2

Design Process Models

2.1 Introduction

Prescriptive and descriptive models of the design process form the central themes of this chapter. A prescriptive model gives directions on how design ought to be done while a descriptive model is a cognitive model that describes how people solve design problems. Along with the prescriptive model of the design process I survey the representational issues that arise during the evolution of a design object. The presentation and comparison of some descriptive models of the design process form the last part of this chapter.

It is recognized that designing is a 'mysterious' activity that is currently only done by human designers. The expansion of computers in the society and their growing utilization in the industrial process engendered the need to develop computerized design systems as well. Recent research into CAD systems resulted in tools for supporting a designer to generate a representation of an artifact, e.g., a drawing. As a consequence, the issues concerning the representation of an artifact are fairly well understood and agreed upon. This chapter starts with a discussion in §2.2 on the several stages of the design process based on a literature study. It is also referred to as a *prescriptive* model because it describes how design should be done [Finger and Dixon, 1989]. It serves a dual purpose; (i) it gives the requirements that are put upon the representation of design objects and (ii) it provides readers of no or very little knowledge of design with a short introduction to the design process. This section has been inspired by the discussion session on "design object representation" at the third IFIP TC 5/WG 5.2 workshop on intelligent CAD [Arbab, 1991], which I attended and to which I contributed.

The second part of this chapter gives a formalization of the design process in terms of a *descriptive* model. I present the model in § 2.3 and compare it with other models in § 2.4. A descriptive model describes how a designer performs a design task. The model, called DPM (Design Process Model), is used as an inspiration for the development of ADDL. I agree that it is rather pretentious and even dangerous to give a descriptive model of a process that is not yet fully understood. Nevertheless, since I am aiming at a system that (only) assists the designer during the design instead of performing the task itself, it suffices to build a model of the external behaviour of a designer. Therefore, developing a formal framework for an ICAD system that solves a design problem in dialogue with a designer, is the main issue of this dissertation. I am thus interested in *how* designers performs their job but not *why* it is done in that way.

Finger and Dixon (1989) have summarized and reviewed research in mechanical engineering design theory and methodology. They state:

> "In a mature field, the research community will share a common view of what are appropriate research methodologies, what are the difficult unanswered questions, and what constitutes high quality research. In the emerging field of design research, no such consensus exists."

Due to this lack of consensus it is neither possible nor desirable to present a single model of the design process. Many design process models have been developed; each having its advantages and disadvantages. Therefore, § 2.4 compares DPM with other competitive but not orthogonal descriptive models. It turns out that it is possible to develop a framework in which each of the descriptive models fits [Waldron, 1991].

2.2 A prescriptive model of the design process

Designing is an activity that is based on both knowledge and experience. Examination of designers at work shows that each individual designer tackles a design problem in a different way. If one asks designers why they actually design in the way they do, one gets an unsatisfactory answer. They do not really know [Rogier, Veerkamp, and ten Hagen, 1989]. However, extensive research on designing has been carried out by German researchers resulting in a number of text books of which [Hubka, 1987] and [Pahl and Beitz, 1988] have been translated to English. Both books include an extensive list of German references. They both distinguish among three successive stages in the design process[1], i.e., *conceptual* design, *fundamental* design, and *detailed* design (see Fig. 2.1). Traditionally, these stages are particularly distinguished in mechanical engineering due to differences

[1] One cannot assume that design has always been viewed and represented in this w
nor can one assume that it will always remain in this from. The development o
design tools may influence the way design is performed.

in the way drawings are used. These stages are also called *functional, basic* and *detail* design respectively [Arbab, 1991].

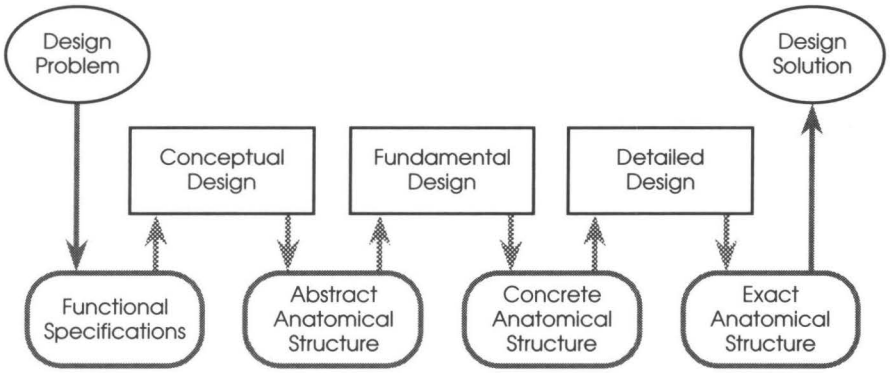


Fig. 2.1 The three successive stages passed through during the design process.

These design stages have in common that they operate on a design object representation. But each stage has its own demands on representational issues. The design object representations are depicted in the rounded boxes of Fig. 2.1. There is an implied need for a design object model that allows for the representation of properties characteristic for each of the three stages. What kind of characteristics these are is shown in §2.2.1 through §2.2.3. There is no clear borderline between these three stages. A designer gradually moves from one design stage to another without noticing. Obviously, this leads to the conclusion that there should be a single design process model that captures all three stages of design. Moreover, there should also be a single design object model. In other words, in each of the different phases of design a uniform design object representation is manipulated. A design process model is presented in §2.3. The sections below treat each of the design stages separately.

2.2.1 Conceptual design

Design starts with a need, the statement of a design problem. A design problem does not necessarily have to be an entirely new problem; it might have been solved by previous designs. Design is thus often a matter of improving existing designs. The necessity for these improvements may be caused by several reasons, e.g., changed requirements, a disappointing performance, excessive costs, etc. In another case, the design problem may be of a totaly new kind. The former is called *routine* design, the latter *creative* design. The way both categories of design problems are solved is basically the same. The difference lies in the amount of time

spend in one of the three stages. However, creative design demands more from an ICAD system than routine design, since for routine design existing design schemes are known and can be applied. For creative design new design schemes have to be developed.

In the early stage of design the problem is analyzed by the human designer, and the output of this analysis consists, amongst others, of:

- a precise statement of the problem in terms of function and behaviour,
- limitations placed upon the resulting product, e.g., spatial requirements, cost constraints, international standards etc.
- the measure of quality that should be worked to.

The last item is in most cases the bottleneck; how to produce reasonable quality at the lowest possible costs. The analysis of the design problem is carried out without use of an ICAD system. The result of this phase is called a *functional specification*.

The functional specification of a design problem is used as a starting point for the design process modeled by the IIICAD system. A designer supplies a functional specification to the system, and the system translates it to an initial design object model. The statement of the design problem is thus transformed into an *abstract anatomical structure*, which describes the problem in terms of broad solutions. The broad solutions are represented in the form of design *schemes*. The transformation from a design problem specification to an abstract anatomical structure is accomplished through an interaction between the designer and the design system. The schemes specify the kind of dialogue being held, in other words, they denote the design process knowledge. In this phase the most important decisions are taken and it makes the greatest demands on the designer. It is the designer who decides what kind of design scheme is executed.

The abstract anatomical structure is the initial representation of the design object in the IIICAD system. It is a rough solution to the design problem, which describes the behaviour of each major function, and gives the spatial and structural requirements of the major components. It allows for inclusion of subparts and attributes to be specified later. It is important that feasibility of a solution can be checked at a state as early as possible.

2.2.2 Fundamental design

During the course of fundamental design the abstract anatomical structure is converted to a more concrete anatomical structure. It is a description of something one can actually make. A rough decomposition of the artifact is created and the principal shape of the design object is fixed. A primary solution for the major components of the decomposition is chosen. To find such a partial solution a designer uses experience obtained during previous design sessions. It is often the

case that a certain part of the design object has been designed before, or that it is similar to a part which has been designed before. For this purpose the designer possesses a collection of *prototype* solutions which are applied as standard components for parts of a new design [Gero, 1990; Rogier, 1991].

Such a collection of possible design solutions is part of the IIICAD system. A library of standard components allows a designer to use a certain component as a prototype for a part of a new design. The prototype may be modified according to a designer's wishes. If a suitable prototype is absent, a designer is allowed to choose a prototype which resembles closest to the desired component and to change it completely. The design knowledge necessary for choosing the right prototypes, and manipulating them, is also denoted by means of design schemes.

The concrete anatomical structure which is the result of the fundamental design phase is represented by a decomposition containing empty and fuzzy parts. The model is by far not complete and most of the parts are still absent. Dimensions and tolerances have not yet been determined. The relationships among the several parts are not yet known. In a certain situation, however, the system may need a value for a certain attribute, or might want to know about a certain relationship (e.g., to generate a geometric representation). In that case the system assumes default values for those attributes, and uses uncertain facts for those relationships. Hence, the behaviour of the model is simulated by assuming default behaviour for the parts which are uncertain or unknown.

This phase of the design process consists of several steps. Initially, there is hardly any structure in the model, most of the parts are unknown. Then during several steps the model is gradually structured. When the last phase of fundamental design is reached, the entire structure of the design object model is determined. Now, the only thing to be done is refining the model and working out the details. This happens in the next phase of the design process.

2.2.3 Detailed design

Prior to this phase of the design process, a complete structure of the design object model has been defined. The major parts of the design object have been determined and a decomposed model represents them. The purpose of detailed design is producing an exact description of the anatomical structure. Dimensions and tolerances are set, all constraints are satisfied and all parts are integrated into one coherent model. Therefore, all attributes of the model receive a definite value, and all relations among the various parts are defined. The model is verified with the initial specifications, and it is evaluated to check whether the requirements are met.

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. Allowing a designer to concentrate on a certain part of the design object is the main issue at this phase. The part is highlighted and it is modeled in its own context, i.e. special conditions which only apply for this particular part are now valid. After being modeled such a part is replaced in the whole, and checked whether it still fits. The IIICAD system allows the designer to generate such models on specific parts of the design object model. It is done by certain design schemes.

During this stage of design, a designer consults various experts, to obtain some information on various aspects of the design. These experts perform domain specific calculations, or they evaluate the design object model in a certain context. They add new information to the design object model. Sometimes a designer asks several experts for information at the same time. Each expert adds data from its own field of expertise to the design object description. Some experts may add contradictory information to the design object description, since they have different background knowledge. It is the designer's task to maintain the consistency of the design object model.

2.3 A descriptive model of the design process

This section gives a *descriptive* design process model (DPM) applicable to the three stages presented in the previous section. I used DPM as a source of inspiration for the ADDL specifications and the IIICAD architecture [Veerkamp, 1989]. They serve as a general framework in which DPM can be implemented. The framework is not restricted to DPM only. The models presented in §2.4 also fit in the framework. Knowledge about the three stages of the design process and about the design object can be embedded in the framework. Thus, the system is always informed about the current state of both the design process and the design object [Akman *et al.*, 1988]. However, a designer decides how to perform the design process and the IIICAD system is an intelligent aid that support designers in achieving their goal by supplying the right tools for each specific stage of the design process.

The General Design Theory of [Tomiyama and Yoshikawa, 1987] serves as a basis for DPM. It is based on axiomatic set theory. It describes design as a mapping from the function space where the design object specifications are described in terms of functions, onto the attribute space where the design solutions are described in terms of attributes. Roughly speaking, one starts with a functional specification of the design object and ends up with a manufacturable description. The overall outlook of DPM is depicted in Fig. 2.2.

The basic ideas behind the DPM are as follows:

- From the given functional specifications a candidate for the design solution is selected and refined in a stepwise manner until a complete solution is obtained, rather than by trying to get the solution directly from the specifications. The latter is not possible in a non-trivial design problem, since

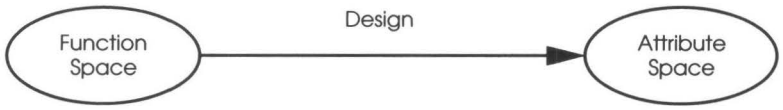


Fig. 2.2 Basic model of the design process.

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. The number of attributes that received a value grows as the design process proceeds and a growing number of functional specifications is met.
- 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.

I call those interpretations of the design object model *contexts* and they can be regarded as interpretations of the design object observed from certain points of view. Contexts allow a designer to model the current state of the design object in a certain environment, i.e. they represent an aspect model. More information about the design object is obtained through these contexts and hence the number of attributes grows. Contexts are created by means of *scenarios*, which contain design knowledge and data necessary to build an aspect model. Scenarios perform the reasoning about a context and they lead the dialogue with the designer.

According to DPM, a design object model is refined in a stepwise manner. An intermediate state of both the design process and the design object is called a *meta-model*. A meta-model consists of the following three components:

1. *Entities* that describe parts of the design object decomposition. Each entity can in turn be composed of other entities. Entities may have one or more attributes that denote its quantitative properties.
2. *Relationships* among entities that represent their qualitative properties. They describe the anatomical structures introduced in § 2.2.
3. *Process parameters* of relationships that give the process information state of the structures. In other words, they represent the design process state of (a part of) the design object.

The stepwise refinement process shown in Fig. 2.3 behaves as follows: at a certain stage of the design process the meta-model M_{i-1} contains the current, incomplete description of the design object. A scenario interprets this state in a context in

order to get a more detailed description. Through this scenario new information about the design object is obtained. After this refinement the new information from the aspect model is merged with the meta-model M_{i-1}. If the merge is successful, i.e., the new information is consistent with the current M_{i-1}, then the result of the merge is a new state of the meta-model, M_i. The move from one meta-model to another is called a *state transition*. This process is continued, obtaining M_{i+1}, etc., until the design object model is a complete and satisfactory description of the desired artifact. I stress that here 'complete' has the meaning of satisfying all initial requirements. As a matter of fact a design can never be complete, there is always something which can be improved, or which can be made cheaper. In this context complete has a rather subjective meaning.

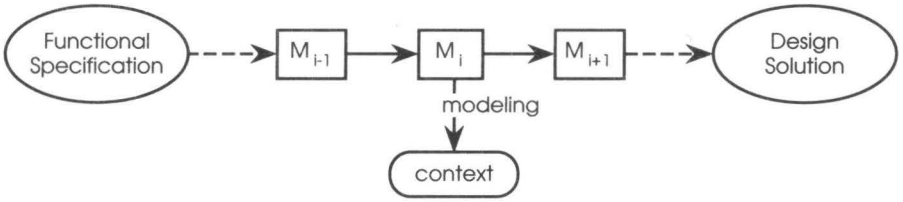
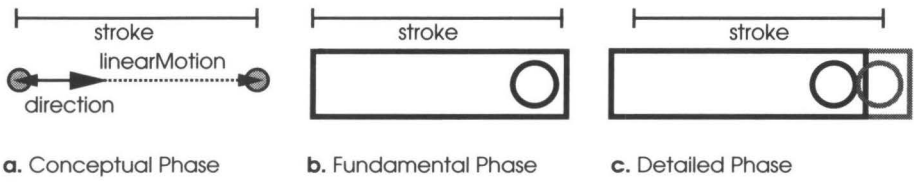


Fig. 2.3 Stepwise refinement of the meta-model.

In Fig. 2.4 an example of this process is depicted. It shows several stages of the design of a linear motion mechanism. In Fig. 2.4.a the state of design is at the conceptual design phase. The meta-model consists of an abstract anatomical description of the design object. Some states later the design is arrived at the fundamental design phase (see Fig. 2.4.b). The meta-model is a concrete anatomical description of the design object without detail, i.e there are some inconsistencies between the geometric and the kinematic representation. The stroke achieved by the geometric representation is not the desired stroke. The detail is present in Fig. 2.4.c when the design is at the detailed phase. Here, the meta-model consists of an exact anatomical description which is almost complete. Inconsistencies caused by results from different contexts are removed.

The design process as described above deals with the ideal situation in which the stepwise refinement process is a linear process from functional specifications straight to the design solution. It can be regarded as a sketch of the design process in retrospect. In practice, it is merely a process of trial and error, rather than the straightforward process shown in Fig. 2.3. The designer might not be satisfied with a certain state of the design and wants to redo it from a certain point. But (s)he keeps in mind the things which were useful and which were not, and the redesign will therefore be more efficient. In another occasion, designers might like to regard



a. Conceptual Phase **b.** Fundamental Phase **c.** Detailed Phase

Fig. 2.4 Three different stages of the design of a linear motion mechanism.

a design object from different points of view at the same time, i.e., they want to create multiple aspect models concurrently in order to compare the outcome from different experts.

A third possibility is that a designer is not sure about the direction the design should go at a certain point and wants to model some possibilities in parallel, i.e. (s)he conducts the design in multiple directions at the same time. Therefore, the stepwise refinement model must be extended with a mechanism to create multiple models in parallel. It allows the designer either to create multiple views on a single design object description, or to create multiple design object representations concurrently. These three different ways to direct the design are presented in the three sections below.

During the course of the design, an occasion frequently occurs that the designer is not satisfied with the current state of design. Instead of redesigning everything from scratch, the designer wants to preserve part of the results. The designer restarts the design from a previous design state which still met his/her demands. The implication for DPM is that it must be possible to withdraw the current meta-model and perform some *backtracking* to a previous one. In consequence, each individual state of the meta-model must be maintained as the design process proceeds. On top of that, when the design is continued from a prior meta-model, it must be prevented from taking the direction which led to the unwanted result. Thus not only the meta-model states, but also the design process history must be maintained.

In Fig. 2.5 an example of the backtracking process is shown. For some reason, the meta-model M_j does not fulfill the designer's requirements, so (s)he decides to redesign from a prior state. In this case, (s)he backtracks to the previous meta-model M_{i-1}. The design is restarted from this state taking a different direction. The design now proceeds to meta-model M_i, and so forth.

In some design situations there are more than one possibility to model the design object. Instead of forcing a designer to choose either of these possibilities, (s)he must be allowed to model multiple models concurrently. Such *multiple*

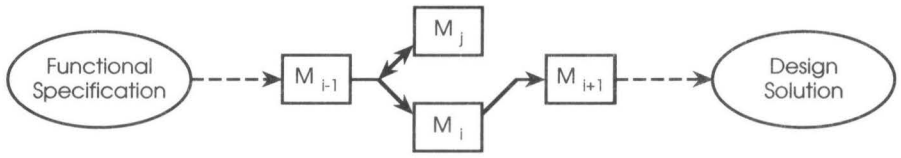


Fig. 2.5 Backtracking to a previous meta-model.

models refer to a single design object description. In other words, multiple models allow the designer to model different aspects of a design object, but these models must merge into a single design object description when there is a transition to a next state (see Fig. 2.6). Multiple models are alternatives that converge into a single meta-model. Therefore, multiple models are only a temporary fork in the design process.

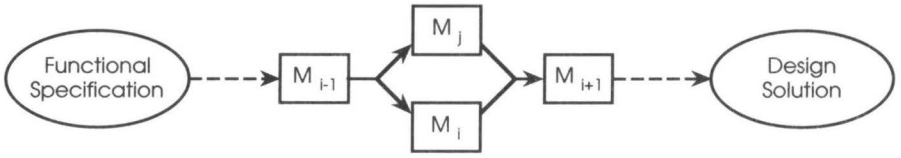


Fig. 2.6 Multiple models.

An illustration of this mechanism is given in Fig. 2.6. From the meta-model M_{i-1} the design process continues with two multiple meta-models M_j and M_i. These two meta-models represent a single design object description. They are therefore merged into a single unique meta-model M_{i+1}. The design process is continued from this meta-model, eventually by generating multiple models once again.

In other cases, the designer faces a situation in which there are several possibilities to solve a design problem. Each of these alternatives looks promising. So, instead of taking a decision at that time, the designer models each of possible solutions in parallel. Hence, the designer models multiple versions of the design object simultaneously. I call these versions *concurrent models*, since they refer to distinct design object descriptions. From this point on the design process proceeds on these concurrent models possibly resulting in multiple design solutions. Each of the concurrent models follows its own path and has nothing in common with other concurrent models.

An example of concurrent models is depicted in Fig. 2.7. Arrived at the meta-model M_{i-1} a designer can continue the design process in two different ways. Therefore, (s)he creates two concurrent meta-models M_j and M_i, and continues the design process following two parallel paths. The meta-models are transferred to M_{j+1} and M_{i+1}, etc. At last the design process may result in two different design solutions.

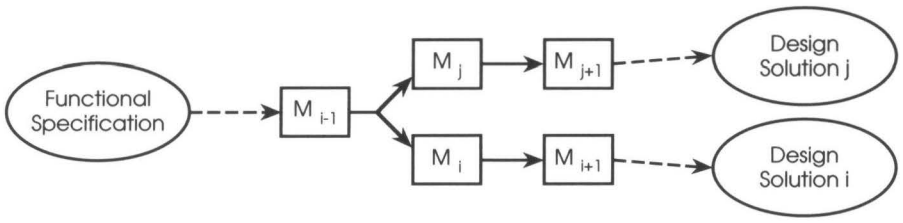


Fig. 2.7 Concurrent models result in different design solutions.

In some cases the designer likes some of the ideas in each concurrent model. Then, the system has a mechanism allowing the designer to merge several concurrent 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 designers with a framework that assists them in their design activities.

2.4 Other descriptive models

There are many types of design problems and many approaches to each of them. Therefore, the model presented in § 2.3 is not the only possible or desirable model. Nevertheless, many of the variations between different design process models have little impact on the requirements for design object representations [Arbab, 1991] as discussed in § 2.2. This section presents some design process models of different researchers. I used DPM as an inspiration for the development of ADDL though there is no strong connection between DPM and ADDL specifications. As a matter of fact, each of the models presented here fits within the general framework of ADDL. It can be regarded as a generic computer-based model.

Waldron [Waldron, 1991] presents a general framework of the design process in which multiple models of different researchers can be incorporated. As is stated in § 2.2 design is a mapping from function (problem) to structure (solution). Fig. 2.8 shows this process as moving along the axis of a cylinder. The spiral around the cylinder is an indication of the amount of knowledge used by a designer to

perform the mapping. Thus the larger the diameter of the cylinder the more knowledge a designer needs for a certain job. The latter may be an indication for the complexity of the task measured by the amount of experience or creativity required from the designer. Thus, a problem that is considered as a routine task because of a designer's experience will proceed along a narrow bound down the axis. Whereas the same problem may require all the knowledge of a designer who is less experienced. Therefore, the process will proceed over the surface of the cylinder, i.e. the designer applies full creativity.

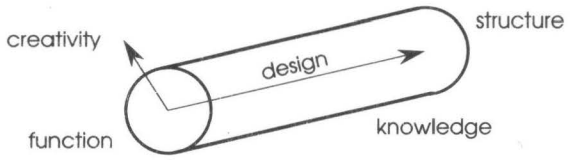


Fig. 2.8 A general framework for the design process.

2.4.1 The design process model by Gero

Gero has formulated a generic model for design that captures *routine* design, *innovative* design as well as *creative* design [Gero, 1990]. In his view, design is basically a transformation of a function F into a design description D (see Fig. 2.9). The described artifact is capable of producing these functions. The basic model is:

$$F \rightarrow D.$$

However, such a direct transformation does not exist. In his model a design description represents the artifact's elements and their relationships denoted by the structure S. The transformation of such a structure to a design description can be done with computer-aided drafting systems, i.e.,

$$S \rightarrow D.$$

Occasionally, there exists a direct transformation between function and structure. According to Gero it is called *catalog lookup* and it is not considered designing. Generally speaking, there is no direct transformation between function and structure, which leaves a requirement for an indirect transformation.

In another context, Bobrow defined function as the relation between the goal of a human user and the behaviour of a system [Bobrow, 1984]. Gero introduces two ways in which behaviour can be viewed in designing. The first view is the behaviour of the structure, i.e., B_s. It is termed *analysis* and it is a direct transformation from structure:

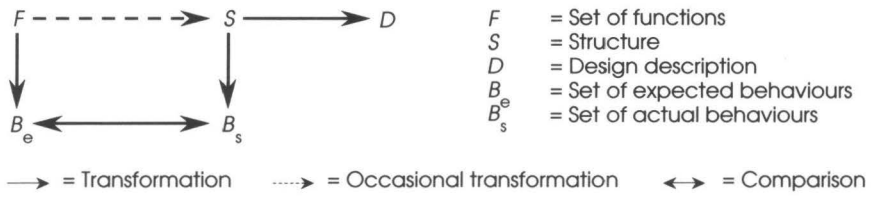


Fig. 2.9 A generic model for design.

$$S \rightarrow B_s.$$

Secondly, it can be viewed as the expected behaviour of the functions, i.e., B_e. The model is:

$$F \rightarrow B_e.$$

This process is called *formulation* or *specification* in design. In §2.2, I call it *specification* and I call the expected behaviour *functional specification.* In order to judge the correctness of the designed structure, the behaviour of the structure needs to be compared with the expected behaviour, hence:

$$B_s \leftrightarrow B_e.$$

The comparison process is called *evaluation* in design.

Another model of design is

$$F \rightarrow B_e$$

$$B_e \rightarrow S(B_s).$$

Here, the expected behaviour is used in the selection and combination of structure based on a knowledge of the behaviours produced by this structure. This process is termed *synthesis.* Synthesized structures produce their own behaviours, which can be a useful superset of the expected behaviours. Synthesis can change the range of expected behaviours and through them the function being designed for, leading to a *reformulation.* Reformulation can also occur when the actual behaviour of the structure is not satisfactory and cannot be made satisfactory. This happens when the evaluation has a negative result. Fig. 2.10 shows the activities involved with the design process: formulation, synthesis analysis, evaluation, reformulation, and production of a design description.

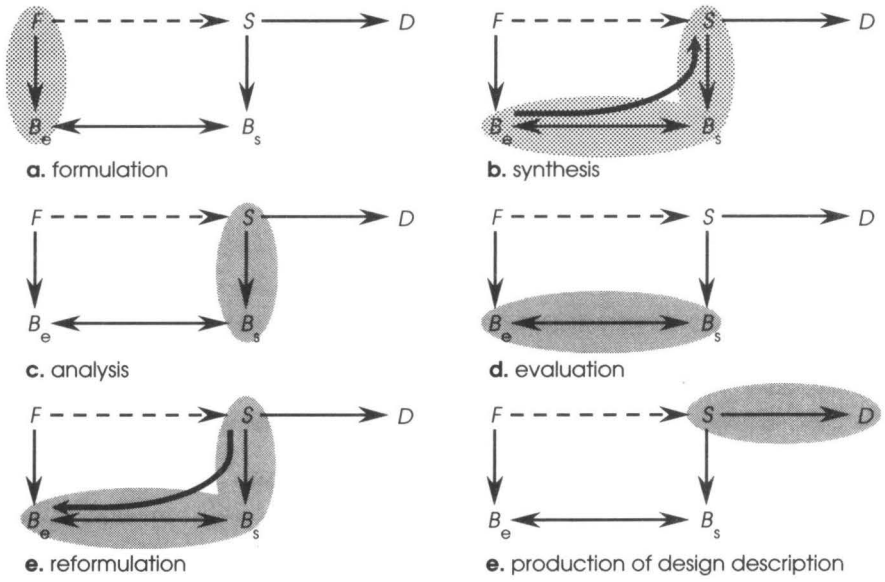


Fig. 2.10 Activities in design.

2.4.2 The design process model by Mostow

According to Mostow, the key research problem in AI-based design is to develop better models of the design process [Mostow, 1985]. In his article, he presents some aspects of the design process that a comprehensive model should address. They are:

1. The state of the design. Design evolves a series of artifact descriptions at various levels of detail. Design is viewed as a sequence of correctness-preserving transformations from one intermediate state of the artifact description into the other.
2. The goal structure of the design process. If design is a purposive activity, goals guide the choice of what to do at each point. These goals are not artifact descriptions but prescribe how those descriptions should be manipulated. An explicit goal structure roughly modeled as a tree makes it easier to replay the design process. The leaves of the tree form a sequence of transformations from functional specification to a design description.
3. Design decisions. Given a goal, there may be several plans for achieving it. Design decisions represent choices among them. Decision making should be represented as explicit goals.

4. Rationales for design descriptions. The rationale for choosing a particular plan to achieve a goal explains why the plan is expected to work and why it was selected instead of the alternatives. They are useful in replaying the design history to solve a new problem. Furthermore, explanation forces the reasoning behind the design to be made explicit which improves the quality of the design.
5. Control of the design process. Guiding design requires choosing which goal to work on at each point and choosing which plan to achieve it with. The reasoning behind the decisions what to do next need to be uncovered and represented explicitly. Various relationships between two goals are possible: independence, cooperation, competition, and interference.
6. The role of learning in design. Solving a design problem requires both general knowledge about the domain and specific knowledge about the problem. Learning is a way to acquire such knowledge.

2.4.3 The design process model by Takala

Takala bases his extended model of designing [Takala, 1987a] on the General Design Theory of Yoshikawa [Yoshikawa, 1981]. He recognizes some limitations in the theory. First of all, the theory assumes a fully specified design problem, which is often not the case in practice. Many larger design projects have specifications that change over time, or there are no exact evaluation procedures that tell objectively whether or not the requirements have been met. A second deficiency is that the theory does not describe the whole design process, but rather an unordered set of specifications and solutions.

According to Takala, a model of the design process contains two complementary representations of the design object: the abstract *intensional* (functional) requirements and the concrete *extensional* (metaphorical) realizations of them [Takala, 1987b]. The design process is aiming at situations where these are consistent with each other (see Fig. 2.11). *Analysis* functions check consistency by recognizing relevant properties of extensional representations. Design *synthesis* is the inverse of these analysis functions. It is a problem solving process to find a solution for the given specifications and implicit constraints.

Both the specifications and the solutions are subdivided in a tree structure. A specification tree represents the intensional description and the proposal tree represents the extensional description. The nodes of the trees contain different states of the design process. A generator-filter pair acts as an interface between the nodes. The generator synthesizes new states in the proposal tree and the filter analyzes them with the current state of the specification tree. During the design process both trees may be extended though it is more likely that only the proposal tree is extended.

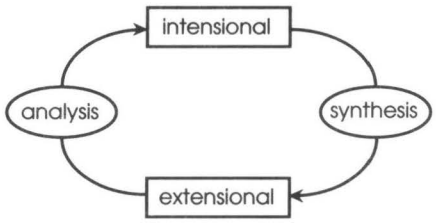


Fig. 2.11 The design process according to Takala.

2.5 Discussion

In this chapter, I have firstly given a prescriptive model of the design process. It describes how design should be done. The model distinguishes three different stages during the design process, viz. conceptual, fundamental and detailed design. Each of the three stages has its distinct demands concerning the representation of the artifact description. The descriptive model (DPM) presented in §2.3 is one of the models that describe design as a cognitive process. DPM is based on the extended general design theory of Tomiyama and Yoshikawa. It describes activities performed during the design process by means of the meta-model. The meta-model is a series of models of the design object which obtains data through stepwise refinement. Aspect models are derived from the meta-model in order to model and evaluate the design object from specific view points.

The meta-model plays a central role in this model. The meta-model serves a dual purpose, i) it is a central description of the design object that evolves during a stepwise refinement process, and ii) it is used as a reference model, from which aspect models can be derived. The meta-model contains the knowledge how to integrate aspect models. Aspect models have only information about the field of expertise they focus on. Therefore, only in the meta-model the knowledge is embedded how to integrate data derived from different aspect models, and how to solve eventual inconsistencies resulting from these data. The multi-context mechanism allows a designer to create multiple aspect models concurrently. Again, the meta-model serves as an intermediate among active aspect models. The multi-context mechanism can either result in a single integrated meta-model, or it can create different meta-models which are treated independently during the remainder of the design process. In the former case, I call them multiple aspect models, and in the latter case I call them concurrent.

Many descriptive models have been developed by different researchers. Three of them are discussed in §2.4. A common property of each of the models (including DPM) is that they regard design as an evolutionary process. The

description of the artifact evolves as the design proceeds. Furthermore, each of the four models regard design as a goal oriented activity. The major difference among the models is the level of detail in which the design subprocesses are described. At the one end, there are DPM and Takala's model that model designing solely as synthesis/analysis. At the other end, Gero's model distinguishes seven distinct activities in the design process.

DPM served as a source of inspiration for the development of ADDL. Chapter 4 lists a number of design maxims that represent the requirements that a programming language for implementing the IIICAD system must fulfill. They serve as a basis for the formal language specifications of ADDL. However, ADDL is not restricted to implementing DPM only. Each of the descriptive models is reflected in ADDL's specifications.

3

A Small Design Problem

3.1 Introduction

Chapter 2 proposed a prescriptive model concerned with three consecutive stages of the design process. The remainder of this dissertation concentrates on a representation language for design. To some extent, demonstrating the language constructs by means of an example is inevitable. Therefore, this chapter introduces a small design problem referred to throughout the dissertation. It discusses how a designer solves this problem in accordance with the prescriptive model. The problem dealt with has been inspired by discussions with the staff of the Yoshikawa/Tomiyama Laboratory at the University of Tokyo in Japan. It involves the design of a mechanism for generating an oscillation motion of a lever from a linear motion of an air cylinder. It originates from a Japanese text book on mechanical engineering (see [Kumagai, 1976], p.88) and it is first introduced in a paper by Xue *et al.* [Xue *et al.*, 1990].

I give a description of the design process resulting in a solution to the design problem. The design process is subdivided into the three consecutive stages presented in the previous chapter, viz. *conceptual*, *fundamental*, and *detailed* design. In the next section a general description of the design problem is presented. In § 3.3, I present the various types of reasoning involved with finding the design solution. The stepwise solution to the problem successively passing through one of the three design process stages is given in § 3.4 till § 3.6. Interwoven with the description of the stepwise refinement process, I give the outcome of the design by actually assigning values to attributes of the lever and its parts. The last section discusses the design problem and the way it is solved.

3.2 A bounded linear motion mechanism

The problem treated in this chapter deals with the design of a bounded linear motion between two specified points. This mechanism is a part of a more complex object. I assume the existence of a lever being part of a rack and pinion assembly (shown in Fig. 3.1). The mechanism is used for converting reciprocal into circular motion. It can be used for a pick-and-place task. The assembly consists of a pinion which is rigidly attached to a crank. Furthermore, it has a rack which is moved by an air pressure cylinder, and it has a lever which performs the circular motion. The motion is achieved by a pin which itself performs a linear motion bounded by a slot inside the lever. The design of the latter mechanism is the problem dealt with in this chapter.

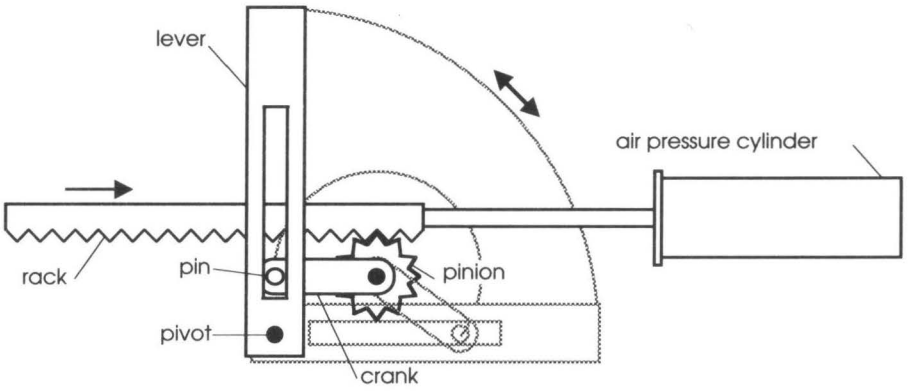


Fig. 3.1 A rack and pinion assembly. The horizontal movement of the rack is converted to a circular movement of the lever.

The assembly behaves as follows. When the rack moves to the right, it makes the pinion rotate and, consequently, the attached crank as well. This results in a linear motion of the pin inside a slot of the lever, which will therefore rotate also. The centre of rotation is determined by the position of a pivot being part of the lever.

The lever is therefore built up by two components, i.e., a pivot and a slot. The position of the pivot is taken as the origin of a (local) coordinate system for determining the geometry of the lever. The coordinates of all vertices of both the lever and the slot are relative to this origin. The position of the pivot itself is given by the distance to the lever's nearest face. The x-axis is parallel to the longest face of the lever, and the y-axis is parallel to the shortest face. The pivot has a diameter of 10 mm, and its position is on the centre line of the width of the lever. The slot is

used to bound the linear motion of the pin. It must be constructed in such a way that it does not obstruct the pin's movement. The pin is not one of the lever's components, it is a separate object. The lever, the pivot, the slot and the pin are depicted in Fig. 3.2.

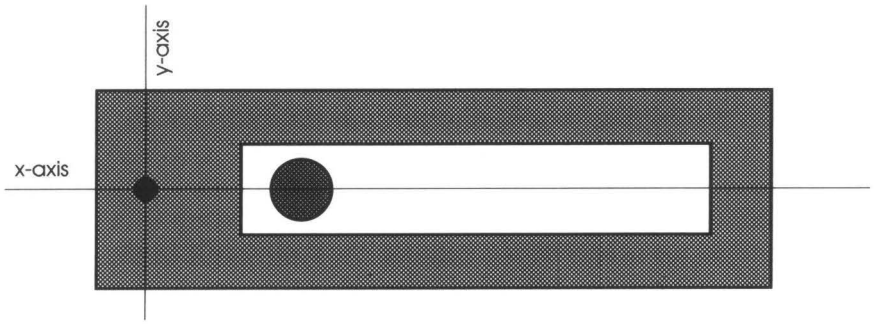


Fig. 3.2 A lever with a slot and a pivot, and a pin. The pin moves along the x-axis.

The functional specification of the linear motion mechanism consists of a starting point of the motion, `S`, and an ending point of the motion, `E`. Both points are given by their distance to the pivot along the x-axis. The design starts by giving requirements of the form: `S=value`$_1$ and `E=value`$_2$. Both `value`$_1$ and `value`$_2$ are values given by the designer. The design is finished when there is a mechanism which allows the pin to move from `S` to `E` without being obstructed by the slot.

3.3 Conceptual design of the linear motion mechanism

During conceptual design of the linear motion mechanism the designer constructs an abstract anatomical description of the mechanism. Since I am dealing with mechanical engineering, this description consists of a *qualitative model* of the desired functionality. This model is a generic model to be used for the solution of a certain category of design problems dealing with linear motion mechanisms. In the example design system presented in Chapter 7, one solution uses a slot and a pin for the guide of the motion and the object in motion respectively. Another uses a shaft and a slider. A third solution uses a rail and a table. The qualitative model of the mechanism reads as follows:

There are two objects, one object that accomplishes the motion, the *object-in-motion*, and another that guides the object in motion, the *guide*. The object-in-motion performs a *linear motion* between two given points, `S` and `E`. In each of these two points there is a *limit arrangement* between the object in motion and

the guide of the motion, i.e., there is a situation in which the object in motion is stopped by the guide. An outline of such a linear motion mechanism is shown in Fig. 3.3. The solution to the design problem, dealt with in this chapter, uses a slot for the guide and a pin for the object-in-motion. The qualitative model of the motion mechanism is extended during the fundamental design, when more properties about the linear motion mechanism are known.

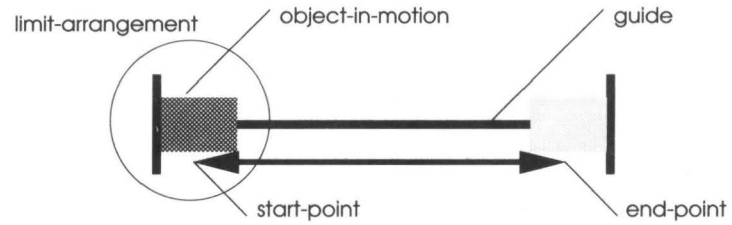


Fig. 3.3 Linear motion mechanism.

The abstract anatomical structure is applicable to three different linear motion mechanisms. The one which use a slot and a pin is discussed in this chapter. The slot and pin solution is chosen by the designer at an earlier stage of design. This choice stems from the nature of the rack and pinion assembly. This assembly puts constraints on the carrier of the linear motion mechanism, viz. it must be erected in such a way that it can rotate. Both the shaft and slider, and the rail and table solution are statically located mechanisms. Therefore, a lever is chosen for the carrier of the motion, and the slot is a feature of the lever. The lever, slot and pin form a triad.

The lever must meet certain requirements that have nothing to do with the linear motion mechanism. For instance, the lever is attached to an arm which picks an item, moves it, and places it in another location. The lever's dimensions are determined during its conceptual design. The lever has the following attributes: a length, L_1, a width, W_1, a thickness, T_1, a position of the pivot, P_1, and a range of the motion, R_1, which is equal to the length minus the position of the pivot. (Which is, in fact, the length of the part of the lever to the right of the y-axis illustrated in Fig. 3.3). In order to meet the constraints imposed by the arm, the designer assigns the values shown in Table 3.1 to the lever's attributes.

When the abstract anatomical structure is determined, the designer specifies values for the requirements the mechanism must fulfill. The value for the starting point S of the motion is given, I call it S_0, and the value for the end point E of the motion, being E_0 is given. The object in motion is regarded as a point mass moving from S to E. In practice, it is the centre point of the object in motion.

Attribute of lever	Notation	Specification
length	L_1	520
width	W_1	100
thickness	T_1	20
position of pivot	P_1	20
range of motion	R_1	$L_1 - P_1$

Table 3.1 Dimensions of the lever[2].

Both values are given by their distance to the origin, i.e., the location of the pivot. These requirements must obey some constraints imposed by the physical properties of the lever. These constraints are (see Table 3.2):

i. The value for starting point of the motion must be greater than or equal to half of the value of the width of the lever and it must be smaller than or equal to value of the range of the lever minus one and a half times the value of the width of the lever.

ii. The value for the ending point of the motion must be greater than or equal to the value of the starting point plus the value of the width of the lever and it must be smaller than or equal to value of the range of the lever minus half of the value of the width of the lever.

Requirement	Notation	Constraint	Specification
start of motion S	S_0	$W_1 / 2 \leq S_0 \leq R_1 - W_1 * 1.5$	100
end of motion E	E_0	$S_0 + W_1 \leq E_0 \leq R_1 - W_1 / 2$	300

Table 3.2 Requirements, notations constraints, and specifications of the linear motion mechanism.

In other words, the first constraint ensures that the starting point is located at a minimum distance from the hole, and it ensures also that a minimum space is preserved for the linear motion mechanism. The latter constraints the end point. It ensures a minimum space for the motion mechanism and it guarantees that the end point is located at a minimum distance form the right-most edge of the lever. The requirements, notations, constraints, and specifications are shown in Table 3.2.

[2] All dimensions are in millimetres.

Besides a general solution to the design problem, I give the working-out of a real design using the example design system presented in Chapter 7. The steps and actions taken by the designer are examined as the design proceeds. The example is presented with indentation in a different font as shown below.

> The designer starts the conceptual design of the linear motion mechanism during the fundamental phase of the rack and pinion assembly shown in Fig. 3.1. The design of the mechanism is therefore strongly influenced by the physical properties of the lever being part of it. First of all, since the motion mechanism will be part of the lever, the designer is forced to choose a slot and pin solution. Secondly, the dimensions of the lever constrain the position and the maximum size of the slot. The lever and its dimensions are shown in Fig. 3.4.

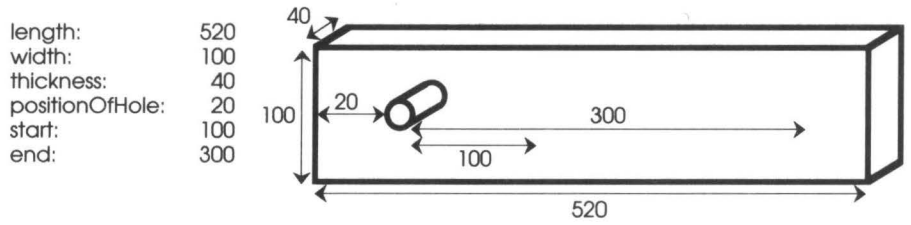


Fig. 3.4 Initial state of the lever.

> The functional specifications of the linear motion mechanism are given by the designer. The motion must start at least at position: $S_0 = 100$ and must end at most at position: $E_0 = 300$. Then the designer builds the abstract anatomical structure of the mechanism. It reads as follows: there is a linear motion of the pin between `100` and `300`, and there are two limit arrangement between the slot and the pin, one in position `100` and the other in position `300`. In other words, the pin must be able to move between `100` and `300` without being obstructed by the slot. The functional specification and the abstract anatomical structure together form a qualitative model describing the function and behaviour of the linear motion mechanism.

In order to build an initial (possibly inconsistent) model of the mechanism, the designer specifies some of the attributes of the mechanism. The concrete structure of the lever is already determined at an earlier stage of design, i.e., the fundamental design of the lever. The detailed design of the lever involves the design of the slot and pin. The attributes of the pin and the slot are: i) the *position* of the slot, P_s (given by the distance to the origin, i.e., the pivot), ii) the *length* of the slot, L_s, iii) the *width* of the slot, W_s, and iv) the *diameter* of the pin, D_p. The attributes of the mechanism can be revised at a later stage of the design process.

The specifications must obey some constraints imposed by the physical properties of the lever, and by already provided attributes (e.g. the length of the slot is constrained by the position of the slot). These constraints are:

i. The *position* of the slot must be greater than or equal to 10 mm since there must be a minimal clearance between the hole and the slot. It must be smaller than or equal to the range of the lever minus the width of the lever minus 10 mm since there must be room for the slot having a certain minimal length specified by the next constraint.

ii. The *length* of the slot must be greater than the width of the lever. Note that this is quite an arbitrary choice based on the fact that the length of the slot must be greater than the width. It must be smaller than or equal to the range of the lever minus the position of the slot minus 10 mm since the slot would otherwise not fit in the lever.

iii. The *width* of the slot must be greater than or equal to 10 mm being the minimal diameter of the pin. It must be smaller than or equal to the width of the lever minus 20 mm allowing a minimal clearance of 10 mm at both sides of the lever.

iv. The *diameter* of the pin must be greater than or equal to 10 mm certifying a minimal strength of the pin. It must be smaller than or equal to the width of the slot for obvious reason.

The attributes, notations, constraints, and specifications are shown in Table 3.3.

Attribute	Notation	Constraint	Specification
position of slot	P_s	$10 \leq P_s \leq R_1 - W_1 - 10$	100
length of slot	L_s	$W_1 \leq L_s \leq R_1 - P_s - 10$	200
width of slot	W_s	$10 \leq W_s \leq W_1 - 20$	50
diameter of pin	D_p	$10 \leq D_p \leq W_s$	40

Table 3.3 Attributes, notations, constraints, and specifications of a linear motion mechanism.

The designer has described the mechanism in terms of function and behaviour. A slot meets such a description. Next, a concrete structure of the mechanism is made, i.e., a rough geometric model of the slot. In order to realize such a model the designer gives some provisional values to the slot's attributes. These values are shown in the fourth column of Table 3.3. The choice of values is quite arbitrary, since the designer is not yet interested in an accurate model. He just wants a rough description whose behaviour can be tested.

The initial specifications for the slot are used to determine an initial concrete anatomical structure of the slot. When the dimensions of the slot are known, its geometric model can be constructed. The coordinates of the vertices of the slot are computed using these dimensions. The geometric model allows the designer for checking the consistency of the specifications. However, finding values for the slot's attributes is the aim of the presented design problem. Therefore, the values for the attributes *given* by the designer are only assumptions used as initial values to work with. These values might be incorrect, e.g. the length of the slot might not be sufficient to realize the linear motion. Such an inconsistency will be detected during the design process and it will result in a revision of the initial specifications.

> In this particular case, the designer takes the specification of the start position of the motion as the value for the position of the slot, i.e., `100`. For the length of the slot, he chooses the stroke of the motion, i.e., `200`. Obviously, these specifications will not lead to a correct solution since it demands a pin with a zero width, which is evidently not possible. Establishing a rough geometric model is the only purpose of these specifications. The specifications are revised during detailed design.

3.4 Fundamental design of the linear motion mechanism

The abstract anatomical structure constructed so far, serves as a basis to perform fundamental design. A concrete anatomical structure is built in three steps. During the first step the geometry of the slot of the lever is constructed using the attribute specifications provided by the designer during conceptual design (see Fig. 3.5). The qualitative model of the motion mechanism is extended by interpreting the geometrical properties. This is the second step. At the third step, the designer creates a kinematic model of the linear motion mechanism to check whether the mechanism fulfills the requirements, i.e., the pin can move between `S` and `E` without being obstructed.

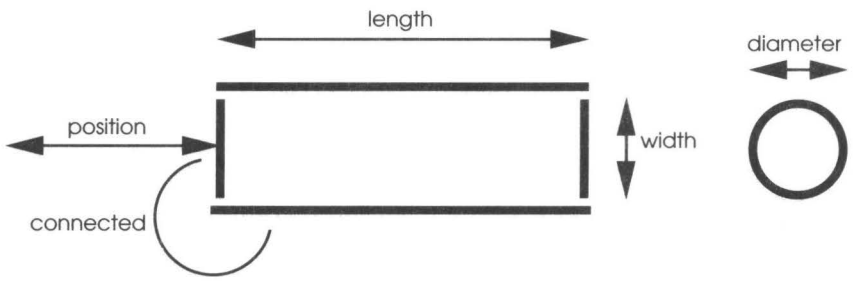


Fig. 3.5 Geometrical properties of the slot and the pin.

Hence, constructing a geometric model of the slot and the pin is the first step the designer takes. The slot has four *faces* forming a rectangle. The faces are one by one *adjacent* and they are in parallel with the faces of the lever. The position of the left face is determined by the specified position of the slot. The length of the left and right face, and the length of the top and bottom face are determined by the width and length of the slot respectively. The pin has three *faces*, one face forming the front face, another face forming the back, and a third face forming a cylinder. The first two faces are not of importance and are omitted from the discussion.

> The designer uses the attributes of the slot to construct a geometric model. It is composed of four adjacent faces having x- and y-coordinates for four vertices. The z-coordinates are omitted since the slot is positioned relatively to the lever in the same surface. The position of the slot is used to determine the values the left-most x-coordinates. The length of the slot is used to determine the values of the right-most x-coordinates. The y-coordinates are determined by taking half the width of the slot in either upward or downward direction. The four adjacent faces have the following vertices: `(100,25)`, `(100,-25)`, `(300,-25)`, `(300,25)`. They are oriented in an anti-clockwise direction starting form top left. This tentative geometric representation of the slot is shown in Fig. 3.6. It also shows two pins. They are positioned on the desired start and end positions indicating that the design is inconsistent.

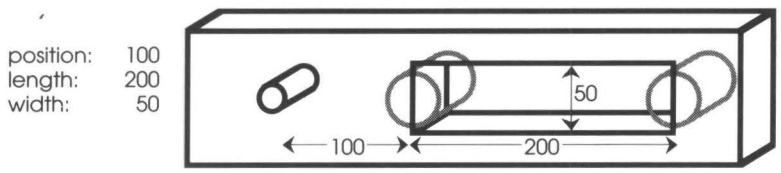


Fig. 3.6 Tentative geometric model of the slot.

At the second step when the geometry of the slot is defined, the designer extends the qualitative model of the linear motion mechanism. The *start* and *end positions* of the linear motion can now be determined. This can be done by applying the knowledge, acquired in §3.4, about the limit arrangement between the object in motion and the guide of the motion. The start position is equal to the location of the centre of the pin, when there is a *contact* between the pin's face and the left face of the slot. The end position is likewise determined by the location of the centre of the pin when there is a contact between the pin's face and the right face of the slot. I call these positions S_1 and E_1.

The designer knows that in a limit position there is a contact between the pin and one of the slot's faces. Since both the diameter of the pin and the coordinates of that face are known, the designer can determine the location of the pin. That location is the attained start position S_1 of the motion, obtained by adding half the diameter of the pin to the x-coordinate of the vertices of the slot's left-most face, i.e., $100+\frac{40}{2}=120$. By the same token, the location of the end position of the motion is obtained by subtracting half the diameter of the pin form the x-coordinate of the vertices of the right-most face, i.e., $300-\frac{40}{2}=280$. The left-most face and the right-most face of the slot are the faces in contact with the pin in the limit positions.

The start and the end positions are used to validate whether the design satisfies the requirements S_0 and E_0. The obtained start position S_1 must be smaller than or equal to the required start position S_0 and the obtained end position E_1 must be greater than or equal to the required end position E_0. The design is consistent, if these requirements are fulfilled. The possible states of design are illustrated in Fig. 3.7.

(a) Consistent design **(b)** Inconsistent design

Fig. 3.7 Consistent and inconsistent designs. In (b) the pin can not move to the required position S_0.

The designer compares the attained limit positions of the motion mechanism with the required positions. The design turns out to be inconsistent at both limit positions, for $S_1 > S_0$ $(120 > 100)$ and $E_1 < E_0$ $(280 < 300)$.

It is important to realize that the attribute values of the slot provided by the designer are only *assumed* values. They are employed as temporary values used to set up the slot's geometry. Since the geometry is now constructed and the requirements have been checked, the design can be refined during detailed design. If no inconsistencies are found, the values given by the designer appear to be correct, and they can be sustained. On the other hand, the slot's attributes must be revised during detailed design, if the values appear to be incorrect, i.e., inconsistencies are detected.

3.5 Detailed design of the linear motion mechanism

The values for the slot's attributes given by the designer are used to build a tentative geometric model of the slot. This model turned out to be inconsistent, and will be refined during detailed design. There are three possible cases of inconsistency, viz. i) the slot obstructs the pin at the start position of the motion, ii) the slot obstructs the pin at the end position of the motion, and iii) the slot obstructs the pin at both positions. In either case, the slot's specifications must be refined to meet the specified requirements. This amounts to a modification of the slot's attribute values.

In the first case, the face of the slot which is in contact with the pin in the start position must be moved to the left. This is achieved by increasing the slot's length and decreasing the slot's position. In the second case, the face of the slot which is in contact with the pin in the end position must be moved to the right, i.e., the length of the slot is increased. In the third case, when both situations apply, the slot's length is doubly increased, and the position is decreased. The three cases are shown in Fig. 3.8.

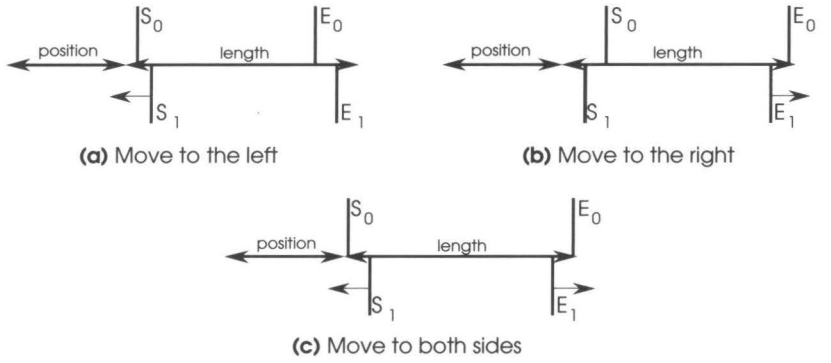


Fig. 3.8 Three cases of inconsistency.

> The designer has detected an inconsistency in the both the slot's limit positions. In order to remove the inconsistency, the slot has to be enlarged at both ends. Hence, the left-most face of the slot must be moved to the left by $S_1 - S_0$ (120−100=20). The right-most face must be moved to the right by $E_0 - E_1$ (300−280=20).

The geometry of slot is updated in accordance with the modified length and position. For the cases in Fig. 3.8(a) and Fig. 3.8(c) the face in contact with the pin in the start position must be moved to the left. The adjacent faces must be updated as well. The face in contact with the pin in the end position must be moved to the

right in the cases Fig. 3.8(b) and Fig. 3.8(c). Now I have adjusted the slot's geometry, I redetermine the start and end position of the motion, resulting in new values for S_1 and E_1. The mechanism finally meets the desired requirements, i.e.,

$$S_0 \geq S_1 \wedge E_0 \leq E_1$$

The designer changes the attributes of the slot in accordance with the desired modifications. The position of the slot is reduced by 20 and the length of the slot is increased by $20+20=40$. The designer employs the slot's modified attributes to create a new geometric model. The slot's attributes and its updated geometric model are shown in Fig. 3.9. The updated coordinates of the slot's faces are: (80,25), (80,-25), (320,-25), (320,25).

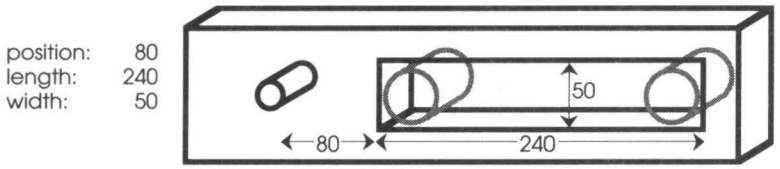


Fig. 3.9 State of the slot after detailed design.

The designer calculates the new limit positions S_1 and E_1 of the linear motion mechanism. The new values are: $80+\frac{40}{2}=100$ and $320-\frac{40}{2}=300$, which is in accordance with the requirements.

3.6 Reasoning about the design process

Concerning the reasoning mechanisms applied by the designer, I distinguish between *meta-level* reasoning and *object-level* reasoning [Takeda *et al.*, 1990; Treur, 1991c]. When reasoning at the meta-level, the designer takes strategic decisions on how to proceed with the design, i.e., what must be done next? The state of the design process is examined. Depending on that state the designer formulates a design *goal* which needs to be fulfilled. An example of such a goal is constructing an abstract anatomical structure of the desired artifact. When a goal is too complex to be solved in a single step, it is subdivided into sub-goals and so on.

At the object-level the designer thinks about the state of the design object. How to extent the design object model in order to reach the current design goal. For instance, when the current goal is to find an abstract anatomical structure, reasoning at object-level concerns with examining the object description and adding facts to it in order to obtain such a structure. The subdivision of goals

describing a design process is therefore a tree structure with meta-level goals in the nodes and object-level goals in the leaves. The design problem introduced in this chapter can similarly be described by a tree. This tree is shown in Fig. 3.10 to 3.13. The words appearing in rounded boxes are meta-level goals. Those appearing in square boxes are object-level goals.

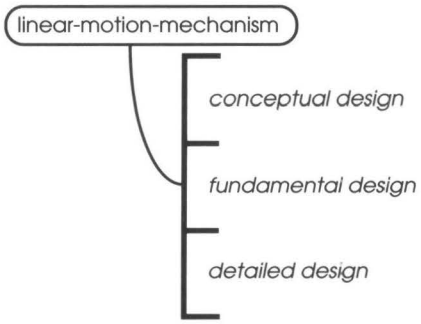


Fig. 3.10 Goal structure of a linear motion design problem.

The goal `linear-motion-mechanism` is the root of the tree. It represents the ultimate goal of the design problem, i.e to design a linear motion mechanism. It is subdivided into eight sub-goals. The first two goals (a meta-level and an object-level goal) represent the conceptual phase of design. The third till the fifth represent fundamental design, and the last three stand for detailed design of the linear motion mechanism. These three branches of the tree representing conceptual, fundamental, and detailed design are now further explained. The top-level goal of the tree is presented in Fig. 3.10, detailed information about the branches is left out.

In order to solve the top-level goal a complete description of a linear motion mechanism must be made. The reasoning at this level is meta-level reasoning. It concerns the state of the design process. The goal is solved in three conceptually different stages, *conceptual*, *fundamental*, and *detailed* design. First of all, if nothing is known about the guide of the motion, its abstract anatomical structure (the meta-model) should be created. When the meta-model has been made, it has to be extended to a concrete anatomical structure. Then, if the concrete anatomical structure is not yet exact, the meta-model has to be refined during detailed design.

In Fig. 3.11 the part of the tree concerning conceptual design is depicted. The first goal to be solved is constructing the meta-model of the linear motion mechanism. It involves object-level reasoning. The designer will try to find a qualitative model which describes the design object in terms of function and behaviour. When the goal has been solved the output of the reasoning process will

be an abstract anatomical description of the design object. The next goal is a meta-level goal. The reasoning pertains to the kind of solution chosen for guide of the linear motion mechanism. In this particular example, a slot will be chosen for the guide. This decision will be taken by the designer and will lead to a sub-division into two sub-goals. Both goals implicate object-level reasoning. At first, the designer has to specify the requirements the motion mechanism must meet. The meta-model will be extended with this information. Then, the designer must specify some provisional values for the attributes of the slot. These assumptions will be necessary for constructing the slot's shape during the next stage of design.

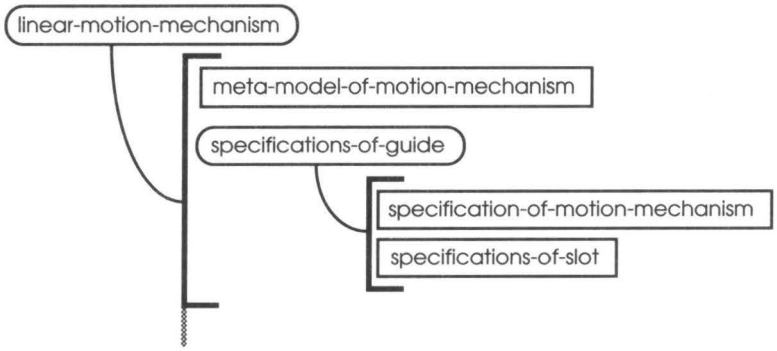


Fig. 3.11 Goal structure of the conceptual phase.

The branch of the tree regarding fundamental design is shown in Fig. 3.12. The first meta-level goal is about the kind of solution chosen for the guide. Since the design deals with a slot, the next goal to be solved is constructing a geometric representation of the slot, which is purely object-level reasoning. When the geometry is known, the designer will extend the meta-model with facts which can be derived from the geometry. This again is object-level reasoning. Arrived at this point of the design process the validity of the design can be checked. Again, a decision on the kind of solution is made. Then, the limit positions of the motion mechanism can be calculated. If these fit within the geometry of the slot, the design fulfills the specifications. If they do not, the provisional specification of the slot must be revised during detailed design.

The remaining part of the goal tree concerning detailed design is shown in Fig. 3.13. This part of the tree will only be reached if the initial specifications of the slot were incorrect. Detecting the fault in the motion mechanism is the first goal. It is solved by object-level reasoning. Again a choice on the kind of solution is made. Next, object-level reasoning about how to modify the slot in order to meet the specified requirements. The last node of the tree `limit-positions-of-guide` is

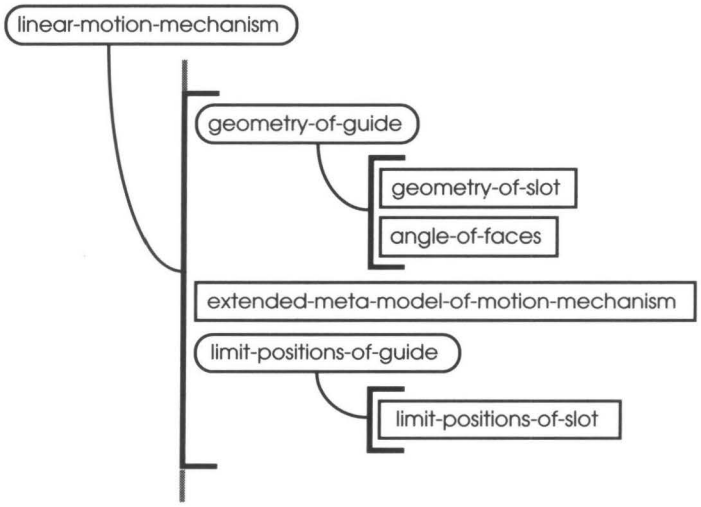


Fig. 3.12 Goal structure of the fundamental phase.

the same as last node of the fundamental design phase. Checking the consistency of the solution is its purpose. When the limit-position of the linear motion mechanism are within the required bounds, the design will be exact. If the design is not yet exact the last steps of the design process have to be redone in order to improve the design. This kind of meta-level reasoning is executed for solving the top-level goal `linear-motion-mechanism`.

3.7 Discussion

This chapter presented a small design problem and it showed a way to solve this problem. The stepwise refinement of the design object in accordance with the design process model (DPM) is demonstrated. A qualitative model of the mechanism is used to serve as the part of the meta-model to derive aspect models from. The qualitative model consists of the following propositions:

```
linear-motion(pin,point1,point2)
limit-arrangement(pin,slot,point1)
limit-arrangement(pin,slot,point2)
contact(slot-face1,pin-face,point1)
contact(slot-face3,pin-face,point2)
startPosition(point1)
endPosition(point2)
```

The aspect models used in this chapter are a geometric and a kinematic aspect model. The former is used to determine the limit positions of the linear motion

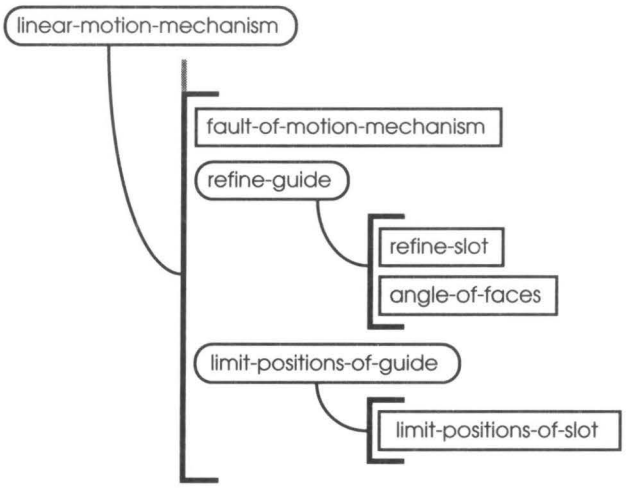


Fig. 3.13 Goal structure of the detailed phase.

mechanism, the latter is used to validate whether the mechanism actually meets the specified requirements. It also contains the knowledge how to correct inconsistencies. The geometric knowledge is subsequently used to adjust the slot's geometrical properties accordingly.

The characteristics of the three stages of the design process presented in the previous chapter became apparent. During *conceptual* design the designer builds a qualitative model of the design object without being focussed on physical structures. The model is described in terms of function and behaviour. Namely, there is a linear motion between two points, and there is a limit arrangement in each of these two points. Then, during *fundamental* design the qualitative model is mapped to a decomposed structure describing the principal shape of the design object. Its geometry is defined and its functionality is conform the desire requirements. However, the design may contain some deficiencies, which are removed during *detailed* design. This is a revision process which detects the slot's attributes causing the inconsistency, and gives them a proper value. The geometry of the slot is therefore corrected in accordance with the inconsistencies found in the kinematic model.

Chapter 7 presents an example design system implemented in ADDL. It solves in dialogue with the designer the design problem introduced in this chapter. Technical details about the solution, for instance how the inconsistency is detected, are given in ample discussion.

4

Design Criteria for ADDL

4.1 Introduction

The *descriptive models* of the design process explained in Chapter 2 form a source of inspiration for the ADDL language specifications and the IIICAD system architecture. The formal language specifications are given in the next three chapters. The subject of this chapter is the transition from the absolutely abstract model to the complete concrete specifications. From the models a number of criteria for the design of ADDL and the underlying IIICAD system can be derived. They are concerned with the architecture of the IIICAD system and the requirements for a programming language to implement the system. Both the requirements for the system architecture and the language specifications are formulated with the concepts presented in the second chapter of this dissertation in mind.

The environment, in which ADDL runs, plays a substantial role in its development. Since ADDL is a special purpose programming language designed for implementing intelligent CAD systems, I am not overly concerned with portability and generality. The system and the language are thus strongly coupled. I employ a number of *design maxims*[3] for the formulation of the system architecture and the language specifications. They are an informal means to bridge the gap between a descriptive model on the one hand, and a formal specification of the design knowledge representation language on the other hand.

---

[3] Design maxims are introduced here to describe a requirement that ADDL must meet. The Concise Oxford Dictionary of Current English states: "**maxim**, n. A general truth drawn from science or experience; principle, rule of conduct."

The next section gives a representation of the IIICAD system. In §4.3, I specify the requirements for the representation of the design process. The resulting programming language constructs are subsequently presented. In §4.4 the language constructs for the specification of the design object are given. All derived language constructs are prefixed with **DM** (Design Maxim). The organization of this chapter is influenced by Veth's paper [Veth, 1987] where the original IDDL language specifications were presented.

4.2 The IIICAD system

I want IIICAD to be a system based on expandable ideas and a framework where designers can exercise their faculties at large. I believe that the essential thing in designing is that the designer creates his own design environment and the IIICAD system must give him the freedom to do so [Tomiyama and ten Hagen, 1987; Rogier, 1989; Bylander and Chandrasekaran, 1987]. The system must understand the designers commands and translate them into system's tasks. Routine tasks must be performed automatically, but irregularities must be detected and reported, so that the designer can react adequately to them. The intended behaviour of the system requires the system to know about many different aspects of design. A lot of different kinds of knowledge must be embedded in the system in order to achieve the above functionality. The IIICAD architecture is composed of several components. Each of these has knowledge about and the responsibility for one or more of the tasks it is charged with. Fig. 4.1 shows the architecture.

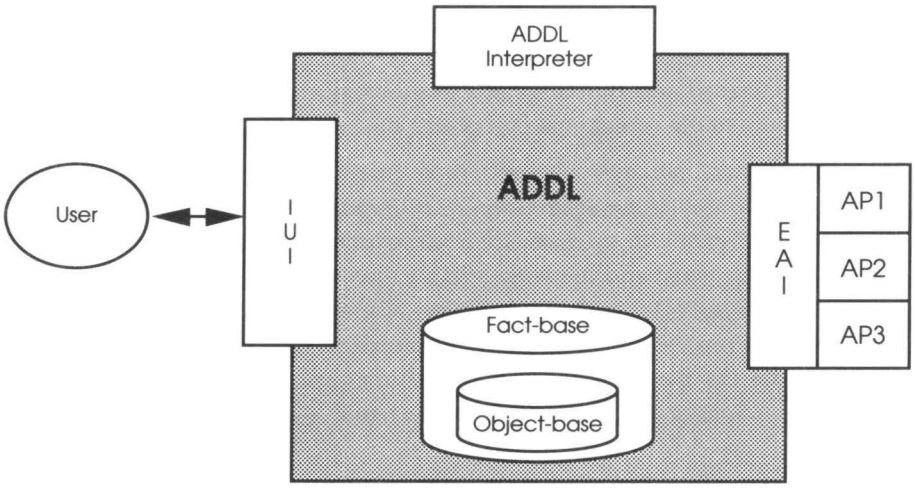


Fig. 4.1 IIICAD architecture.

The IIICAD system consists of i) an *ADDL interpreter*, ii) a *fact-base*, iii) an *object-base*, iv) an *intelligent user interface* and v) an *external application interface*. The kernel language of the system is ADDL, i.e., the several system components are either an integral part of the language (the object-base and the fact-base) or they have an interface to ADDL (the ADDL interpreter, the intelligent user interface, and the external application interface). An important construct in ADDL is a *scenario*. It is a set of methods and rules applicable to a certain stage of the design process. The execution of an ADDL program involves a sequence of scenarios being applied to the design object model. The ADDL interpreter controls the execution of scenarios and the flow of information among the components of the system. Therefore, it maintains the consistency of the object-base and the fact-base and it also directs the dialogue with the designer.

The fact-base contains all the literal facts currently known about the object being designed. It is gradually extended as the design proceeds to contain more and more detailed information about the artifact. The fact-base describes the structure of all parts an artifact is composed of. The object-base stores these parts as separate objects, each object having its own internal state. The objects are recognized by the relationships that are defined among them. The object-base is embedded in the fact-base, i.e., it is a part of the fact-base. The fact-base and the object-base together contain the data currently known about the artifact, called *object information state*.

The Intelligent User Interface (IUI) interacts with the designer. It translates tasks given by the designer into system commands. Furthermore it shows the designer the design tasks the system is currently involved in, and it reports the most recent state of the design object.

External applications are programs that provide the IIICAD system with information not available from any of the system's components. The information can be provided in the form of extra information about the design object description or it can be the evaluation of the design object in a certain context, e.g., FEM analysis. The External Application Interface (EAI) takes care of the contents of the flow of information between the IIICAD system and an external application. External applications can be written either in ADDL or in another programming language. In the former case, the EAI has a nearly trivial job.

4.2.1 The interpreter

An ADDL scenario is a piece of design knowledge employed by the system to perform a design step as mentioned in §2.2.2. It consists of a set of methods and rules that query the object information state and derive some information based on that state (more about scenarios in §4.3). The interpreter's control loop is as follows: depending on the current design *goal*, a scenario is activated. This scenario is interpreted until it terminates. The state of either the design process or

the design object is updated. A next scenario is chosen. When an applicable scenario cannot be found the interpreter asks the designer to provide more information, which is either a more precisely described design goal or more data concerning the design object. A sequence of consecutively active scenarios represents the design process. The above can be summarized in the following design maxim:

DM 1. *The ADDL interpreter controls the execution of scenarios in order to conduct the design process.*

Furthermore, the interpreter takes care of the backtracking of the system. When, at a certain stage of the design process, the designer decides that the current direction, in which the design process is going will not lead to anything, the interpreter allows the designer to restart from a certain point back in time. The interpreter allows for two types of backtracking, *complete* backtracking and *partial* backtracking. Both methods are based on *belief revision* [De Kleer, 1986b]). In the former case, all assumptions generated from a certain point in time will be removed, and the design process will recontinue from that point on. The unsuccessful sequence of scenarios is remembered by the interpreter. Repetition of the same undesirable sequence of scenarios can then be avoided by choosing different design goals. Recall that the designer will always be the one who actually selects the design goals. By the same token, it is the designer who initiates complete backtracking.

DM 2. *The ADDL interpreter allows for complete backtracking by adding a time stamp to assumptions and by storing a history of scenario sequences.*

In the latter case (i.e. partial revision), the designer decides that a certain assumption, done at a certain point in time will be rejected. The interpreter, then, removes all assumptions that depend on the rejected assumption. Assumptions depending on these are removed as well, and so on until all dependencies are removed. The interpreter continues the design process from the current state.

DM 3. *The ADDL interpreter allows for partial backtracking by maintaining a dependency tree of assumptions.*

The difference between the two types of backtracking it that with complete backtracking all assumptions asserted after a retracted assumption are removed regardless of a dependency tree. With partial backtracking only the assumptions that depend on a retracted assumption are removed. In the first case the interpreter 'steps back' in time, while in the second case it does not. While the interpreter corrects the obvious designer 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.

4.2.2 The fact-base and the object-base

While scenarios encode the design process representation staticly, the fact-base and the object-base represent the object information state dynamicly. During conceptual design an *abstract anatomical* description of the design object is constructed. The description consists of *entities* and *relationships* among entities. The design object description represented by the entities and the relationships among these is called a *meta-model*. It is a qualitative model of the design object describing its intended function and behaviour. The relations and entities are represented in the fact-base in the form of first order propositions because they allow for a flexible representation [Lloyd, 1987]. A predicate symbol denotes a relationship and its arguments (*constant* terms) refer to the entities.

DM 4. *ADDL should have a fact-base to store the meta-model in the form of first order propositions.*

The abstract anatomical description is transferred to a *concrete anatomical* description while the design proceeds. The latter describes the design object as a structural decomposition, i.e. an assembly, of entities. These entities have attributes and certain operations attached to them. They are represented as objects in the object-base. Each object has a unique name and a constant term in the fact-base refers to the name of an object in the object-base. Only scenarios can access the object-base or the fact-base.

DM 5. *ADDL should have objects that assemble a design object description stored in an object-base.*

4.2.3 Intelligent user interface

The IIICAD system must be a tool that allows designers to construct expressions that give them control over the behaviour of the system. The system must avoid getting in the way of designers, i.e., the designer should not unnecessarily be hindered by the system. This stipulation leads to a number of conditions that the system, or rather the IUI, must fulfill.

- The designer and *not* the system determines the way the design process is directed.
- The IUI must allow the designer to express his ideas in his own terminology.
- The designer must always be informed about what the system is doing and what has been achieved so far.
- The IUI must adapt to the level of expertise and experience of the designer.

The communication between the IUI and the ADDL interpreter is accomplished through special scenarios that carry out the instructions given by the designer. The IUI itself is not written in ADDL and its actual design and implementation goes beyond the scope of this dissertation. I refer the interested reader to

[van Klarenbosch, 1991] for a description of the IUI. For ADDL it is important that it can handle tasks and instructions given by the IUI and that it can provide information for the IUI.

DM 6. *ADDL should have constructs to maintain a dialogue with the IUI.*

4.2.4 External applications

The object information state describes the design object as the design proceeds ranging from an abstract anatomical structure to an exact anatomical structure. At certain stages of design it needs to be evaluated in a certain context through an *aspect* model. An aspect model can be a geometric model, a kinematic model, a dynamic model, a mathematical model and so on.

An external application is a separate system attached to the IIICAD system. It uses ADDL data about the design object description as input to generate an aspect model. The external application produces some data, which are transferred back to ADDL. The EAI secures the mapping between the object information state and aspect models generated by external applications. An application program is not necessarily written in ADDL. The EAI is capable of translating the information of the object information state into code understandable by a certain modeler.

DM 7. *ADDL should have constructs to interface to an external application.*

The relationships between the design maxims encountered so far are outlined in Fig. 4.2. This tree structure comprises rectangular and rounded boxes. The former represent the system's components. The latter stand for ADDL design maxims. The keywords appearing in transparent boxes are abstract requirements that the system or the language must fulfill. They can be implicitly retrieved in either the IIICAD system or ADDL. Those in gray boxes refer to derived language constructs. They are explicitly present in the language specifications. The number appearing in a circle identifies the design maxim. The boxes are connected with arrows. In the sequel, I call a box that has a leaving arrow an *original* box, and one that has an arriving arrow a *terminal* box. There are two types of arrows, viz. *has-component* and *makes-use-of* arrows (respectively indicated by ◎ and ⊕). The semantics of the former is that the construct described in the terminal box is a part of the construct described in the original box. The latter means that the construct described in the original box uses the one in the terminal box.

4.3 Specification of the design process

The following two sections give the design criteria for a ADDL. The IIICAD system will be implemented in this language. The previous section presented ADDL's criteria imposed by the IIICAD system architecture. This section gives the design maxims concerned with the design process representation. Special language constructs are needed to represent design knowledge in order to implement the

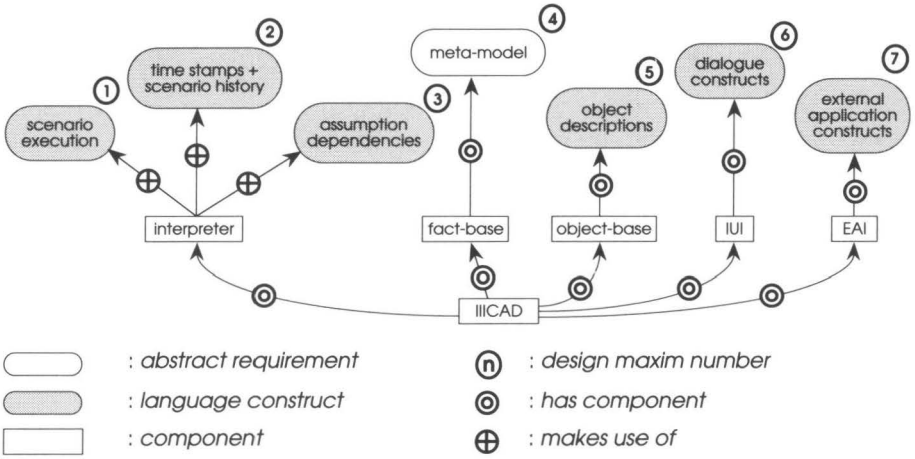


Fig. 4.2 IIICAD system requirements.

IIICAD system, that inhabits a meta-model and a process model based on stepwise refinement.

DM 8. *ADDL should have constructs to describe not only design objects but also design processes.*

The next section (§ 4.4) provides the design maxims concerning the design object representations.

4.3.1 Description of the stepwise nature of the design process

Designing is a process that refines a design object model step by step. This model gradually evolves from an incomplete to a detailed description. A solution to a design problem is thus obtained by stepwise refinement rather than by direct mapping from the specifications. Therefore, I need a construct in ADDL to model an intermediate description of the design object. A design step is performed by the execution of a *scenario*. For each incomplete state of a design object a scenario appropriate to that state is selected and executed. It contains the design knowledge necessary for refining the design object description. A scenario consists of rules and operations applicable to the state involved.

DM 9. *ADDL should have a scenario construct to describe the stepwise nature of the design process.*

I distinguish two levels of the design process in considering the designer's mental activity [Takeda *et al.*, 1990; Brumsen, Pannekeet, and Treur, 1992]. On the one hand there is the *object-level*, at which the designer thinks about the design objects

themselves, e.g. what properties the design object has, how it behaves in a certain context, and so on. On the other hand there is the *meta-level*, at which the designer thinks about how to proceed with the design, i.e., what he should do next. Therefore, I need two types of scenarios, viz. meta-level and object-level scenarios. Meta-level scenarios evaluate the current state of the design process (*process information state*), and assert design *goals* to be solved in order to obtain more refined description of the design-object.

DM 10. *ADDL should have meta-level scenarios to evaluate the process information state and to choose design goals.*

Object-level scenarios evaluate the object information state. They assert new literal facts about the design object to the fact-base or they assign values to objects' attributes. These literal facts and values are derived from the design knowledge incorporated in the scenario.

DM 11. *ADDL should have object-level scenarios to evaluate the object information state and to add new object information.*

Design goals stated by meta-level scenarios can either be solved by object-level scenarios or by meta-level scenarios. The former may add information to the object information state while the latter may add information to the process information state.

A design step is performed by the application of the knowledge embedded in a scenario. Since designing is regarded as a stepwise refinement process, forward reasoning seems to be the proper inference mechanism. The design knowledge is represented by means of IF-THEN rules and the inference strategy is forward chaining. [Davis, Buchanan, and Shortliffe, 1977; Davis and King, 1977]. A rule consists of an *antecedent* and a *consequent*. The intuitive meaning of a rule is: "if the antecedent holds true, then it is reasonable to assume that the consequent holds true as well." Rules have a purely declarative meaning.

DM 12. *An ADDL scenario should incorporate a collection of IF-THEN rules and a forward chaining inference engine.*

4.3.2 Meta-level scenarios

The knowledge embedded in a meta-level scenario is applied to add information about the design process state. The knowledge represented by meta-level scenarios is also described by IF-THEN rules. During the course of a design process the design object description changes continuously. Recall that object information state represents the current state of the design object. For controlling its stepwise refinement there is a need to express status information about the literal facts that represent it. The status of the literal facts is represented by process parameters. They are used both to query and assert information of the process state of literal facts.

DM 13. *ADDL should have process parameters to represent the design process status of literal facts, viz.* `abstract`, `concrete` *and* `detailed`.

Depending on the object information state and the process information state a scenario is selected. The scenario will contribute to the information states and hence the design object model will be refined. The selection of scenarios is done by means of the meta-predicate symbol `goal`. The assertion of a goal predicate states a new design goal that needs to be solved.

DM 14. *ADDL should have meta-predicate symbols to assert design goals.*

With the selection of scenarios in ADDL, the designer creates a *context*, in which the design object is modeled. An example of a context might be a geometric representation of the design object. Extending the geometrical representation of the design object is the use of such a context. Another example of such a context is a kinematic aspect model, in which the motion of the design object is modeled. A context allows a designer to focus on a specific part of the design object. Parts of the design object that are irrelevant at that stage of the design process are kept hidden.

DM 15. *ADDL should have a mechanism to activate a scenario that looks only at a subset of the object information state. The subset is relevant to the context it describes.*

However, a designer should be allowed to model multiple aspects of the design object simultaneously. This amounts to two or more scenarios being active at the same time. They work on the same design object description.

DM 16. *ADDL should have constructs to activate multiple scenarios at the same time.*

Meta-level scenarios allows a designer to choose a next design goal. Quite often such a choice is not uniquely determined, and a designer may want to model different alternatives simultaneously. Concurrent scenarios enable a designer to model alternative information about the design object. They may result in different design solutions.

DM 17. *ADDL should have constructs to activate concurrent scenarios.*

4.3.3 Object-level scenarios

The knowledge embedded in an object-level scenario is applied to extend the information about the design object's state. The antecedent of a rule checks whether the rule is applicable to the object information state. The consequent depends on the antecedent, it add information to the object information state. The antecedent can query either the object information state, or the designer (the IUI), or an external application (the EAI).

DM 18. *ADDL should have constructs to query the object information state, the designer, and an external application.*

The consequent is the part of a rule that follows logically from the antecedent. Two kinds of statements can be made by a consequent. It can (i) extend the meta-model by asserting literal facts, and (ii) it can refine the design object description by assigning values to objects' attributes.

DM 19. *ADDL should have constructs to assert literal facts and to assign attributes.*

The directed acyclic graph in Fig. 4.3 shows the design maxims concerned with the design process representation. The symbols in the figure have the same meaning as for Fig. 4.2.

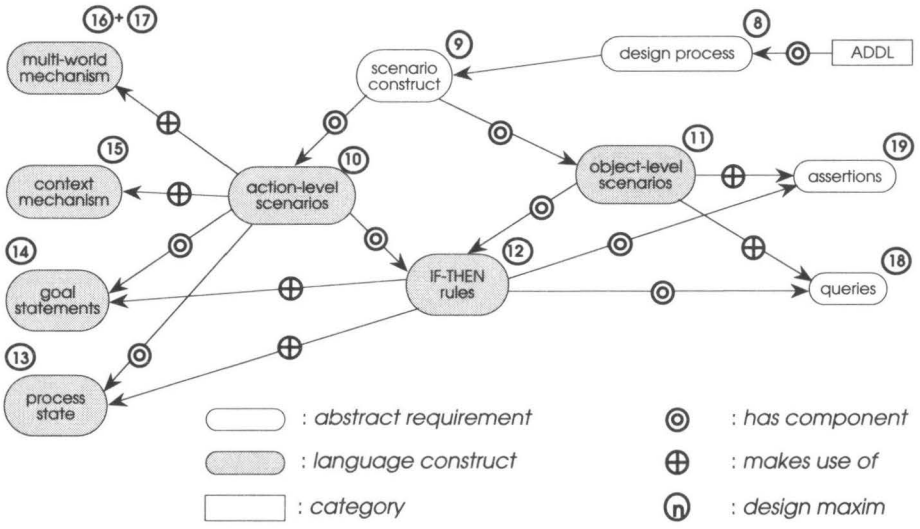


Fig. 4.3 Design process requirements.

4.4 Specification of the design object

Thus far, I have given language constructs for the specification of the design process. This section introduces ADDL constructs for the specification of the design object. Since design is regarded as a mapping form function space onto attribute space, it requires ADDL to have both attributive and functional representations. There are several issues in representing attributive information. First of all, an attribute represents the value of a certain property of an object. They define the *internal* properties, e.g. the height of a table. Secondly, attributive information

refers to the structure of an entity, e.g. a leg of a table. The structure of an entity is characterized by a decomposition of the object into sub-structures and by relationships among sub-structures. The structural decomposition defines the *external* properties of an object.

DM 20. *ADDL should have constructs to describe both the internal and external properties of objects.*

The external properties of objects are stored in the fact-base, the internal properties of objects are stored in the object-base. All system components (e.g. scenarios, the object-base, the fact-base, and so on) refer to an object by its (unique) name.

DM 21. *Each ADDL object should have a unique reference.*

4.4.1 The meta-model

The ADDL meta-model describes the external properties of objects. It is used as a central model for the design object representation, from which aspect models can be derived. Thus, it describes the design object in terms of function and behaviour. The meta-model is represented by first order propositions that consist of objects and relationships among objects. In ADDL these propositions are called *literal facts*, and they are stored in the fact-base.

DM 22. *ADDL should ideally be based on first order predicate logic to specify literal facts about objects.*

The entire design object is composed of several objects, which in turn are decomposed. This part-whole hierarchy is represented in ADDL by a binary built-in predicate symbol `hasPart`. It denotes a relationship between two objects: the latter is a part of the former (e.g. `hasPart(pinion1,pin1)`). The whole physical object decomposition is tied up by the `hasPart` predicate symbol. A class of predicate symbols with a special meaning is the set of unary object *instantiation* predicate symbols starting with `is` followed by the capitalized name of a type, e.g. `isPin()`). Upon assertion it instantiates its argument to an object of its type. In ADDL, there is a library of prototype descriptions that are used as templates for object creation. Each instantiation predicate symbol has a attached procedure that creates a copy of such a prototype description. The issue of instantiation will be discussed in detail in the next section. Another example of a built-in predicate symbol is `value`. The assertion of a such a literal fact assigns a value to an object's attribute.

DM 23. *ADDL should have a number of built-in predicate symbols with a special meaning obtained by an attached procedure.*

4.4.2 ADDL objects

The meta-model describes the relationships among objects, i.e., the external properties. The literal facts and rules act together as a deductive data-base. The internal properties of an object describe the specific properties of an object itself. It consists of *attributes* and *operations*. The object-base acts as an object-oriented environment. Attributes and operations are equivalent, respectively, to instance variables and methods in object-oriented terminology [Wegner, 1990]. The operations of an object share a state that is formed by the object's attributes. The names of the attributes and operations determine the *functions*, to which an object can respond. The collection of functions that can be applied to an object determine the object's interface and its behaviour. They bridge the gap between the fact-base and the object-base. Functions are similar to messages in object-oriented languages.

DM 24. *ADDL objects should have a collection of attributes and operations that represents its internal structure.*

DM 25. *ADDL should have data abstraction, the object's internal structure can only be accessed through functions.*

For the construction of a design object model a designer employs so called 'building blocks' [Hayes, 1979]. 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* [Lieberman, 1986]. Prototypes serve as templates, from which objects are created. When during fundamental design the concrete anatomical structure is created, it is made by copying prototypes from the prototype library. I call this copying the instantiation of an object. Whereas the attributes of a prototype are copied to the instantiated object, the operations of the prototype are shared by all objects instantiated from the same prototype. An object is instantiated in the fact-base by a built-in predicate.

DM 26. *ADDL should have a prototype library that is used for the instantiation of objects.*

During design, it may be desirable to modify an object's internal structure. Due to the absence of a proper prototype, a designer may wish to add/delete attributes or operations to/from the instantiated object. In case of attributes it is simply a matter of adding/deleting the attribute and its access function. Where operations are concerned, deletion is a matter of removing the operation and access function, whereas for adding an operation a designer must be given an interface to write such an operation using his own terminology.

DM 27. *ADDL should have a mechanism to dynamically modify an object's internal structure.*

There are several reasons for using multiple prototype definitions to instantiate a single object. First of all, it might be the case that for a given design problem a suitable prototype definition cannot be found. In many cases such a description can be made by merging different prototypes into a single description. Secondly, in order to reduce the size and the number of prototype definitions it is vital to use multiple prototype definitions. For instance, when a designer creates a lever to build a linear motion mechanism, he wants it to behave as a lever as well as a motion mechanism. Thus instead of having a separate prototype definition for an object that has the functionality of both a lever and a motion mechanism, the designer can combine two prototype definitions.

The third and most important reason is related to the concept of aspect models. Since multiple aspects of the same design object need to be highlighted during the design process, objects must have multiple representations. For example, geometric information about the above mentioned lever can be made available by further instantiating it as a geometrical object, e.g. as a block. The same mechanism applies when a dynamic model must be made by instantiating it as physical object, e.g. a lever, and so on.

DM 28. *ADDL should allow for multiple typed objects.*

A non-trivial design problem easily results in a design object model that consists of an enormous number of objects. Many of these objects have some properties in common while other properties differ. Objects in ADDL that share properties with a particular prototype, but have some extra properties added to them are called a specialization of that particular prototype. The prototype library contains a hierarchy of prototype definitions. Prototypes are defined in terms of other prototypes. Therefore, if a prototype is a specialization of another prototype, the former inherits properties from the latter [Cook, 1987]. This allows the system designer to reuse previously defined code and to specialize a certain prototype.

DM 29. *ADDL should have constructs for the definition of a prototype hierarchy, and for an inheritance mechanism.*

In Fig. 4.4, the design maxims related to the design object representation are depicted. The arrows have the same meaning as those in the previous figure. The design maxim concerning functions (no. 24) plays a special role. It has only *makes-use-of* arrows attached to itself, and no *has-component* arrows. The reason for this stems from the role of a function as an interface between the part of ADDL describing the design object and the part describing the design process. In the next section, I merge the three diagrams into a single coherent diagram describing the relationships amongst all design maxims.

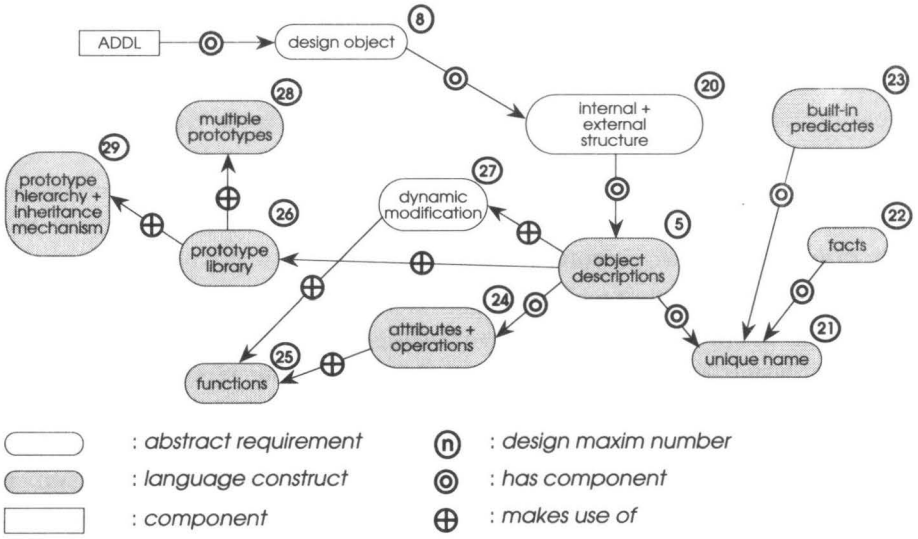


Fig. 4.4 Design object requirements.

4.5 Discussion

In the preceding sections, I have encountered twenty-nine design maxims (**DMs**). The derivation of ADDL specifications took place as follows. First, I classified them into several functional components. These components were: *system architecture, design process specification,* and *design object specification.* The first component is subdivided into the five system components shown in Fig. 4.1: *interpreter, fact-base, object-base,* IUI, and EAI. The latter two components form the ADDL language specifications. These components are shown in Fig. 4.5. It represents a directed acyclic graph showing the relationships between the counted **DMs**.

The **DMs** have been distributed over these components representing them by keywords, and established links between them. Fig. 4.5 shows the relationships among **DMs**, keywords, components and derived language constructs. In this figure, the small circles correspond to **DMs**, rounded boxes are keywords, and the rectangular boxes are components. The keywords appearing in transparent boxes are abstract requirements that the system or the language must fulfill. They can be implicitly retrieved in either the IIICAD system or ADDL. Those in gray boxes refer to derived language constructs. They are explicitly present. The arrows are either *has-component* or *makes-use-of* relationships.

It is interesting to note that the design of ADDL has proceeded in accordance with DPM. The graph shown in Fig. 4.5 has looked quite different during previous

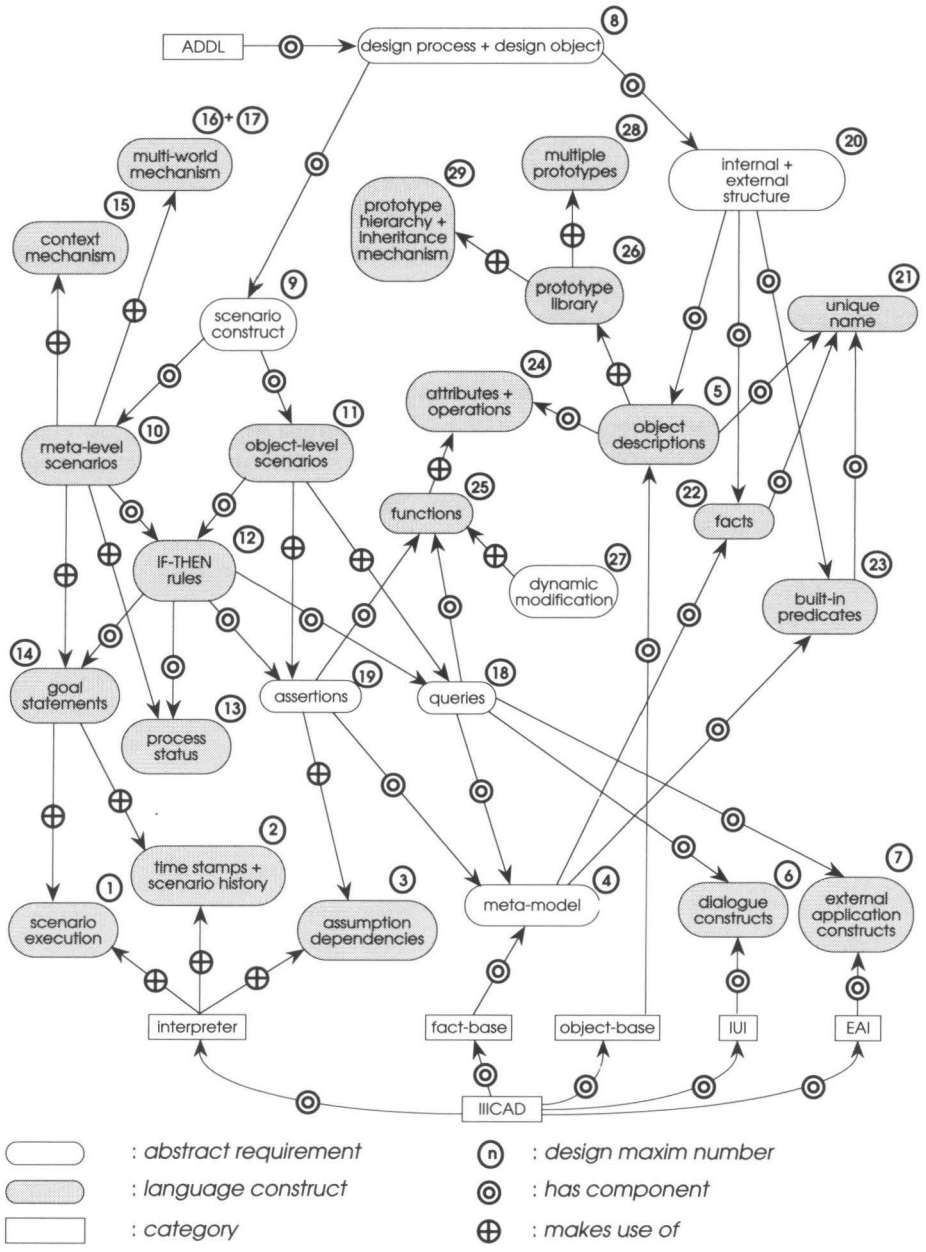


Fig. 4.5 Classification of ADDL design maxims.

stages of its development. Parts of it were missing and other parts looked entirely different. An overview of the design history of ADDL (and formerly IDDL) can be obtained by comparing the successive papers [Veth, 1987; Veerkamp *et al.*, 1989; Veerkamp, Pieters Kwiers, and ten Hagen, 1991; Tomiyama, Xue, and Ishida, 1991; Xue *et al.*, 1990]. The difference is especially notable when comparing Veth's paper [Veth, 1987] and this chapter. Both have a similar structure, but the former presented a number of **DMS** that a future implementation should meet. While the **DMS** presented in the latter reflect the current implementation of ADDL.

For example, the concept of *modal operators* [Hughes and Cresswell, 1972], which seemed to be an important feature of IDDL has completely disappeared from the current implementation. They have been replaced by *meta-predicate* symbols. The reason stems from the need to make a clear distinction between process and object knowledge. In an early implementation these two kinds of knowledge were mixed at the same level. But during the course of writing a serious application, this mixture made the maintenance and understanding of ADDL code very hard. The implementation of a meta-level architecture was a natural consequence of the decision to split the representation of object and process knowledge. A sub-graph of the entire graph will now be explained in detail. The remainder of the graph can be understood by analogy. Fig. 4.6 shows the sub-graph centered around **DM** 24. The box has two incoming *has-component* arrows and a single incoming *makes-use-of*. (From the point of view of the box it must be read *is used by*. The box has a single outgoing arrow of the type *make-use-of*. In other words this can all be translated as:

> Both assertions and queries are (partly) composed of functions. In order to use dynamic modification one needs a function. Finally, attributes and operations can only be accessed by functions.

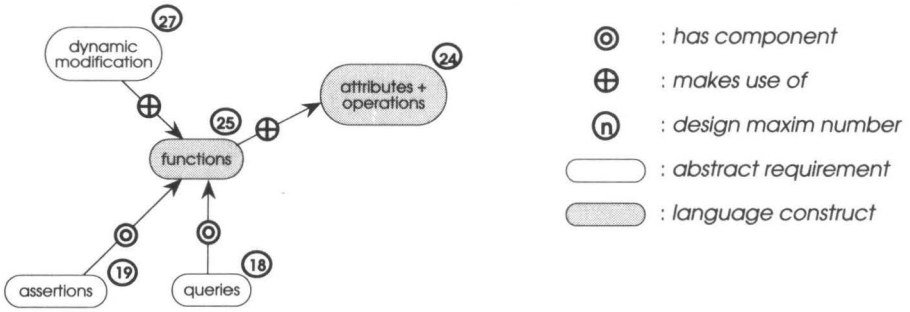


Fig. 4.6 Sub-graph concerning functions.

When taking a closer look at the entire graph, it can be observed that the box containing the keyword *functions* plays a central role. It is the only part of the sub-graph describing the design-object that is directly connected to the part describing the design process. This stems from the requirement that design object's internal structure may only be accessed by functions. Hence there is no other direct link between the design object representation and the design process representation. Hence, ADDL objects have strong *encapsulation*, they are protected against external access. In object-oriented terminology it is said that ADDL supports *data abstraction*. Potential conflicts between literal facts about an object and its internal properties must be solved by the knowledge embedded in the rules of a scenario. They can access both the object-base and the fact-base, and thus an object's internal and external properties. The concept of data abstraction shows up in Fig. 4.5, since there is only a single *makes-use-of* arrow arriving at box no. 23 (attributes + operations). The *meta-model* plays a similar role concerning the object's external structure.

4.6 Conclusions

This chapter presented a unifying framework for representing design knowledge. The starting point, DPM, inspired me to formulate design maxims that are converted into ADDL specifications. The model enabled me to understand, clarify, model, and formalize design process and design object knowledge in an intelligent CAD environment.

The design maxims can be grouped into three functional components, *system architecture*, *design process* representation, and *design object* representation. Each component was represented by a directed acyclic graph. Such a graph shows the design maxims belonging to a component and the relationships amongst these. The three graphs were merged into a single graph representing the full functionality of ADDL and the IIICAD system. The next three chapters give an overview of the formal ADDL specifications that were a result of the design maxims presented in this chapter.

5

Representation of Objects in ADDL

5.1 Introduction

The knowledge representation and processing language ADDL (Artifact and Design Description Language) presented in this chapter aims at implementing Computer Aided Design systems (CAD systems). Every human controlled production process for some artifact is containing numerous design tasks. Only very few of those are supported by computerized design tools. ADDL contains new constructs that support writing CAD systems for a wide range of design tasks. In the previous chapter, I distilled a number of design maxims that such a language must meet. These maxims have been translated into language specifications. This chapter and the next two present the ADDL specifications.

ADDL is basically a logic programming language. However, to facilitate an easy and flexible representation of objects some constructs from the object-oriented programming paradigm are embedded in ADDL. An essential property of CAD systems is the ability to represent a complex artifact. In ADDL, an artifact is decomposed into a large set of objects that represent its parts. Each ADDL object has a private state that can be accessed through functions and it has a type that defines the characteristics of the private state. The object types are classified in a type lattice. This is in short a simplified characterization of the object-orientedness of the language. Relationships among these objects can be defined in a deductive database with only unit clauses, called the fact-base. Reasoning about these objects and relationships among them is performed by a rule-based knowledge base. This knowledge-base is modularized by a set of *scenarios*. A scenario consists of a collection of IF-THEN rules.

Chapter 4 presented the design maxims in a top-down manner. Initially, the language concepts at the highest conceptual level were presented. Then, these concepts were detailed till the basic design maxims showed up. The language specifications are organized in an opposite fashion, i.e., bottom-up. I start by giving the basic language constructs. Thereafter, I gradually increase the level of complexity. This approach is commonly used in books on programming languages and it has proven to be successful.

The components of ADDL consist of a *static* part and a *dynamic* part. The dynamic part consists of an *object-base*, a *fact-base* and a set of *process parameters*. The static part consists of the *object knowledge* representation and the *process knowledge* representation. Fig. 5.1 shows the components. The object-base stores the ADDL objects that constitute a model of the artifact during the design process. The fact-base contains the relationships among these objects. The object-base is embedded in the fact-base. The process parameters describe the current state of the design process. The object knowledge representation is used for *object-level reasoning* about the model of the artifact. The process knowledge representation is used for *meta-level* reasoning about the state of the object-level reasoning. It controls the object-level. The arrows between meta-level and object-level reasoning indicate a flow of control while the other arrows indicate a flow of information.

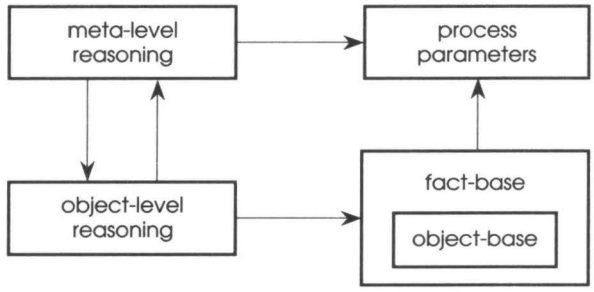


Fig. 5.1 The static and dynamic components of ADDL.

The representation of objects is the subject of this chapter. Chapter 6 and Chapter 7 discuss object-level and meta-level reasoning respectively. The next section presents the object-base. Since Chapter 6 gives a formal specification of the fact-base, § 5.3 only gives an informal introduction to the fact-base. Chapter 7 gives a formal specification of the process parameters. In the sequel, I refer to the design object by *artifact* in order to prevent confusion with an ADDL object. The artifact is represented by means of objects and relationships among these objects. The object structuring is explicitly obtained by asserting relationships. These

relationships together make up an object's external properties. Equally, objects have internal properties that are made up by attributes and operations.

5.2 The object-base

The artifact representation is composed of a single uniform data structure, an *ADDL object*. An ADDL object consists of one or more attributes (the data) combined with a set of operations for manipulating that data. The attributes can have values which define an object's internal state. The interface to an object's internal state is accomplished through *functions*. ADDL attributes, operations, and functions can respectively be compared with instance variables, methods, and messages in an object-oriented language such as Smalltalk-80 [Goldberg and Robson, 1983; Goldberg, 1984]. An operation consists of a *selector* and a *body* containing the operation's code. A selector is represented by the selector's denominator and one or more argument(s) between parentheses, e.g. `distance(point1,point2)`. The body of an operation is executed when a function denoted by the selector is applied to an object. The first argument is the object to which the function is applied. The code of a body is simply Smalltalk-80 code. Since I did not want to reinvent the wheel, I tried to take as much advantage as possible of the underlying programming environment.

In their modeling task CAD systems need to establish structures. Objects have particular properties which make it possible to treat them as either manipulable entities (constants) or to extract information from them. Every object has i) an *object name* and ii) an *object type*. The former serves as a reference to the object-base and is used as a constant symbol in the logical language, the latter is a reference to the object's *prototype* definition. Such a prototype serves as a template to build up the object's internal structure in the object-base. Prototypes are further treated in §5.2.2. There are two kinds of objects: viz. *primitive objects* and *composite objects*. The former are nothing but the value that they represent while the latter have an internal structure, i.e., attributes.

5.2.1 Primitive objects

Primitive objects are the building blocks of the object-base, they are recognized by their value. For example, `8`, `3.14` and `'foo'` are primitive objects. ADDL provides four types of primitive objects: *number*, *symbol*, *string*, and *array*. Opposite to that of composite objects, the set of primitive object types can not be extended by the system programmer. Each type of primitive object has its own set of operations to which the object responds. The four types are separately discussed below.

Number: Numbers in ADDL are represented in the usual way. An example of a number is `4`, `-456.88` or `1.23e2`. Objects belonging to the type number respond to the following operations. Note that for convenience an infix notation is used for the well known arithmetic operators:

Selector	Comment
+	addition
-	subtraction
*	multiplication
/	division
**	raised to
abs()	returns the absolute value of the number
gcd(,)	Returns the greatest common divider of the first and the second argument.
negated()	Returns the negative value of the number. If the number is negative, it returns the positive value.
sin()	Returns the sine value of the number.
sqrt()	Returns the square root of the number.

This list is not complete. It gives some insight what kind of operations one can expect for numbers.

Symbol: Symbols are words starting with a lower-case letter. It can contain letters but punctuation is not allowed. Examples of symbols are: `foo`, `slot1`, and `y45I71`. Symbols ar reserved for names of composite objects. They have no operations defined on them. A special symbol is used to denote an undefined object, viz. `nil`. It represents the null value given to attributes that have not yet received a value.

String: Strings are sequences of characters enclosed by single quotes. A quote can be included in a string by preceding it by a quote. Examples of strings are: `'qwerty'`, `'length of slot'`, and `'Tom"s house'`. The following operations on strings are defined:

Selector	Comment
atPut(,,)	Puts the third argument at the second argument's position in the string denoted by the first argument.
add(,)	Returns a copy of the first argument concatenated with the second one.

Array: An array is an indexable number of objects of a fixed size. An array of size `n` is represented by a *hash* (`#`) followed by `n` objects between parentheses separated by spaces. The index starts at one. An element of an array may be an array as well. Examples of arrays are: `#(1 2 3)`, `#(foo bar 12)`, and `#(#(1 22) #(12 13 14) 3)`. The following operations are applicable to arrays:

Selector	Comment
`at()`	Returns the element at the argument's position in the array.
`atPut(,,)`	Puts the third argument at the second argument's position in the array denoted by the first argument.
`first()`	Returns the first element of the array.
`indexOf(,)`	Returns the index of its argument in the array. It returns `0` if the argument is not present.
`last()`	Returns the last element of the array.
`size()`	Returns the number of elements of the array.

The observant reader has already noted that none of the above types have operations that perform a comparison, such as `greater(,)`, `equal(,)` and so on. This is due to the fact that functions in a logical language never return a truth value as result. The evaluation of (primitive) objects is done by built-in predicate symbols. This mechanism will be discussed in Chapter 6.

5.2.2 Composite objects

The second kind of objects is a composite object. The previous section stated that primitive objects are recognized by their value. Composite objects are identified by a unique name, viz. a symbol. A composite object may have next to a number of operations a number of attributes. They represent properties of the object. The attribute names are symbols and their values are restricted to primitive objects. This restriction stems from the requirement on flexible object representations (see Chapter 4 **DM** 22) that the structuring of the artifact must be represented in the fact-base and *not* in the object-base. The part-whole hierarchy of composite objects is therefore composed by a built-in predicate (see §5.3.5). For a discussion on the representation of a part-whole hierarchy in an object-oriented system, I refer to [Blake and Cook, 1987].

During the design process, a composite object is instantiated by taking a copy from a prototype that serves as a template. The definition of a prototype consists of five fields: *prototype*, *name*, *parent*, *attributes*, *objects*, and *operations*. The definition looks like:

```
type object-type
name object-name
parent parent-type
attributes 'attribute-name1 attribute-name2 '
objects ''
operations 'selector1 selector2 '
```

The first field represents the prototype's **type**. The prototype's field **name** is filled in upon instantiation of the object. Another hierarchy *is* represented in the object-base, viz. the *is-a* hierarchy. All prototypes are organized in a hierarchy of specialization. At the root of the tree is the prototype `composite`. A prototype is a

direct descendant of the prototype mentioned in the field **parent**. A prototype inherits the attributes and operations of its parents, grand-parents, and so on.

The field **attributes** stores the attribute names of the composite object. When an object is instantiated from a prototype, its attributes can be accessed by an operation whose selector is a colon followed by the name of the attribute. For example, when a prototype has an attribute `length`, then it has an operation `length()` as well. I am very much aware that this mechanism somehow violates the principle of encapsulation. This is due to the fact that this mechanism allows for access of an object's private state. For convenience, I adopted this strategy, however, for a future version of the language I consider making a distinction between *public* and *private* attributes. The former have implicit access operations while for the latter access operations must be created explicitly. In the current version, the implicit definition of an operation accessing an attribute is overridden when such an operation is explicitly defined. For instance, when a prototype has both the attribute `foo` and the operation `foo()` in its definition, the latter is used to access the attribute value.

The field **objects** is initially empty in the prototype definition. It is used during the lifetime of an instantiated object to store object names being part of the object. This structuring is defined in the fact-base by the binary built-in predicate `hasPart` denoting that the second argument is a part of the first one, but it can also be useful for an object to know about its parts. Therefore, when a built-in predicate `hasPart` is asserted to the fact-base, the name of the second argument is added to the field **objects** of the object denoted by the first argument. This mechanism is used when an object wants to propagate a message to its parts. For instance, when a geometrical object gets the instruction to draw itself, it may propagate this instruction to its components as well.

The field **operations** contains the selectors of the operations being applicable to the prototype. How and when the body of an operation is defined will be discussed in Chapter 8 on the implementation. Owing to the inheritance mechanism, an ADDL object can be accessed by the operations defined by its prototype and all of its parents. When an inherited operation does not have the desired functionality, it can be redefined by a child's prototype. Hence, when multiple operations with the same selector are defined along a path in the hierarchy, the one appearing nearest in the hierarchy is chosen.

At the root of the prototype hierarchy is the prototype `composite`. All other prototypes are descendants of `composite`. Its definition is as follows:

```
type composite
name nil
parent nil
attributes ''
objects ''
```

```
operations 'name() parent() addAttribute(,) addOperation(,)
    removeAttribute(,) removeOperation(,) '
```

It is evident that the prototype composite does not have a parent. It has no attributes either. The operations defined on `composite` are those which are valid for all composite objects. They describe a general interface for all objects. Each of the operators is explained below:

Selector	Comment
`name()`	Returns the name of the object.
`parent()`	Returns the object's parent.
`addAttribute(,)`	Adds the argument to the list of attributes.
`addOperation(,)`	Adds the argument to the list of operations.
`removeAttribute(,)`	Removes the argument from the list of attributes.
`removeOperation(,)`	Removes the argument from the list of operations.

The last four operations may need some explanation. They stem from the design maxim on dynamic modification (see Chapter 4 **DM** 26). Modifying an object's internal properties is their purpose. Note, however, that adding a selector to the list of operation does not always suffice. When a proper body for the operation is not yet present in the system, an interface to the designer is opened. It allows him to edit the operation's body. This issue will further be explained in the Chapter 8 on the implementation.

Since ADDL is an empty shell in which intelligent CAD systems can be implemented, `composite` is actually the only prototype present in the language. The ADDL programmer has to build up a hierarchy of prototype definitions next to his programming task. An example of such a system is presented in Chapter 9. For instance, the prototype `point` is frequently used for describing an object's geometry. Its definition is:

```
prototype point
name nil
parent composite
attributes 'x y '
objects ''
operations 'distance(,) '
```

The definition is rather trivial to comprehend. The attributes represent the point's coordinates, and the operation computes the distance between the point self and the point which is given as an argument. The complete prototype library for the example design system in Chapter 9 is given in Appendix 3.

5.3 The fact-base

The fact-base is used for representing relationships among objects. The next chapter gives a formal specification of the fact-base. However, since the fact-base defines the decomposition of the objects appearing in the object-base, a short introduction may enlighten the reader. The fact-base acts as a (deductive) database [Minker, 1988], and it is built of *literal facts*. A literal fact is either a unit clause as defined in [Lloyd, 1987; Clocksin and Mellish, 1981] or the negation of a unit clause. In order to define literal facts I first introduce *predicate symbols*. A predicate symbol is a symbol, e.g.

```
isLever,
hasPart,
material.
```

A literal fact is either a *positive fact* or a *negative fact*. A positive fact is a predicate symbol followed by list of primitive objects separated by commas and enclosed by parentheses, e.g.

```
isLever(lever1),
hasPart(lever1,slot1),
material(lever1,'metal').
```

A negative fact is as might come up to expectation a negation of a positive fact. It is a means to express negative information about the artifact. The next chapter discusses issues concerning the consistency of the fact-base. Examples of negative facts are:

```
~canFly(penguin).
~adjacent(lever1,pin1)
```

Note that the terms of a literal fact are primitive objects. The symbols amongst them are names of composite objects. Hence, they are references to descriptions of composite objects in the object-base. Exceptions are names of types, such as `penguin` which is a sub-type of bird (see the discussion §5.4). In the example both `lever1` and `slot1` are (unique) composite object names. The type of the primitive objects `'metal'` is a string.

5.4 Discussion

Both the fact-base and the object-base are initially empty, the contain respectively no literal facts and objects. During the execution of an ADDL program their contents gradually grows. Which amounts in adding literal facts and objects. However, the growth of the fact-base dictates that of the object-base. In other words, the only means to add an object to the object-base is to assert a corresponding literal fact to the fact-base. They query and assertion mechanism of both the fact-base and the object-base will be the topic of the next chapter.

The object-base is embedded in the fact-base. For each composite object occurring in the object-base there must be at least one positive fact in the fact-base. For example, suppose the object `foo1` of type `foo` occurs in the object-base. Then, the positive fact `isFoo(foo1)` must be present in the fact-base, because the built-in predicate symbol `isFoo` is used for instantiating the object `foo1`. A unique object name is created by the built-in predicate symbol `typeFor`. For example, the following expression generates a new name for an object of type `foo`: `typeFor(X,foo)`. The first argument `X` is bound to a new name that has the form `foo#` where `#` is a number. The way to instantiate an object is by the expression: `isFoo(X)` where `X` has been bound to a new name.

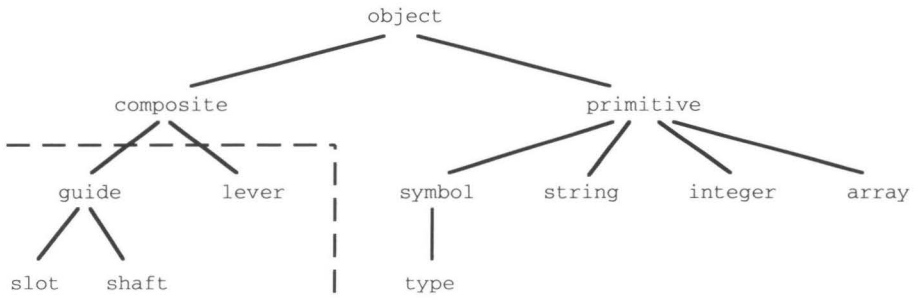


Fig. 5.2 Type hierarchy employed by ADDL.

In ADDL, the type hierarchy shown in Fig. 5.2 is employed. The types `object`, `composite`, `primitive` and the sub-types of `primitive` are hard-wired in ADDL. They are part of each ADDL application. An application programmer must add the appropriate sub-types of `composite`. They depend on the field of design and the kind of application. Fig. 5.2 shows four of them: the type `guide`, its sub-types `slot` and `shaft` and the type `lever`. An object of type `slot` inherits the attributes and operations of the prototype `guide`. In other words, `slot` is a *specialization* of `guide`. Chapter 6 introduces the notion of *generalization* which uses the type hierarchy in an opposite manner. The type hierarchy is used to query whether an object of a certain type or a sub-type of that type is present in the fact-base. For example, suppose the fact-base contains the literal fact:

```
isSlot(slot1)
```

and the query:

```
isGuide(X)
```

is posed to the fact-base. The query will match against the literal fact, since `guide` is a generalization of `slot`.

6

Object Knowledge Representation in ADDL

6.1 Introduction

The ADDL constructs dealing with the description of an artifact have been specified in Chapter 5. The representation of the knowledge about how to model such a description is the subject of this chapter. Design as a stepwise refinement process is represented by a number of IF-THEN rules which are applied one after the other. The rules contain information about the artifact description. The application of rules result in an extended description. The number of rules is already enormous for a rather straightforward design problem. Hence, there is a strong need for grouping the rules. Scenarios consist of a collection of rules which together represent the knowledge for performing a design step.

The logical part of ADDL consists of two separate first-order languages, an object-level language and a meta-level language. For the interested reader, [Lloyd, 1987; Apt, 1990] give the fundamentals of logic programming. Expressions in the object-level language make statements about an artifact, while expressions in the meta-level language say something about (the status of) these statements. Indeed, a meta-level language is a language about a language. The architecture of a system, which is based on both languages is called a *meta-level architecture*. This chapter presents the object-level language. The meta-level language is discussed in Chapter 7.

6.2 Object-level languages

The object-level language consists of *literal facts* and *rules*. The literal facts represent a state of the artifact. The rules represent propositions expressing logical relations among these literal facts. A rule is a piece of design knowledge that essentially has a declarative meaning. This section discusses the syntax of a first-order language. It is employed for the description of the rules in an object-level scenario. Chapter 7 presents a meta-language for representing rules appearing in a meta-level scenario. The syntax is very similar to that of standard logic. The only exceptions are the definition of single-level terms, antecedents and consequents. An alphabet, single-level terms, formulae, antecedents and consequents are subsequently introduced for the definition of rules.

6.2.1 DEFINITION: An *alphabet* consists of six classes of symbols:

1. *variables*,
2. *constant symbols*,
3. *function symbols*,
4. *predicate symbols*,
5. *connectives*,
6. *punctuation symbols*.

The symbols of the language are *order-sorted typed*, each ranging over a certain domain. Types are denoted (by convention) by the Greek letter τ. Variables are symbols beginning with an upper case letter (e.g. `X`, `Y`, `Slot`). A special instance of a variable is the *pseudo variable* 'ω'. It is used when the programmer does not care to which object a variable will be bound. Note that 'ω' is equivalent to Prolog's 'don't care' symbol ('_'). For each type τ, there is a pseudo variable ω_τ. Constant symbols are the primitive objects introduced in §5.2. Thus, `12`, `slot1`, `'size of slot'` and `#(1,2)` are constant symbols of type `number`, `symbol`, `string` and `array` respectively. A constant symbol of type `symbol` is either the name of a composite object or the name of a type. Names of composite objects end with a number. E.g. the symbol `slot1` refers to a composite object and the symbol `slot` refers to a type. A constant symbol's type is either defined by the primitive object that it represents, or it is (in case of a `symbol`) the type of the composite object to which it refers.

Function symbols are symbols starting with a lower-case letter[4]. By convention, I use the letters `f`, `g` and `h` for function symbols. Functions of arity n

[4] This chapter and the next adhere to this notation. Chapter 8 and Chapter 9 on the implementation use a notation with a less declarative reading that was more convenient to implement. A function `f(a,b,c)` is there denoted by `a:f[b,c]`, which has a more object-oriented reading.

have types such as $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$. Predicate symbols are also symbols starting with a lower-case letter. By convention, the letters `p`, `q`, and `r` denote predicate symbols. A predicate symbol with a zero arity is called a *proposition symbol*. Some predicate symbols have a predefined meaning, they are called *built-in predicates*. The full set of built-in predicates is given in §6.4.5. These include: `equal`, `isNil`, `value`, etc. A predicate symbol of arity `n` (`n` > 0) has a type such as $\tau_1 \times \cdots \times \tau_n$. A proposition symbol has type `nil`.

The connectives are limited to `&`, `|`, `~`, and → meaning logical *and*, *or*, *not*, and *implication* respectively. For the latter, I adapted the well-known notation **`IF`** ··· **`THEN`** ··· in order to improve readability. The punctuation symbols are '`(`', '`)`', and '`,`'.

Over this alphabet terms and formulae can be defined. The definition of a term progresses in two steps:

6.2.2 DEFINITION: A *simple term of type* τ is a variable or a constant symbol of type τ.

6.2.3 DEFINITION: A *single-level term of type* τ is defined as follows:

1. A simple term of type τ is a single-level term of type τ.
2. If `f` is an n-ary function symbol (`n` > 0) of type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$ and each t_i is a simple term of type τ_i, then `f(`t_1`,...,`t_n`)` is a single-level term of type τ.

In the sequel, I simply say term instead of single-level term. Thus, `X`, `123` and `'aString'` are simple terms and `f(X)` and `g(X,123,'aString')` are non-simple terms. They are all single-level terms. If the type of the function symbol `f` is $\tau_1 \rightarrow \tau$ then the type of the variable `X` must be τ_1. Note that the arguments of single-level terms are simple terms, i.e., there is no nesting of terms.

Using the definition of terms, literal formulae can be defined:

6.2.4 DEFINITION: A *typed atomic formula*, or in short an *atom*, is defined as follows:

1. If `p` is a proposition symbol, then `p` is an atom of type `nil`.
2. If `p` is an n-ary predicate symbol (`n` > 0) of type $\tau_1 \times \cdots \times \tau_n$ and each t_i is a term of type τ_i, then `p(`t_1`,...,`t_n`)` is an atom of type $\tau_1 \times \cdots \times \tau_n$.

6.2.5 DEFINITION: If ϕ is an atom, then both ϕ and ~ϕ are *literal formulae*.

By convention, I use Greek letters such as ϕ and ψ for formulae. The definition of a literal fact and a fact-base given in Chapter 5 can now be formalized in the following definitions.

6.2.6 DEFINITION: If `p` is an n-ary predicate symbol (`n` > 0) of type $\tau_1 \times \cdots \times \tau_n$ and each c_i is a constant symbol of type τ_i, then `p(`c_1`,...,`c_n`)` is a *positive fact*, and ~`p(`c_1`,...,`c_n`)` is a *negative fact*.

6.2.7 DEFINITION: A *literal fact* is either a positive or a negative fact.

6.2.8 DEFINITION: A *fact-base* is a finite set of literal facts.

Examples of literal formulae are `p(X)` and `~q(X,f(123),'aString')`, which can be transformed into literal facts by mapping the terms to constant symbols (§6.4), e.g. `p(a1)`, and `~q(a1,246,'aString')` are literal facts. The definition of a typed formula is:

6.2.9 DEFINITION: A *typed formula* is inductively defined as follows:

1. A literal formula is a typed formula.
2. If ϕ and ψ are typed formulae, then ϕ `&` ψ, and ϕ `|` ψ are typed formulae.
3. If ϕ is a typed formula, then `(`ϕ`)` is a typed formula.

6.2.10 DEFINITION: A *ground formula* is a typed formula with only constant symbols as terms.

The expression `p(X) & q(X,f(X)) | ~q(Y)` is an example of a typed formula.

The definition of rules is the next issue. Rules consist of an *antecedent* and a *consequent*. The antecedent is the condition of a rule and the consequent is the conclusion. The definition of an antecedent is equivalent to that of a typed formula while the definition of a consequent allows only conjunctions and no disjunctions. Thus, the binary connective *or* is absent in the definition of a consequent. A consequent and an antecedent are defined as:

6.2.11 DEFINITION: An *antecedent* is a typed formula.

6.2.12 DEFINITION: The definition of a *consequent* is equal to DEFINITION 6.2.9 with the restriction that the second induction rule is replaced by:

2′. If ϕ and ψ are typed formulae, then ϕ `&` ψ is a typed formula.

Recall that variables can occur in both the antecedent and the consequent. When an antecedent is evaluated with respect to some fact-base, its variables are replaced by constant symbols. The variables occurring in the consequent receive the same bindings. However, if a variable of the consequent does not occur in the antecedent, it can not receive a binding. Hence, the following definitions:

6.2.13 DEFINITION: Let ϕ and ψ be formulae and let χ_ϕ and χ_ψ be the set of variables occurring in respectively ψ and ϕ. Then the formula ψ is called *restricted to* ϕ, iff $\chi_\psi \subseteq \chi_\phi$.

6.2.14 DEFINITION: If ϕ is an antecedent and ψ is a consequent and ψ is restricted to ϕ, then $\phi \rightarrow \psi$ is a *rule*.

Because rules play such an important role in the language, it will be convenient to adopt a more readable notation for rules. Therefore, the keywords `IF` and **`THEN`** are used instead of the connective $\rightarrow$. In the sequel a rule looks like:

```
IF antecedent THEN consequent
```

6.2.15 DEFINITION: The *object-level language* given by an alphabet consists of the set of rules and literal facts constructed from the symbols of the alphabet.

6.2.16 DEFINITION: An *object-level scenario* <name, rule-set> is a finite set of rules that has a unique *name*.

Examples of rules are:

```
IF p(X) & q(a,f(b)) THEN r(X) & s(X,a,b)
IF p(f(X,a)) | p(f(X,b)) & q(h(X,c)) THEN r(X)
```

The informal semantics of a rule is "if for the assignment to a constant symbol of each variable occurring in the rule the antecedent holds, then the consequent holds as well". The next section will say more about the semantics of rules and scenarios.

6.3 Declarative aspects of object-level languages

6.3.1 Declarative semantics

This section discusses the truth or falsity of rules. The declarative semantics of the rules in the object-level language gives the meaning of a scenario. A scenario is defined as a set of rules. It has a domain associated with it, to which the rules are interpreted. Variables range over this domain and are assigned to a constant symbol. The terms are mapped to elements of the domain returning a constant symbol. The predicate symbols are assigned to relationships in the same domain. Thus, an *interpretation* gives a meaning to each symbol of a rule.

The semantics of a rule can inductively be defined by defining the semantics of formulae, antecedents, and consequents. The truth values of literal formulae can be determined, and consecutively the truth value of composed formulae. Since a design situation essentially deals with incomplete information, the classical pair *true* and *false* does not suffice. Therefore, to describe this completeness a third truth value *unknown* is introduced [Treur, 1989]. For the definition of truth values, the *Strong Kleene Truth Definition* is employed. For a discussion on three-valued logic, I refer to [Turner, 1984; Blamey, 1986]. The definition of an interpretation of the object-level language is:

6.3.1 DEFINITION: An *interpretation I* of an object-level language *L* consists of:

1. For each type τ, a non-empty set D_τ called the *domain of type* τ of the interpretation.
2. For each constant symbol c of type τ in *L*, the assignment of an element c_I in D_τ.

3. For each n-ary function symbol $\mathtt{f}$ of type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$ in L, the assignment $\mathtt{f}_I$ of a mapping from $D_{\tau_1} \times \cdots \times D_{\tau_n}$ to D_τ.

4a. For each n-ary predicate symbol $\mathtt{p}$ of type $\tau_1 \times \cdots \times \tau_n$ in L, the assignment of a subset $\mathtt{p}_I$ of $D_{\tau_1} \times \cdots \times D_{\tau_n}$.

4b. For each n-ary predicate symbol $\mathtt{p}$ of type $\tau_1 \times \cdots \times \tau_n$ in L, the assignment of a subset $\sim\mathtt{p}_I$ of $D_{\tau_1} \times \cdots \times D_{\tau_n}$.

4c. For each n-ary predicate symbol $\mathtt{p}$ of type $\tau_1 \times \cdots \times \tau_n$ in L, the subset $\mathtt{p}_I \cap \sim\mathtt{p}_I$ of $D_{\tau_1} \times \cdots \times D_{\tau_n}$ is empty.

Notice that in the declarative semantics, I restrict the partiality to predicate symbols; partial functions are not being used here. In other words, for each function there is a known assignment in the domain.

Using the defined interpretation, a mapping of terms onto elements of the domain can be defined. For that purpose, the definitions of a variable assignment and a term interpretation are given.

6.3.2 DEFINITION: Let I be an interpretation with domains $\{D_\tau\}$ of an object-level language L. A *variable assignment V (with respect to (wrt) I)* is the assignment to each variable $\mathtt{X}$ of type τ in L of an element $V_\tau(\mathtt{X})$ in D_τ.

6.3.3 DEFINITION: Let I be an interpretation with domains $\{D_\tau\}$ of an object-level language L and let V be a variable assignment. The *term interpretation (wrt I and V)* of a term of type τ in L is inductively defined as follows:

1. Each constant symbol $\mathtt{c}$ of type τ is given its interpretation in accordance with I to $\mathtt{c}_I$.
2. Each variable $\mathtt{X}$ of type τ is given its assignment $V_\tau(\mathtt{X})$.
3. If $V_{\tau_i}(\mathtt{t}_i)$ is the term interpretation of each $\mathtt{t}_i$ of type τ_i and $\mathtt{f}_I$ is the interpretation of the n-ary function symbol $\mathtt{f}$ of type τ, then $\mathtt{f}_I(V_{\tau_1}(\mathtt{t}_1), \ldots, V_{\tau_n}(\mathtt{t}_n))$ is the term interpretation of $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$.

The expression '$\mathtt{T}^I_V\,\phi$' means that the formula ϕ is *true* in an interpretation I using the variable assignment V. By the same token, the expression '$\mathtt{F}^I_V\,\phi$' means that the formula ϕ is *false*, and '$\mathtt{U}^I_V\,\phi$' means that the formula is *unknown*.

6.3.4 DEFINITION: Let I be an interpretation with domains $\{D_\tau\}$ of an object-level language L and let V be a variable assignment. Then a formula in L can be given a *truth value, true, false,* or *unknown, (wrt I and V)* as follows:

1. If $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ is an atom of type $\tau_1 \times \cdots \times \tau_n$ then

   $\mathtt{T}^I_V\,\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ iff $(V_{\tau_1}(\mathtt{t}_1), \ldots, V_{\tau_n}(\mathtt{t}_n)) \in \mathtt{p}_I$

   i.e., the sequence of elements associated with $\mathtt{t}_1 \cdots \mathtt{t}_n$ belongs to $\mathtt{p}_I$.

   $\mathtt{F}^I_V\,\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ iff $(V_{\tau_1}(\mathtt{t}_1), \ldots, V_{\tau_n}(\mathtt{t}_n)) \in \sim\mathtt{p}_I$

   i.e., the sequence of elements associated with $\mathtt{t}_1 \cdots \mathtt{t}_n$ belongs to $\sim\mathtt{p}_I$.

$U^I_V\,p(t_1,\ldots,t_n)$ iff neither $T^I_V\,p(t_1,\ldots,t_n)$ nor $F^I_V\,p(t_1,\ldots,t_n)$ i.e., the sequence of elements associated with $t_1 \cdots t_n$ belongs neither to p_I nor to $\sim p_I$.

2. If ϕ and ψ are typed formulae, then the truth values of the typed formulae $\sim\phi$, $\phi \,\&\, \psi$, and $\phi \mid \psi$, are given in the following table:

ϕ	ψ	$\sim\phi$	$\phi \,\&\, \psi$	$\phi \mid \psi$	$\phi \rightarrow \psi$
true	true	false	true	true	true
true	false	false	false	true	false
true	unknown	false	unknown	true	unknown
false	true	true	false	true	true
false	false	true	false	false	true
false	unknown	true	false	unknown	true
unknown	true	unknown	unknown	true	true
unknown	false	unknown	false	unknown	unknown
unknown	unknown	unknown	unknown	unknown	unknown

The truth values of an antecedent, and a consequent follow directly from the definition of the truth value of typed formulae. Finally, the definition of the truth value of a rule comes into being.

6.3.5 DEFINITION: Let I be an interpretation with domains $\{D_t\}$ of an object-level language L and let V be a variable assignment. Suppose ϕ is an antecedent and ψ is a consequent. Then a rule in L can be given a truth value, *truth, false*, or *unknown, (wrt I and V)* in accordance with the above truth table.

Finally, an interpretation is a model for a scenario and a fact-base if the evaluation of every rule to the fact-base is true in the interpretation. They are consistent when they have a model.

6.3.6 DEFINITION: A formula ϕ is *true in the interpretation* I under a variable assignments V, if $T^I_V\,\phi$; it is written as $I \models_V \phi$. If a formula ϕ holds for all variable assignments (or if ϕ is a ground formula), it is written as $I \models \phi$.

6.3.7 DEFINITION: Let Λ be an object-level scenario with only ground formulae and let Γ be a fact-base. An interpretation I is a *model for* $\Lambda \cup \Gamma$ if $I \models \phi$ for every ground formula $\phi \in \Lambda \cup \Gamma$.

6.3.8 DEFINITION: A ground formula ϕ is a *logical consequence* of $\Lambda \cup \Gamma$, denoted by $\Lambda \cup \Gamma \models \phi$, if ϕ holds in each model of $\Lambda \cup \Gamma$.

6.3.9 DEFINITION: Let Λ be an object-level scenario with only ground formulae and let Γ be a fact-base. A set $\Lambda \cup \Gamma$ is called *consistent* when it has a model.

Let me demonstrate the above definitions with an example. Suppose I examine the following rule:

```
IF p(X) & q(a,f(b)) THEN r(X,a,b)
```

When the rule is interpreted to some domain, the variable `X` is mapped to an element that occurs in the domain. The same happens to the constant symbols `a` and `b`. The function symbol `f` is mapped to an operation. The mapping of the terms is accomplished by the defined term interpretation. If for some variable assignment the antecedent of the rule is true, the rule as a whole is true and the mapping for `X` is `c`, then the literal fact `r(c,a,b)` is also true in the domain.

6.3.2 The object-level derivation relation

Recall DEFINITION 6.2.16 stating that a object-level scenario is a finite set of rules and literal facts. I call the rules a knowledge-base and the set of literal facts a fact-base. The knowledge-base is used to derive conclusions from the literal facts in the fact-base. A similar approach is taken by [Tan and Treur, 1991]. The basic derivation relation used in the object-level language is "from ϕ and $\phi \rightarrow \psi$ conclude ψ" written as[5]:

Modus Ponens: $\phi, \phi \rightarrow \psi \vdash \psi$

The symbol $\vdash$ denotes a derivation relation. The formulae before the relation symbol are the *premises*, and the one after the relation is the *conclusion*. The above derivation *applied* an object-level rule $\phi \rightarrow \psi$. The used inference mechanism (chaining, see §6.4.3) amounts to drawing conclusions from the premises in the fact-base and the knowledge-base. Only literal facts or conjunctions of them are derived as conclusions.

Since the antecedent of a rule consists of both conjunctions and disjunctions of literal formulae, the following derivation relations which *introduce* the connectives `&` and `|` are needed:

And Introduction: $\phi, \psi \vdash \phi \,\&\, \psi$
Or Introduction: $\phi \vdash \phi \mid \psi$
Or Introduction: $\psi \vdash \phi \mid \psi$

The derivation relations that introduce a connective are used to deduce the validity of the antecedent of a rule. The consequent of a rule consists of only conjunctions of literal formulae. Therefore, only the following two derivation relations that *eliminate* the connective `&` are necessary:

And Elimination: $\phi \,\&\, \psi \vdash \phi$
And Elimination: $\phi \,\&\, \psi \vdash \psi$

In logical languages it is extremely important that derived literal facts are actually *valid*, i.e., that the conclusions made by the deduction process are indeed a logical

---

[5] Notice that the implication $\rightarrow$ denotes the same as an `IF-THEN` rule in the object-level language.

consequence of the knowledge-base and the fact-base. A language that has such a property is called *sound*. The definition of *soundness* is that everything which can be derived from a scenario and a fact-base is a logical consequence, in other words:

$$\Lambda \cup \Gamma \vdash \phi \Rightarrow \Lambda \cup \Gamma \models \phi$$

Tan and Treur show ([Tan and Treur, 1991] and [Tan, 1992] pp. 28-29) that any standard derivation relation is sound with respect to the strong Kleene semantics if only literal facts are allowed as final conclusions. If the conclusion of a disjunction, such as $\vdash \sim\phi \mid \phi$, is allowed the inference relation is not sound since $\sim\phi \mid \phi$ is not always true with respect to the strong Kleene semantics while it *is* true in classical logic [van Dalen, 1985]. The counterpart of soundness is *completeness*: everything that is a logical consequence can be derived:

$$\Lambda \cup \Gamma \models \phi \Rightarrow \Lambda \cup \Gamma \vdash \phi$$

Proving the completeness of a derivation relation is much more demanding than proving the soundness. Below, I give an example showing that the object-level derivation relation is incomplete though I also argue that in practice such incompleteness does not really matter.

It is now time to give a definition of the derivation relation. The used chaining is a subrelation of natural deduction. The formulae used in the definition do not have variables or function symbols. The terms are restricted to constant symbols. An atomic formula can therefore be regarded as a propositional constant and the object-level language can be treated as propositional logic rather than predicate logic.

6.3.10 DEFINITION: Let Λ and Λ' be sets of ground formulae and let ϕ and ψ be formulae. The derivation relation $\vdash$ is inductively defined as follows:

1. $\Lambda \vdash \phi$ if $\phi \in \Lambda$. (ϕI)
2. If $\Lambda \vdash \phi$ and $\Lambda' \vdash \psi$, then $\Lambda \cup \Lambda' \vdash \phi \,\&\, \psi$. (&I)
3. If $\Lambda \vdash \phi$ or $\Lambda \vdash \psi$, then $\Lambda \vdash \phi \mid \psi$. (|I)
4. If $\Lambda \vdash \phi \,\&\, \psi$, then $\Lambda \vdash \phi$ and $\Lambda \vdash \psi$. (&E)
5. If $\Lambda \vdash \phi$ and $\Lambda' \vdash \phi \rightarrow \psi$, then $\Lambda \cup \Lambda' \vdash \psi$. ($\rightarrow$E)

Consider an object-level scenario consisting of literal facts and rules. A step taken by the reasoning process consists of the application of a rule from the knowledge-base to the fact-base using DEFINITION 6.3.10. A formula is constructed form the fact-base by using the introduction derivation relations &I and |I, which is matched with the antecedent of the rule. If the match succeeds, literal facts are derived from the consequent using the elimination derivation relation &E. These literal facts are added to the fact-base as *derived literal facts*. In this way, the rule is applied using the Modus Ponens derivation relation $\rightarrow$E.

As said before, the above derivation relation is sound with respect to the strong Kleene semantics. However, this does not guarantee that all derived conclusions are consistent with the fact-base. It might be the case that the knowledge engineer has created conflicting rules in a scenario. Let me demonstrate this with an example of the following fact-base and scenario:

Γ = { p(a), q(a) }
Λ = { p(a) → r(a), q(a) → ~r(a) }

The first rule derives r(a) and the second one derives ~r(a), either of which is a valid conclusion. However, both conclusions together create an undesirable situation, since the conclusions contradict each other when both rules are applied. Such an inconsistency can occur due to the application of incorrect knowledge supplied by the user (i.e., p(a) and q(a) as supplied by the user are not consistent with Λ). When such a situation occurs the reasoning process must halt and the scenario needs to be repaired. It is therefore important that the derived literal facts are consistent with the fact-base, i.e, they do not contradict the current information state. The notion of consistency is further discussed in §6.4.3

The incompleteness of the object-level inference relation is fairly easy to show. Consider the following fact-base and scenario:

Γ = { ~r(a) }
Λ = { p(a) → r(a) }

then the literal ~p(a) is a logical consequence. However, since the derivation relation is based on chaining such a conclusion can not be drawn and thus the derivation relation is not (always) complete. It is the duty of the knowledge engineers to represent the knowledge in the scenarios in such way that everything that is a logical consequence can indeed be derived. In [Langevelde and Treur, 1991], it is shown that this can actually be achieved.

6.4 Procedural aspects of object-level languages

As shown in §6.3.1, the declarative semantics of an object-level language gives a meaning to the symbols and syntactic structure of the language. In §6.3.2, the derivation relation has been discussed. However, it does not reveal anything about how such a language computes, i.e., what are the consequences of the execution of an object-level scenario? The *procedural mechanism* –or *operational* semantics– of an object-level language deals with the methods how an object-level scenario is executed. In other words, a procedural interpretation computes what has been specified by the declarative specification. The declarative specification gives a meaning independent of a computer implementation.

The purpose of an object-level scenario is to perform reasoning about the information state of a (partial) design object description. This reasoning is a deduction process that derives new literal facts from an object information state.

The next section gives an example how this process takes place, and the following sections discuss the methods that compute this process.

6.4.1 An example

I can imagine that after quite a few definitions the reader may need some explanation how things work in practice. The application of an object-level scenario is discussed in order to illustrate the concepts presented in this chapter. Each scenario has a *signature* that describes the domain of an interpretation of the object-level language. It contains the names of the types, the names and types of the constant symbols, the names and types of predicate symbols and the names and types of function symbols. Suppose `solve-limitPositions` is the name of an object-level scenario whose goal is to determine the limit positions of a linear motion mechanism. The scenario is a part of the example design system implemented in ADDL discussed in Chapter 9. Its signature and rules are denoted by Σ(`solve-limitPositions`) as follows:

Σ(`solve-limitPositions`)

Type	**Notation**
composite	C
face	FA
guide	GU
number	NU
objectInMotion	OM
point	PT
Function	**Type**
angle	FA→NU
Predicate	**Type**
contact	FA×FA×PT
equal	NU×NU
hasPart	C×C
isFace, isPin, isSlot	C
limitArrangement	OM×GU×PT
linearMotion	OM×PT×PT
startPosition, endPosition	PT
limitPositions	nil

Rules

```
1   IF limitArrangement(O,G,PT) & linearMotion(O,PT,ω)
       & isFace(F1) & hasPart(G,F1) & equal(angle(F1),270)
       & isFace(F2) & hasPart(O,F2)
    THEN startPosition(PT) & contact(F2,F1,PT)
2   IF limitArrangement(O,G,PT) & linearMotion(O,ω,PT)
       & isFace(F1) & hasPart(G,F1) & equal(angle(F1),90)
       & isFace(F2) & hasPart(O,F2)
    THEN endPosition(PT) & contact(F2,F1,PT)
3   IF startPosition(ω) & endPosition(ω)
    THEN limitPositions
```

The scenario consists of a knowledge-base with three rules. In the next chapter –on the meta-level language– the world-mechanism is discussed. Here it suffices to mention that when the scenario is activated, it receives a *world* that it is viewing. A world is a sub-set of the fact-base. Suppose `limitPositions` is activated viewing the following world (note that attribute values, which appear in the object-base, are kept out of consideration; thus information about the angles of faces is not shown):

```
isFace(face1)      hasPart(pin1,face5)
isFace(face2)      hasPart(slot1,face1)
isFace(face3)      hasPart(slot1,face2)
isFace(face4)      hasPart(slot1,face3)
isFace(face5)      hasPart(slot1,face4)
isPin(pin1)        limitArrangement(pin1,slot1,point1)
isPoint(point1)    limitArrangement(pin1,slot1,point2)
isPoint(point2)    linearMotion(pin1,point1,point2)
isSlot(slot1)
```

Viewed globally, the inference mechanism deduces the following conclusions from the first rule and the world. These conclusions are registered in a *set of hypotheses*.

```
startPosition(point1)
contact(face5,face4,point1)
```

The second rule produces:

```
endPosition(point2)
contact(face5,face2,point2)
```

The third rule confirms that the required goal has been satisfied:

```
limitPositions
```

Now, I examine the application of the first rule in more detail. The computation of

the truth value of the antecedent is illustrated in Fig. 6.1. I adopted the notation used in [Clocksin and Mellish, 1981]. The derivation procedure searches the world in a top-down manner. It tries to unify a literal formula with the first matching literal fact. Therefore, the third literal formula in the antecedent is unified with `isFace(face1)`, which is correct with respect to the fourth literal formula; `face1` is indeed a part of `slot1`. However, according to the fifth literal formula, which is a built-in predicate, the value of the attribute angle of the face must be `270` degrees, which is not the case for `face1`. The truth value of the fifth atom is thus false and the derivation procedure will backtrack to the previously visited atoms.

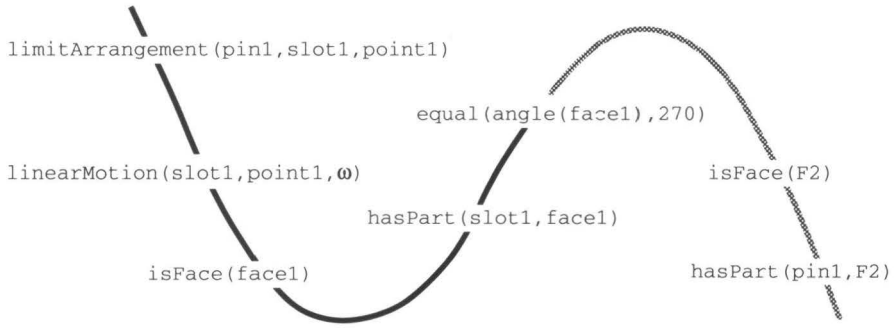


Fig. 6.1 Application of the derivation procedure up to the unification of the literal formula `equal(angle(F1),270)`, which fails.

ADDL will try to find an alternative fact by backtracking over the previous literal formulae. Backtracking is unsuccessful over the fourth literal formula, but succeeds with the third literal formula. The derivation procedure will be retried with `F1` bound to `face2`. This process is continued until finally the fifth literal formula holds for `face4`, since the value of its angle is equal to `270` degrees. This information is stored in an attribute of `face4` and can be obtained by the evaluable term `angle(F1)`. The derivation procedure binds `F2` to `face5` after four times backtracking over the sixth literal formula. In Fig. 6.2 the result is depicted.

Obviously, the efficiency of the derivation procedure relies heavily on the way the rules are implemented. Suppose the first rule was written down as follows:

```
1   IF isFace(F1) & isFace(F2)
       & limitArrangement(O,G,PT) & linearMotion(O,PT,ω)
       & hasPart(O,F2) & hasPart(G,F1) & equal(angle(F1),270)
    THEN startPosition(PT) & contact(F2,F1,PT)
```

Only at the last literal formula the derivation procedure can notice that the binding of `F1` is wrong. The backtracking mechanism will unnecessarily try to resatisfy the

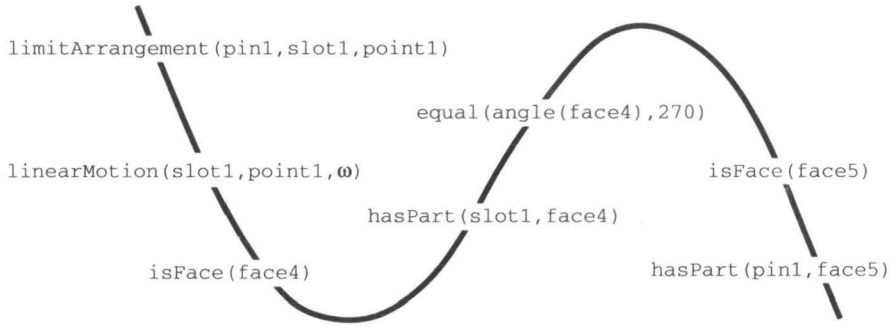


Fig. 6.2 Successful resolution after backtracking.

second till the sixth literal formula. Though the ultimate result of the procedure is the same for both versions of the first rule, the amount of effort to satisfy the antecedent is much greater in the second case.

The literal facts derived from the three rules of `solve-limitPositions` are consistent with the world. The state transition after application of the three rules thus contains the conclusions derived from the rules and the world. Therefore, the contents of the new fact-base is (supposing that the original fact-base had the same contents as the world):

```
isFace(face1)          hasPart(slot1,face2)
isFace(face2)          hasPart(slot1,face3)
isFace(face3)          hasPart(slot1,face4)
isFace(face4)          startPosition(point1)
isFace(face5)          contact(face5,face4,point1)
isPoint(point1)        endPosition(point2)
isPoint(point2)        contact(face5,face2,point2)
isPin(pin1)            limitArrangement(pin1,slot1,point1)
isSlot(slot1)          limitArrangement(pin1,slot1,point2)
hasPart(pin1,face5)    linearMotion(pin1,point1,point2)
hasPart(slot1,face1)   limitPositions
```

The conclusion `limitPositions` is not literally included in the fact-base. It is information used by the meta-level interpreter stating that the goal `limitPositions` has been satisfied. The next chapter discusses the meta-level language. The switching between the object-level interpreter and the meta-level interpreter will there be presented.

6.4.2 Term evaluation and unification

The purpose of a unification algorithm is to compute bindings. When a literal formula is matched against the fact-base, the terms appearing in the literal formula are substituted by constant symbols. Unification is an important mechanism to generate ground formulae. Recall that the derivation relation presented in §6.3.2 required formulae to be ground. This section presents the unification algorithm employed to ensure the groundness of formulae. This algorithm differs quite a lot from the unification algorithms described in literature [Lloyd, 1987; Martelli and Montanari, 1982], because the terms in the object-level language are *evaluable*. An *evaluable term* is a non-simple single-level term as defined by the second entry of the definition of a single-level term. It has a certain procedure attached to it that returns a value upon evaluation. An example of such a procedure is an operation as defined in §5.2. An evaluable term can be evaluated in accordance with the following definition.

6.4.1 DEFINITION: An *evaluation mapping* is a function `Eval : evaluable term → constant symbol` that maps an evaluable term `t` of type τ to a constant `c` of type τ as follows: `Eval(t) = c`.

The `Eval` function can be compared with a set of *rewrite rules*. The evaluable term is rewritten as a constant symbol. With respect to the definition of the evaluation of an evaluable term, the following claim to an interpretation holds:

STIPULATION: Let `Eval` be an evaluation mapping let `t` be an evaluable term of type τ. Then for any interpretation I with variable assignment V: V_τ`(Eval(t))` = V_τ`(t)`.

The above stipulation of interpretations guarantees that each evaluable term is mapped onto an operation in the domain.

Another reason for using a different algorithm stems from the nature of the fact-base. Since the literal facts appearing in the fact-base only have constant symbols as their terms, the terms of the literal formula being unified can only be unified with constant symbols. This simplifies the unification algorithm drastically. Another advantage is the ability to treat the object-level language as propositional logic since every term is replaced by a constant. The algorithm makes a distinction between two kinds of variables.

6.4.2 DEFINITION: Let ϕ be an m-ary literal formula with the terms $t_1 \cdots t_m$ and let $X_1 \cdots X_n$ be the variables appearing in the literal formula. Then the variables X_j for which $t_i \equiv X_j$ for some i are called *first-order variables*. All other variables are called *second-order* variables.

From DEFINITION 6.4.2, it follows unequivocally that second-order variables are those variables being part of an evaluable term that are *not* first-order variables. Now the definition of variable substitutions can be given:

6.4.3 DEFINITION: A *substitution* θ is a finite set of the form $\{X_1 / c_1, \ldots, X_n / c_n\}$ where each X_i is a distinct variable of type τ_i and each c_i is a constant symbol of type τ_i. Each element X_i / c_i is called a *binding* for X_i.

6.4.4 DEFINITION: Let θ be a substitution and t be a *simple* term, then $t\theta$ stands for the result of applying θ to t. If t is a variable, then it is replaced by its corresponding constant symbol. If t is a constant symbol, then nothing is done.

6.4.5 DEFINITION: Let θ be a substitution and t be a *evaluable* term, then $t\theta$ stands for the result of applying θ to t. This is obtained by replacing each occurrence of a variable of t by its corresponding constant symbol and further by rewriting the term as a constant symbol by `Eval(t)`.

6.4.6 DEFINITION: Let ϕ be a literal formula of type $\tau_1 \times \cdots \times \tau_n$ and p be a literal fact of type $\tau_1 \times \cdots \times \tau_n$. A substitution θ such that for each term t_i of ϕ and each constant c_i of p, $t_1\theta \equiv c_1, \ldots, t_n\theta \equiv c_n$ is called a *unifier*.

Thus for obtaining a unifier of a literal formula and a literal fact it suffices to find bindings for all the variables of the literal formula and to evaluate all evaluable terms. Each resulting constant symbol at position i of the literal formula must then be equal to the constant symbol at position i of the literal fact.

The unification algorithm employs a strategy called *immediate evaluation*, i.e., first the first-order variables are bound, and then the evaluable terms are processed. If an evaluable term contains second-order variables, it evaluates to the constant symbol `nil`.

The following algorithm finds a unifier if possible:

UNIFICATION ALGORITHM: Let ϕ be an n-ary literal formula of type $\tau_1 \times \cdots \times \tau_n$ and let p be an n-ary literal fact of type $\tau_1 \times \cdots \times \tau_n$. The unification algorithm consists of three steps:

1.i. Choose from the set of first-order variables a variable X of ϕ on position i. Bind X to the constant symbol c on position i of p and replace all occurrences of X of ϕ by c.

ii. Repeat step i. until all first-order variables are bound.

2.i. Choose from the set of evaluable terms a term t on position i. The term is replaced by `Eval(t)`.

ii. Repeat step i. until all terms are evaluated.

3. Compare ϕ and p. If they are equal, then θ, the set of bindings for the first-order variables, is a unifier of ϕ and p. Otherwise, ϕ and p are not unifyable and θ becomes $\varnothing$.

Some examples may clarify the above unification algorithm. Suppose there is a successor function symbol `succ` of type `number` $\rightarrow$ `number`. Its attached procedure returns the increment by one of the argument when it is evaluated.

Now suppose I want to unify the following literal formula (i) of type `symbol × number × number`

```
(i)     p(X,Y,succ(1))
```

with a fact-base that contains the following literal facts (ii) and (iii) of type `symbol × number × number`:

```
(ii)    p(a,2,2)
(iii)   p(a,2,3)
```

The literal fact (i) contains the first-order variables `X` and `Y`. The unification algorithm is used to find a unifier for (i) and (ii). At first, the bindings `X/a` and `Y/2` are found by application of step 1. The evaluable term `succ(1)` is replaced by the constant symbol `2` by means of step 2. The algorithm succeeds with the unifier `{X/a,Y/2}`. However, the application of the algorithm to (i) and (iii) fails because of `succ(1)` $\neq$ `3`.

Now suppose that the literal formula (iv) is to be applied with the same fact-base.

```
(iv)    p(X,Y,succ(Y))
```

A unifier for (iv) and (ii) cannot be found. After application of step (1) the bindings `X/a` and `Y/2` are obtained like in the previous example. But now the occurrence of `Y` in the evaluable term is also replaced. Its evaluation returns the number `3` which is obviously not equal to the third term of (ii). A unifier *can* be found for (iv) and (iii), viz. `{X/a,Y/2}`.

6.4.1 UNIFICATION THEOREM: Let ϕ be an n-ary literal formula of type $\tau_1 \times \cdots \times \tau_n$ and `p` be an n-ary literal fact of type $\tau_1 \times \cdots \times \tau_n$. If ϕ and `p` are unifiable, then the unification algorithm terminates and returns a unifier of ϕ and `p`. If ϕ and `p` are not unifiable, then the unification algorithm terminates and returns $\varnothing$.

PROOF: The unification algorithm terminates because ϕ has only a finite number of first-order variables and evaluable terms. Each application of step 1 replaces a first-order variable by a constant symbol and each application of step 2 replaces an evaluable term by a constant symbol.

Application of step 1 of the algorithm binds a first-order variable of ϕ to a constant symbol of `p` occurring on the same position. Multiple occurrences of the same variable are replaced by the same constant symbol. Step 1 is repeated until all first-order variables are bound. It is evident that after step 1, ϕ only consists of constant symbols, second-order variables, and evaluable terms.

Obviously, repeated application of step 2 replaces the evaluable terms of ϕ by constant symbols. Now, ϕ consists of only constant symbols since second-

order variables only occur in evaluable terms. The literal fact p also consists of only constant symbols. Let $b_1 \cdots b_n$ be the constant symbols of ϕ and let $c_1 \cdots c_n$ be the constant symbols of p. If for each $\{i \mid 1 \leq i \leq n\}$ holds that $b_i \equiv c_i$, then the set of bindings for the first-order variables of ϕ is indeed a unifier of ϕ and p. □

The next two sections discuss the computational mechanism used for the evaluation of rules. It shows how the unification algorithm is employed in order to evaluate a rule. Such an evaluation is called a *derivation procedure*. The applicability of a rule is checked with the fact-base. The derivation procedure computes the truth of the antecedent and it finds bindings for the variables occurring in the antecedent. This procedure actually uses the derivation relations &I and |I to compute the truth value of the antecedent of a rule. These bindings replace the variables occurring in the consequent. Recall that since the consequent is *restricted to* the antecedent, each variable occurring in the consequent can be replaced this way.

6.4.3 Derivation procedures for the antecedent

The derivation procedure for the antecedent computes the derivation relation presented in §6.3.2 and it uses the derivation rules that introduce connectives. Thus conceptually, this section does not provide any new information. Giving insight in the procedural methods used to implement the object-level interpreter is its purpose. In the Chapter 8 about the implementation of ADDL, these methods are further worked out at the implementational level. A reader who is not interested in these rather technical issues may want to skip §6.4.3.

Using the unification algorithm, a procedure to compute the truth value of the antecedent of a rule can be defined. Some supporting definitions are given first.

6.4.7 DEFINITION: A *node* is a tuple <field, left, right>, where field is a string and left and right are pointers to other nodes.

6.4.8 DEFINITION: Let n_0, n_1, and n_r be nodes. If n_0 has pointers to both n_1 and n_r, then n_0 is called a *predecessor* of n_1 and n_r. The nodes n_1 and n_r are called *successors* of n_0.

6.4.9 DEFINITION: A *leaf* is a node with no successors, i.e., a tuple <field, nil, nil>.

6.4.10 DEFINITION: A *tree* is a finite set of nodes which has the following properties:

1. There is one node, called the *root*, that has no predecessors.
2. Each node other than the root has exactly one predecessor.

For each formula of the object-level language a tree representing the formula can be constructed. Such a tree is called a parse tree. I define tokens and parse trees as follows:

6.4.11 DEFINITION: A *token* is either a literal formula, the connective '&' or '|', or a punctuation symbol '(' or ')'.

6.4.12 DEFINITION: A *parse tree* is a tree which has a binary connective stored in the field of each node except for leaves whose leaf contains a literal formula.

Nodes have exactly two successors that represent the arguments of its binary connective. Some procedures to construct parts of the trees are now defined. These procedures are described in a pseudo-language which has some resemblance to the programming language C [Kernighan and Ritchie, 1978]. Before the procedures to construct a parse tree are given, I give an example. Suppose there is the following antecedent:

```
a & ~b | (c & (d | e | f)) & g
```

The corresponding parse tree is shown in Fig. 6.3. A formula between parentheses is a *nested* formula. The above formula contains two nested formulae, and its parse tree has therefore two sub-trees. The reader can easily grasp the construction of the tree by parsing the antecedent from left to right making use of the construction procedures presented below.

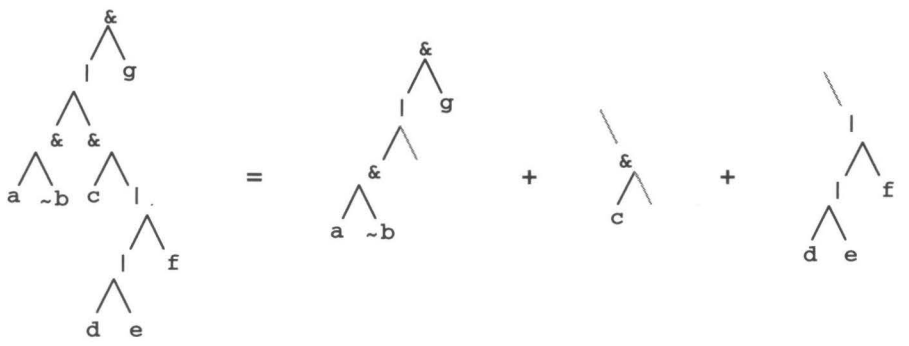


Fig. 6.3 Parse tree and sub-trees of the formula `a & ~b | (c & (d | e | f)) & g`.

The following procedure creates a node and adds it as a leaf to a tree. The token being a literal formula is stored in the field. If the tree is empty it returns the root node of a new tree, else the right pointer of the current node will be directed towards the new node:

PROCEDURE: Let `t` be a literal formula and let `n` be the current node. Then a leaf `nn` with field `t` is added to the tree of nodes `n` as follows:

```
addLeaf (n,t) {
   nn = newNode ();
   nn.field = t;
   if (n != nil) n.right = nn;
   return nn; }
```

The following procedure adds a node to the tree, which stands for a (binary) connective. The previous procedure dealt with the situation that a successor was added to the tree, the following procedure creates a node and adds it as a predecessor of the current node. The left pointer of the new node will point to the current node:

PROCEDURE: Let `t` be a token and let `n` be a node. Then a node `nn` with field `t` is added to the tree of nodes `n` as follows:

```
addNode (n,t) {
   nn = newNode ();
   nn.field = t;
   nn.left = n;
   return nn; }
```

DEFINITION 6.2.9 of a typed formula consisted of three induction rules. The first two induction rules have been handled by the last two procedures. The last rule defined a formula as being a formula between parentheses. Such a formula is implemented as a sub-tree inside a parse tree. The following procedure adds a sub-tree to a tree. Note that such a sub-tree might be a single node (i.e., a leaf).

PROCEDURE: Let `n` and `s` be nodes. Then the sub-tree of nodes `s` is added to the tree of nodes `n` as follows:

```
addTreeLeaf (n,s) {
   if (n != nil) n.right = s;
   return s; }
```

Now using the above definitions the main procedure for constructing a parse tree can finally be given. As mentioned before the procedure stems from the definition of a typed formula. It distinguishes three cases which have a direct correspondence with the three induction rules of that definition. The definition uses some procedures that have not explicitly been defined. The procedure `nextToken` returns the next token of a sequence of tokens and the procedure `root` returns the root of a tree.

PROCEDURE: Let `s` be an antecedent represented by a sequence of tokens, let `t` be a token, and let `n` and `st` be nodes. Then a *parse tree* being the tree of nodes `n` is built up as follows:

```
makeTree (s) {
   n = nil;
   t = nextToken (s);
   do {
      switch (t) {
      case 'literal formula':
         n = addLeaf (n,t); break;
      case '&':
      case '|':
         n = root (n);
         n = addNode (n,t); break;
      case '(':
         st = makeTree (s);
         n = addTreeLeaf (n,st); break; }
      t = nextToken (s); }
   while (t != nil && t != ')');
   return root (n); }
```

Now a strategy for traversing a parse tree is discussed. Through traversal of the tree variables are bound and a truth value for each node is determined. The truth value of a node can be computed if the truth value of both its successors is known. The truth value of the root of the tree is equivalent to the truth value of the entire antecedent. For traversing a parse tree a left-first depth-first strategy is adopted, which technically amounts to a left to right evaluation of formulae. During the search of the tree, a path is created which consists of a sequence of 'true' nodes. Whenever a node becomes 'false', the search will *backtrack* to previous nodes along the path trying to establish new variable bindings. This process is continued as long as new variable bindings can be found.

During traversal of a parse tree, the variable bindings need to be registered. The registration is done by means of an instantiation set:

6.4.13 DEFINITION: Let ϕ be an antecedent and let $X_1, \ldots, X_n$ be the variables occurring in ϕ. The substitution $\{X_1 / c_1, \ldots, X_n / c_n\}$ is called the *instantiation set* of ϕ.

It is evident that if the antecedent ϕ is a literal formula, the instantiation set of ϕ is equivalent to the *unifier* of ϕ. Upon constructing a parse tree for an antecedent ϕ an instantiation set of ϕ can easily be initialized. Each time a literal formula is encountered its variables which are not present in the instantiation set are added to the set. The bindings are set to *nil*.

For convenience the following terminology is adopted. An *and-node* is a node whose token is the connective `&`. The same applies for an *or-node* and the connective `|`. A *leaf* is a node with a literal formula. The direction of search is *forward* when the tree is traversed in a left-first depth-first order. In case of backtracking, the direction of search is *backward* following the path which has previously been chosen in opposite direction. The truth value of a node `n` with

respect to an instantiation set I is called V_n^I. The truth value of a leaf can then be defined as follows (see Fig. 6.4):

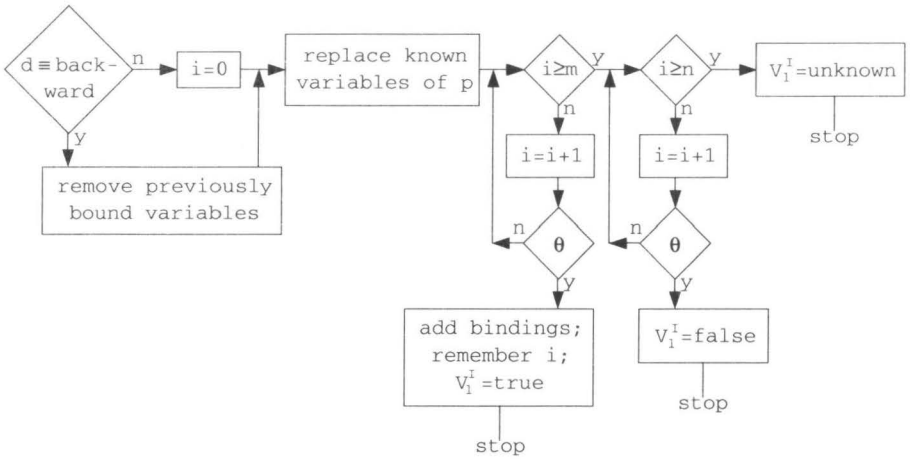


Fig. 6.4 The algorithm to find the truth value of a leaf. The symbol θ stands for the application of the unification algorithm.

6.4.14 DEFINITION: Let l be a *leaf* with the literal formula ϕ, let I be an instantiation set and let d be the direction of search. An array a of size n contains literal facts with the same predicate symbol, arity and type as ϕ. If ϕ is an atom, then the first m elements of a are *positive* facts, the remaining m–n elements are *negative* facts. If ϕ is a negation of an atom, then the negative facts come first and the positive facts last. Let i be an index of a. The truth value V_l^I of l with respect to I is obtained by performing the following steps:

i. If d ≡ backward then remove those variables bindings from I which have been bound during the last visit of l.
   Otherwise put i = 0.
ii. Replace those variables of ϕ, which have a binding in I, by their corresponding constant symbols.
iii. If i ≥ m then go to v. Otherwise increment i.
iv. If θ is a unifier of ϕ and a[i], then add the bindings of θ to I, remember i, put d = forward, put V_l^I = true and stop.
   Otherwise go to iii.
v. If i ≥ n then put V_l^I = unknown and stop. Otherwise increment i.
vi. If θ is a unifier of ϕ and a[i], then put V_l^I = false and stop.
   Otherwise goto v.

Fig. 6.4 shows the six steps in a block diagram. As stated before, a parse tree can be traversed in two directions. Hence, a leaf can be encountered either during forward search or during backward search. Let me discuss both situations separately:

Forward: If the search is forwardly directed, a leaf is encountered in a so-called empty state. The leaf has not yet been reached with the current state of the instantiation set. For example, if taking a look at Fig. 6.5, I notice a leaf `l` with the atom ϕ. The array `a` contains four elements: three positive facts and a single negative fact. Hence, $n \equiv 4$ and $m \equiv 3$. I now follow the algorithm during its first visit. Since the direction is forward, the index `i` will be set to zero. The variable `X` in ϕ will be bound to `a` and `i` will be incremented. The unification of ϕ and `a[1]` does not succeed and `i` is incremented again. The unifier `{Y/a}` of ϕ and `a[2]` is found and added to the instantiation set. The truth value of the node is true.

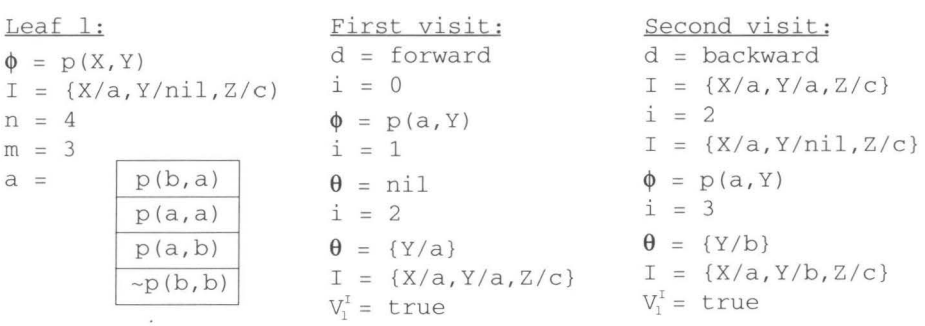


Fig. 6.5 Example of the application of the leaf algorithm to the atom `p(X,Y)`.

Backward: Now the search continues along the remainder of the parse tree. When a next node becomes false or unknown the search will backtrack over `l`. The index having the value two is remembered from the previous visit. The binding `{Y/a}` is removed from the instantiation set. The variable `X` of ϕ is bound to `a` and the index is incremented. It now points to the fact `p(a,b)` whose unification with ϕ results in the unifier `{Y/b}`. The unifier is added to the instantiation set and the leaf succeeds (see Fig. 6.5). When the search will try to backtrack over `l` once again the algorithm will return `unknown` since `~p(b,b)` and ϕ are not unifiable[6] and no other facts remain in `a`.

[6] If they were unifiable, the leaf would have become `false`.

The above definitions are concerned with the truth values of leaves. I will now focus on the truth value of and- and or-nodes. The truth value of an and-node is defined as follows (see Fig. 6.6):

6.4.15 DEFINITION: Let n be an and-node and let n_l and n_r be its left successor and its right successor, respectively, let I be an instantiation set and let d be the direction of search. The truth value of n with respect to I is obtained by performing the following steps:

i. If d ≡ backward, then go to step iv.
ii. Compute V_l^I, the truth value of the left successor of n.
iii. If $V_l^I \equiv$ false or $V_l^I \equiv$ unknown then put $V_n^I = V_l^I \,\&\, V_r^I$ and stop.
iv. Compute V_r^I the truth value of the right successor of n.
v. If $V_r^I \equiv$ false or $V_r^I \equiv$ unknown, then put d = backward and go to step ii. Otherwise put V_n^I = true and stop.

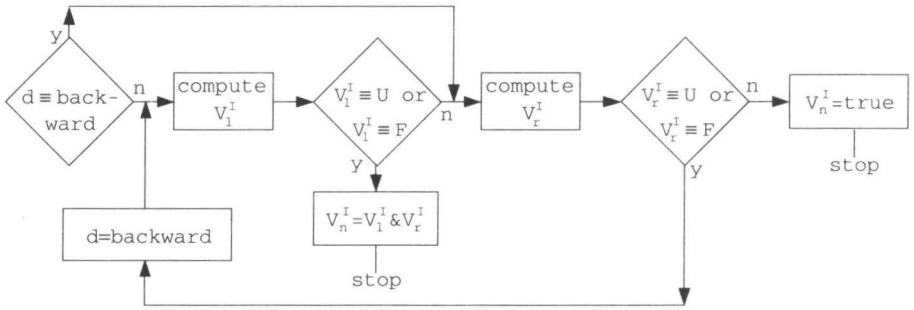


Fig. 6.6 The algorithm to find the truth value of an and-node. Here, F stands for false and U stands for unknown.

Note that the expression "put $V_n^I = V_l^I \,\&\, V_r^I$" means: "assign to V_n^I the truth value of the expression $V_l^I \,\&\, V_r^I$ using the truth table of & defined in §6.4." The presented algorithms are biased towards finding the truth of a node rather than unknown or falsity. Thus, it might occur that for a formula $\phi \,\&\, \psi$ the formula ψ will not be evaluated because ϕ evaluated to unknown or false. In such a case I assume for ψ the value unknown. As a result, it may happen that a formula $\phi \,\&\, \psi$ evaluates to unknown while ϕ is unknown and ψ is false. In this case I consider performance more important than an incorrect evaluation to unknown instead of false.

The truth value of an or-node is defined as follows (see Fig. 6.7):

6.4.16 DEFINITION: Let n be an or-node and let n_l and n_r be its left successor and its right successor, respectively, and let I be an instantiation set. The state of n is called σ. The truth value of n with respect to I is obtained by performing

the following steps:

i. If $d \equiv$ `backward` and $\sigma \equiv$ `right`, then go to step iv.
ii. Compute V_l^I, the truth value of the left successor of `n`.
iii. If $V_l^I \equiv$ `true`, then put σ = `left`, put V_n^I = `true`, and stop.
iv. Compute V_r^I the truth value of the right successor of `n`.
v. If $V_r^I \equiv$ `true`, then put σ = `right`, put V_n^I = `true`, and stop. Otherwise put $V_n^I = V_l^I \mid V_r^I$ and stop.

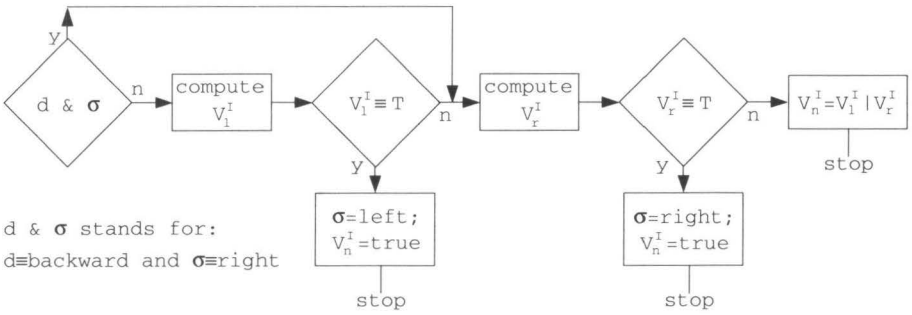


Fig. 6.7 The algorithm to find the truth value of an or-node. Here, `T` stands for `true`.

The expression $V_l^I \mid V_r^I$ is evaluated in accordance with the truth table for $\mid$ given in §6.4. The algorithm for the or-node is sound with respect to the falsity and unknownness of the node. It always finds the proper truth value.

Using the above definitions, the truth value of a parse tree can easily be defined:

6.4.17 DEFINITION: Let `T` be a parse tree and let `n` be the root of the tree and let `d` be the direction of search. The truth value of `T` is obtained by computing the truth value of `n` with `d = forward`.

6.4.4 Derivation procedures for the consequent

With the methods presented above the truth value of the antecedent of a rule is derived from a fact-base. If the consequent of the rule is consistent with the fact-base, then it is assumed to be a valid conclusion. The inference rule `E&` is used to derive a set of literal facts from the consequent. The methods which are used to derive these literal facts and to check their consistency are given in the remainder of this section.

A set of literal facts can be derived from the consequent. The set constructor is called Δ and the expression $\Delta\,\phi$ denotes the set of literal facts derived from a formula ϕ. The definition of the derivation of a set of literal facts is as follows:

6.4.18 DEFINITION: Let ϕ be a consequent and let `I` be an instantiation set. A *ground consequent* χ is obtained by replacing each variable in ϕ by its binding in `I` and by evaluating each evaluable term.

6.4.19 DEFINITION: Let χ be a ground consequent. The set of derived literal facts $\Delta\chi$ is inductively constructed as follows:

1. If χ is an literal formula, then return $\{\chi\}$.
2. If χ is of the form ϕ & ψ, then return $\Delta\phi \cup \Delta\psi$.
3. If χ is of the form (χ), then return $\Delta\chi$.

Note that a proposition symbol is not being treated by the above definition. It is done in the following section on built-in predicates. Due to an inconsistency between a fact-base and a scenario the set of derived literal facts may contain literal facts whose negation occurs in the fact-base. Such a situation is undesirable. The set of derived literal facts may also contain literal facts that are already present in the fact-base. The state transition avoids the generation of duplicates by adding time stamps to duplicates. It is constructed as follows:

6.4.20 DEFINITION: Let Φ be a set of derived literal facts and let Γ be a fact-base. Then a *state transition* $\Gamma \Rightarrow \Gamma'$ where $\Gamma' = \Gamma \cup \Phi$ is obtained by the following sequence of steps:

i. Put $\Gamma' = \Gamma$.
ii. Take a literal fact ϕ from Φ.
iii. If ϕ is a positive fact and $\sim\phi \in \Gamma'$, then `user error`.
iv. If ϕ is a negative fact of the form $\sim\psi$ and $\psi \in \Gamma'$, then `user error`.
v. If there is a $\psi \in \Gamma'$ such that $\psi \equiv \phi$, then add a time-stamp to ψ. Otherwise add ϕ to Γ'.
vi. If $\Phi \equiv \varnothing$, then stop. Otherwise, go to step ii.

If during a state transition a `user error` is raised, it implies that an inconsistency is detected. The following theorem proves that such a situation cannot occur if a scenario and a fact-base are consistent before the state transition and if the derivation does *not* use user knowledge. When user knowledge *is* being used a state transition maintains only consistency if the information provide by the user is consistent with the model. Thus in case of user supplied information the system cannot guarantee consistency. Therefore, the extra check in term iii) and iv) is necessary when user knowledge is being applied.

6.4.2 CONSISTENCY THEOREM: Let Φ be a set of derived literal facts, let Γ be a fact-base and let Λ be a scenario. Suppose $\Lambda \cup \Gamma$ is consistent, $\Lambda, \Gamma \vdash \Phi$ and Γ' is the fact-base obtained by a state transition of Γ and Φ, then $\Lambda \cup \Gamma'$ is consistent.

PROOF: Because of the soundness of the object-level derivation relation, only *valid* conclusions are derived. Since $\Lambda \cup \Gamma$ is consistent and the set of derived literals Φ contains only valid conclusions, the set $\Lambda \cup \Gamma'$ is also consistent. □

If an error of the user is encountered, the reasoning will halt and the cause of the inconsistency is removed in dialogue with the user. This mechanism is discussed in Chapter 8 (see §8.3.4).

Recall that §6.2 defined an object-level scenario as a finite set of rules and literal facts. Execution of a scenario amounts to evaluating the rules against the literal facts one by one till the goal of the scenario has been reached. Each time a rule has been applied, the fact-base of the scenario is subject to a state transition. This issue has informally been discussed by the example in §6.4.1.

6.4.5 Built-in predicates

So far, the truth value of literal formulae has been computed by matching its predicate symbol with that of literal facts appearing in the fact-base. Some predicate symbols however have a predefined meaning. Such a predicate symbol is called a *built-in* predicate. A certain procedure is attached to these symbols. When a built-in predicate is encountered by the inference mechanism, the procedure is executed. Negations of built-in predicates are not allowed. The procedure of built-in predicates occurring in the antecedent, computes its truth value rather than that it attempts unification. The procedure of a built-in predicate that occurs in the consequent, computes additional information about the asserted literal fact. This section presents the built-in predicates of the object-level language and it defines their meanings.

In the object-level language I distinguish two sorts of built-in predicates: those appearing in the antecedent and those appearing in the consequent of a rule. The former are the predicate symbols `equal`, `notNil`, `isNil`, and `typeFor` predicate symbols which are object definitions, predicate symbols which stand for relational symbols, and predicate symbols which address the user-interface. The latter are `value`, `hasPart`, predicate symbols which look like `isType` and predicate symbols which solve a goal. They are presented in that order.

Built-in predicates occurring in an antecedent. The procedures of built-in predicates used in the antecedent compute a truth value.

equal: For the computation of the truth value of the binary built-in predicate `equal` its two terms are compared. The value will be true if they are equal. It will be false if they are not. If one of the two terms is an unbound variable, it will receive the value of the other term and the value will be true. The truth value will be `unknown` if more than one unbound variable is present. The following procedure computes the truth value of `equal`:

PROCEDURE: Let `p` be an atom of type `object` × `object` of the form `equal(`t_1`,` t_2`)` and let `I` be an instantiation set. The truth value V_p^I of `p` with respect to `I` is computed by the following sequence of steps:

i. Replace all variables in p which have a binding in I.
ii. Evaluate the evaluable terms in p.
iii. If t_1 and t_2 are variables, then put V_p^I = unknown and stop.
iv If one of the terms t_i is a variable, e.g. X, and the other term is a constant symbol, e.g. c, then add the binding {X/c} to I, and put V_p^I = true. Otherwise put $V_p^I = t_1 \equiv t_2$.

notNil and **isNil**: The unary predicate symbols notNil and isNil are also used to test the values of their terms. The truth value of the built-in predicate notNil is true if its term is *not* equal to nil. It is false otherwise. The built-in predicate isNil has an opposite meaning. Its truth value is false if its term is not equal to nil. It is true otherwise. The following procedures perform the computation:

PROCEDURE: Let p be an atom of type symbol of the form notNil(t) and let I be an instantiation set. The truth value V_p^I of p with respect to I is computed by the following sequence of steps:
i. Replace all variables in p which have a binding in I.
ii. If t is an evaluable term, then evaluate it.
iii. If t is a variable or $t \equiv$ nil, then put V_p^I = false.
Otherwise put V_p^I = true.

PROCEDURE: Let p be an atom of type symbol of the form isNil(t) and let I be an instantiation set. The truth value V_p^I of p with respect to I is computed by the following sequence of steps:
i. Replace all variables in p which have a binding in I.
ii. If t is an evaluable term, then evaluate it.
iii. If t is a variable or $t \equiv$ nil, then put V_p^I = true.
Otherwise put V_p^I = false.

typeFor: The binary predicate symbols typeFor reserves a name for an object of the type given by the second argument. Its truth value is true if the first argument is an unbound variable which will be bound to the new name. Otherwise the truth value is false. The following procedure computes the new name:

PROCEDURE: Let p be an atom of type symbol×symbol of the form typeFor(t_1, t_2) and let I be an instantiation set. The truth value V_p^I of p with respect to I is computed by the following sequence of steps:
i. Replace all variables in p which have a binding in I.
ii. Evaluate the evaluable terms in p.
iii. If t_1 is not an unbound variable or t_2 does not represent a type then put V_p^I = false. Otherwise, create a new name c bind it to the variable t_1 and add the binding to I.

object definitions: A unary built-in predicate of the form `isFoo` defines an object of type `foo`. This category of built-in predicates has two procedures attached to it. One procedure which is executed when the built-in predicate occurs in the antecedent, and another which is executed when it occurs in the consequent of a rule. The second case is presented in the section about built-in predicates occurring in the consequent. Recall that the types employed by ADDL are organized in a type hierarchy. This type hierarchy has its impact on the computation of the truth value of object definition literal formulae occurring in the antecedent of a rule.

The truth value of an object definition literal formula `p` is computed using the definition of the truth value of a *leaf*. Note that, I *do* allow negations of object definition predicates as opposed to other built-in predicates. If that value is `unknown`, the predicate symbol of `p` is replaced by a sub-type of `p` and the truth value is computed again. This process is continued till the number of sub-types is exhausted:

PROCEDURE: Let `p` be an object definition literal formula of type τ of the form `isT`, let `I` be an instantiation set and let `U` be a collection of sub-types of τ. Then the truth value V_p^I of `p` with respect to `I` is computed by the following sequence of steps:

i. Compute V_p^I the truth value of a *leaf*.
ii. If either $V_p^I \equiv$ `true` or $V_p^I \equiv$ `false`, then stop.
iii. If `U` is exhausted, then stop ($V_p^I \equiv$ `unknown`).
Otherwise, take the next sub-type υ from `U` and replace the predicate symbol of `p` by `isY`. Go to step i.

relational symbols: The relational symbols are the following binary built-in predicates: `greater`, `greaterEqual`, `smaller`, `smallerEqual` and `notEqual`. Their truth value is obtained by comparing the two terms in accordance with the relation symbol in question. They are defined as follows:

PROCEDURE: Let `p` be an atom type `number` × `number` of the form ρ`(`t_1`,` t_2`)` where ρ is one of `greater`, `greaterEqual`, `smaller`, `smallerEqual` and `notEqual` and let `I` be an instantiation set. The truth value V_p^I of `p` with respect to `I` is computed by the following sequence of steps:

i. Replace all variables in `p` which have a binding in `I`.
ii. Evaluate the evaluable terms in `p`.
iii. If t_1 or t_2 is an unbound variable then put V_p^I = `unknown`.
Otherwise put $V_p^I = t_1 \rho t_2$.

user-interface symbols: The user-interface symbols are built-in predicates, which address questions to the designer. Their truth values depend on the answer given. The user-interface symbols provide only a very limited means of dialogue between the user and the system. It boils down to a matter of question and answer.

The user-interface symbols will be substituted by a more sophisticated mechanism in the near future. For the time being, they are the only way to address the user.

The user-interface symbols start with 'ui' followed by a symbol starting with a capital. The currently implemented user-interface symbols are: `uiMessage`, `uiNumber`, `uiString`, `uiYesOrNo`, and `uiNoOrYes`. The built-in predicate `uiMessage` exists in unary and binary form. They are defined in the following ways:

PROCEDURE: Let `p` be an atom of type `string` of the form `uiMessage(t)` and let `I` be an instantiation set. The truth value V_p^I of `p` with respect to `I` is computed by the following sequence of steps:

i. Replace all variables in `p` which have a binding in `I`.
ii. If `t` is an evaluable term, then evaluate it.
iii. If `t` is an unbound variable then put V_p^I = unknown.
Otherwise show the contents of `t` to the user and put V_p^I = true.

The procedure for the atom `uiMessage(`t_1`,`t_2`)` of type `string`×`string` is analogous to the above procedure except that the contents of both t_1 and t_2 is shown to the user.

The truth value of the binary built-in predicate `uiNumber` is computed as follows:

PROCEDURE: Let `p` be an atom of type `string`×`number` of the form `uiNumber(`t_1`,`t_2`)` and let `I` be an instantiation set. The truth value V_p^I of `p` with respect to `I` is computed by the following sequence of steps:

i. Replace all variables in `p` which have a binding in `I`.
ii. Evaluate the evaluable terms in `p`.
iii. If t_1 is an unbound variable, then put V_p^I = unknown and stop.
Otherwise pose the contents of t_1 as a question to the user and let `c` be the answer.
iv. If t_2 is a variable, e.g. `X`, then add the binding `{X/c}` to `I` and put V_p^I = true. Otherwise put $V_p^I = t_2 \equiv c$

There is an other version of `uiNumber` which is a quadrary built-in predicate of type `string`×`number`×`number`×`number`. Its truth value is computed along the same line as the binary version. The difference lies in the third step which goes as follows:

iii. If t_1, t_3 or t_4 is an unbound variable, then put V_p^I = unknown and stop.
Otherwise pose the contents of t_1 as a question to the user and let `c` be the answer. If $c < t_3$ or $c > t_4$, then put V_p^I = false and stop.

The procedure for computing the truth value of the binary built-in predicate `uiString` is the same as for the binary predicate symbol `uiNumber`. The type of the second term is a `string` instead of a `number`.

The last user-interface symbol, being treated here, is the unary built-in predicate `uiYesOrNo`[7]. It asks the user a question to which he can respond 'yes' or 'no'. The definition is as follows:

PROCEDURE: Let `p` be an atom of type `string` of the form `uiYesOrNo(t)` and let `I` be an instantiation set. The truth value V_p^I of `p` with respect to `I` is computed by the following sequence of steps:

i. Replace all variables in `p` which have a binding in `I`.
ii. If `t` is an evaluable term, then evaluate it.
iii. If `t` is an unbound variable, then put V_p^I = `unknown` and stop.
Otherwise pose the contents of `t` as a question to the user and let `c` be the answer.
iv. If `c` ≡ 'yes', then put V_p^I = `true`. Otherwise put V_p^I = `false`

The built-in predicates presented above are those which are currently implemented in the system. This set can easily be extended by adding the appropriate procedure into the language specifications.

Built-in predicates occurring in a consequent. The section presents the built-in predicates that can be used in the consequent of a rule. The number is less than those used in the antecedent. They are: `value`, `hasPart`, predicate symbols which look like `isType` and predicate symbols which indicate a satisfaction of goals. The procedures of built-in predicates appearing in the consequent are less complex those appearing in the antecedent owing to the absence of the need to compute variable bindings. They may succeed or raise an error.

value: The binary built-in predicate `value` is used to access the internal structure of composite objects. It assigns values to attributes. Its first argument is a unary function that represents an object's attribute. Its second argument represents the value assigned to the attributes. Its procedure is defined as follows:

PROCEDURE: Let `p` be an atom of type `object` × `object` of the form `value(`t_1`,`t_2`)` and let `I` be an instantiation set. The following sequence of steps is performed:

i. Replace all variables in `p` by their bindings in `I`.
ii. If t_2 is an evaluable term, then evaluate it.
iii. If t_1 does *not* represent an object's attribute, then `error`.
Otherwise assign the value of t_2 to the attribute represented by t_1 and return ∅.

hasPart: The binary built-in predicate `hasPart` builds up the artifact structure. Its intuitive meaning is that the second argument is a part of the first argument. The

[7] Actually there is yet another built-in predicate named `uiNoOrYes`. It is equivalent to `uiYesOrNo`, but the default answer is 'no' instead of 'yes'.

type of `hasPart` is therefore `composite` × `composite`. In contrast with the built-in predicate `value`, which is not asserted to the fact-base, the built-in predicate `hasPart` is a derived (positive) fact. However, next to the 'normal' derivation procedure, an additional procedure is executed. It is defined as follows:

PROCEDURE: Let `p` be an atom of type `composite` × `composite` of the form `hasPart(`t_1`,`t_2`)`, let `I` be an instantiation set and let Γ be a fact-base. The following sequence of steps is performed:

i. Replace each variable in `p` by its binding in `I` obtaining the positive fact `f`.
ii. If `f` $\in \Gamma$, then add a time-stamp to `f` and stop.
iii. If t_1 and t_2 are object names, then add t_2 to the list of objects of t_1 and return `{f}`.

goal names: A number of proposition symbols is reserved for goal names. These symbols indicate that the goal, which was supposed to be satisfied by the current scenario, is indeed satisfied. They therefore halt the evaluation of the object-level scenario satisfying the goal. Control is given back to its parent (meta-level) scenario. They do not result in a derived fact:

PROCEDURE: Let `p` be a proposition symbol representing a goal name and let `I` be an instantiation set. If the set of hypotheses Γ_h is consistent with Γ, then the new state of the fact-base $\Gamma' = \Gamma \cup \Gamma_h$ is obtained by the state transition of Γ and Γ_h. Then the goal `p` has been satisfied and the execution of the current scenario is stopped. Return $\varnothing$

The next built-in predicate actually represents a whole class of predicate symbols. Their function is to *instantiate* composite objects.

object instantiation: The unary built-in predicate of the form `isFoo` instantiates an object of type `foo`. Such an atom is called a *object definition* atom. Its argument denotes the object's name. If the name already refers to an object of a different type in the object-base, that object will become a *multi-type* object. It will be extended with the attributes, operations and constraints of the prototype `foo`. Its attached procedure goes as follows:

PROCEDURE: Let `p` be an object definition atom of type `composite` of the form `isComposite(t)` and let `I` be an instantiation set. Object instantiation is achieved by the following sequence of steps:

i. Replace each variable in `p` by its binding in `I` obtaining the positive fact `f`.
ii. If `f` $\in \Gamma$, then add a time-stamp to `f` and stop.
iii. If `composite` is an existing type and `t` represents a new name, then create an object of type `composite` and name `t`, return `{f}` and stop.
iv If `composite` is an existing type and `t` is the name of an object, then the description of `t` is extended with the description of the prototype

`composite` and the type of `t` becomes `multiple`, return `{f}` and stop. Otherwise return `error`.

The last built-in predicate, being presented here, is given without explanation. Its purpose is to change the rule selection strategy being in use. It is extensively discussed in Chapter 8.

directive: The unary built-in predicate `directive` changes the current rule selection strategy to the one referred to by its argument. It is defined as follows:

PROCEDURE: Let `directive(t)` be an atom of type `symbol`. If `t` is a proper selector of a rule selection method, then the current rule selection method is set to `t` and return `true`. Otherwise return `false`.

6.5 Discussion

It seems reasonable to compare the object-level language with Prolog [Clocksin and Mellish, 1981], the most commonly used logic programming language. There are at least three fundamental differences between Prolog and the object-level language.

i. Prolog has implicit control of the reasoning process while ADDL offers the meta-level language to control the behaviour of the object-level language. Besides, the inference mechanism in Prolog is backward chaining with an automatic backtracking facility. ADDL's inference mechanism is forward chaining with a backtracking mechanism (over rules) that is explicitly controlled at the meta-level (see next chapter).

ii. Prolog only offers primitive objects as modeling entities. The prototype library in ADDL allows for modeling complex objects in a decomposed description of an artifact. ADDL objects have attributes and operations for representing internal properties. It uses functions to interface to an object's internal properties. Such a mechanism does not exist in standard Prolog.

iii. All clauses in a Prolog program are grouped in a single knowledge-base. Scenarios offer a facility to group rules together in separate functional units. The scenario mechanism improves performance because the search space is drastically reduced. The grouping also facilitates control.

Another difference between Prolog and the object-level language regards the implementation of *local operations*. The evaluable terms in the object-level language are kept only a single level deep. This restriction stems from my decision to keep the unification algorithm as simple as possible, but still sound. I certainly did not want to omit the *occur check* like is done in the most Prolog implementations causing the language to be *unsound* (see [Lloyd, 1987] pp. 43-45). Having evaluable terms with an unrestricted depth easily leads to a unification algorithm which is very expensive. I avoided this pitfall by allowing for evaluable

terms that may only have simple terms as their arguments.

An undesired property of the restriction to single-level terms is that it easily leads to a long sequence of 'equal' atoms. Let me explain this phenomenon by an example. Suppose I want the use the expression

```
plus(X,times(2,plus(Y,times(3,Z)))),
```

(equivalent to the infix expression X+2*Y+3*Z) in the antecedent of a rule (the variables X, Y, and Z are already bound). The expression cannot be written down in a single term and has to be split into the following formula:

```
equal(T1,times(3,Z)) & equal(T2,times(2,Y))
& equal(T3,plus(T1,T2)) & equal(T4,times(X,T3))
```

There are two ways to overcome this problem. The first solution is to define the expression as an operation belonging to one of the objects referred to by either X, Y, or Z. The expression itself is then replaced by a function call. This solution is undesirable if the operation does not reflect some general property of the object in question. The operation may only be used once or twice. Local operations defined in the scenario header offer a second solution to the problem. Their definition is as follows:

```
selector( optional parameters ) ) = { operation body }
```

The syntax of an operation body is given in Appendix 2. The following local operation computes the expression given above:

```
foo(X,Y,Z) = { 3 * Z + Y * 2 + X }
```

The operation can be applied by the function of type

```
number × number × number → number:   foo(_,_,_).
```

For the well known arithmetic operators an infix notation is used. The expressions are evaluated left to right. Local operations do not add to the functionality of ADDL. They are only an implementational issue introduced to aid the knowledge engineer. Chapter 8 on the implementation discusses local operations in more detail.

7

Process Knowledge Representation in ADDL

7.1 Introduction

The object-level language provides a means to represent knowledge about an artifact description. The reasoning involved is a forwardly directed activity. Initially the artifact model consists of a minimal description that is gradually extended as the design proceeds. The rules presented so far evaluate the model and based on this evaluation extend its contents. In order to direct this evaluation process, ADDL offers a *meta-level* language that is able to evaluate expressions from the object-level language. The meta-level language is the subject of this chapter.

Meta-level architectures can be classified into several types of architectures [Weyhrauch, 1980; Perlis, 1988; Rosenbloom, Laird, and Newell, 1988; Aiello and Levi, 1988; Sterling, 1988; Jackson, Reichgelt, and van Harmelen, 1989; Treur, 1989]. The last paper presents an architecture that exclusively deals with reasoning about design problems. A common property of meta-level architectures is that they consist of two levels. The object-level at which reasoning is performed about the application domain and the meta-level at which the object-level reasoning is monitored and controlled. The main advantage of such architectures is the separation between what the system knows (the object-level) and how this knowledge is applied (the meta-level). When I exercise this notion to the design process model, it becomes apparent that the designer's knowledge about design objects is represented at the object-level, and the designer's expertise how to apply this knowledge is found at the meta-level. Therefore, at the meta-level the knowledge about the *design process* is represented. This knowledge is used to

apply the knowledge about the *design object*, which is represented at the object-level.

7.2 Meta-level languages

A meta-level language consists both of names of atoms from the object-level language, literal meta-formulae and meta-rules. The meta-rules evaluate the information state of the design process (in short *process information state*) and assert *design goals*. The process information state consists on the one hand of information about the truth values of object-level atoms which is obtained from the object information state by a so called *reflection principle* or reflective transformation [Weyhrauch, 1980; Treur, 1991b]. On the other hand, the process information state consists of other process parameters whose values depend on the state of the design process. These parameters provide additional information about the process state of the design object description, and the design goals being satisfied so far. In Fig. 7.1, an example of a process information state is depicted.

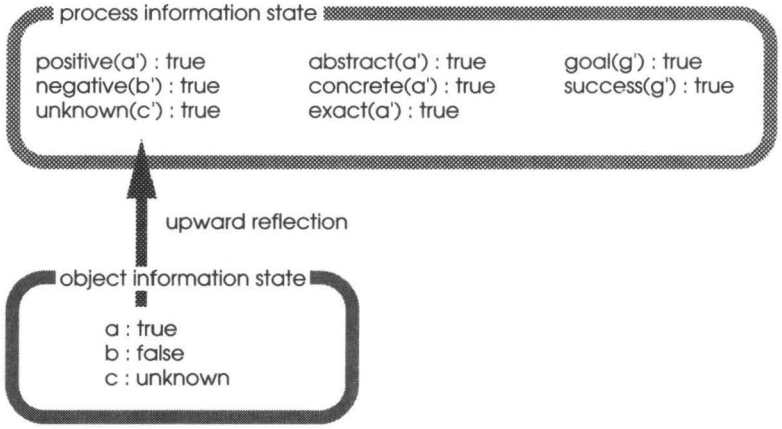


Fig. 7.1 Process information state and related object information state.

The design goals, that are asserted by the meta-rules, are solved by either object-level or meta-level scenarios. In this section the syntax of the meta-level language is presented. Since the construction of a meta-level language proceeds in accordance with the construction of an object-level language I try to be as brief as possible without omitting essential facts. For the definition of meta-rules, I introduce an alphabet, meta-terms, meta-atoms, and meta-formulae.

7.2.1 DEFINITION: An *alphabet* of a meta-level language consists of six classes of symbols:

1. *meta-variables*,
2. *meta-constant symbols*,
3. *meta-function symbols*,
4. *meta-predicate symbols*,
5. *connectives*,
6. *punctuation symbols*.

The first three classes are specific to a certain meta-level language. The last three classes are generic, i.e., they are the same for each meta-level language. The symbols obey the same lexical conventions as for object-level languages. The symbols of a meta-level language are also order-sorted typed. Meta-constant symbols are either names that refer to proposition symbols of an object-level language or they refer to object-level constant symbols. Meta-function symbols are either names that refer to n-ary (`n > 0`) predicate symbols of an object-level language or they refer to object-level function symbols.

There is a fixed set of eight generic meta-predicate symbols being used here. They are the following symbols starting with a lower-case letter. The second column gives an informal meaning of the meta-predicate symbol:

`positive`	*the argument is an atom that is true in the object information state*
`negative`	*the argument is an atom that is false in the object information state*
`unknown`	*the argument is an atom that is unknown in the object information state*
`abstract`	*the description of the argument as given by the object information state is abstract*
`concrete`	*the description of the argument as given by the object information state is concrete*
`exact`	*the description of the argument as given by the object information state is exact*
`goal`	*the argument needs to be satisfied*
`success`	*the argument has been satisfied*

The first three meta-predicate symbols are only used in the antecedent of a meta-rule. The meta-predicate symbol `goal` is only used in the consequent. All meta-predicate symbols are unary. The connectives are limited to `&`, `|`, `~`, and $\rightarrow$, meaning logical *and*, *or*, *not*, and *implication* respectively. The punctuation symbols are '`(`', '`)`', and '`,`'.

As stated before, the terms of meta-atoms refer to object-level atoms. In case of a nullary atom a meta-term is a meta-constant symbol and in case of an n-ary atom a meta-term is constructed by means of a meta-function symbol. The symbols of an object-level language are mapped to symbols of a meta-level language as will be shown in the next section. The result of this mapping is indicated by *primed*

symbols. For example, the object-level atom `p(X)` occurs as a meta-term in a meta-atom as `positive(p'(X'))`. The following definitions specify a two-level meta-term.

7.2.2 DEFINITION: A *simple meta-term of type* τ' is defined as follows:

1. Any meta-variable or meta-constant symbol of type τ' is a simple meta-term of type τ'.
2. If `f` is a n-ary meta-function symbol of type $\tau'_1 \times \cdots \times \tau'_n \rightarrow \tau'$ and t_i is either a meta-variable or a meta-constant symbol of type τ'_i, then $f(t_1, \ldots, t_n)$ is a simple meta-term of type τ'.

7.2.3 DEFINITION: A *two-level meta-term of type* τ' is defined as follows:

1. If `c` is a meta-constant of type τ', then `c` is a two-level meta-term of type τ'.
2. If `f` is a n-ary meta-function symbol of type $\tau'_1 \times \cdots \times \tau'_n \rightarrow \tau'$ and t_i is a simple meta-term of type τ'_i, then $f(t_1, \ldots, t_n)$ is a two-level meta-term of type τ'.

In an object-level language, the nesting of single-level terms is *not* allowed. As shown by the DEFINITION 7.2.3, a two level nesting of meta-terms is allowed in a meta-level language. Besides, simple meta-terms only occur as arguments to meta-function symbols. These symbols refer to object-level predicate symbols and meta-variables (which only occur in simple meta-terms) can thus only refer to object-level constant symbols. The following expressions are examples of two-level meta-terms: `a'`, `p'(X',b')`, `p'(f'(X',Y'),c')`. For convenience, if no confusion is expected, I use the same symbols for object-level predicate symbols on the one hand and meta-constant symbols and meta-functions on the other hand, but they are definitely different. In the sequel, I simply say meta-term instead of two-level meta-term.

Using the above definitions, one can define meta-atoms and typed meta-formulae. Since each meta-predicate symbol is unary, a meta-atom contains only a single meta-term. All meta-predicate symbols have the same type, viz. `objectAtom`.

7.2.4 DEFINITION: Let `p` be a meta-predicate symbol of type τ' and let `t` be a two-level meta-term of type τ'. Then `p(t)` is a *meta-atom.*

7.2.5 DEFINITION: If ϕ is a meta-atom, then both ϕ and $\sim\phi$ are *literal meta-formulae.*

The name of the type τ' in the DEFINITION 7.2.4 is `objectAtom`. The definitions of typed meta-formulae, meta-antecedents and meta-consequents are:

7.2.6 DEFINITION: A *typed meta-formula* is inductively defined as follows:

1. A literal meta-formula is a typed meta-formula.
2. If ϕ and ψ are typed meta-formulae, then ϕ & ψ, and ϕ | ψ are typed meta-formulae.
3. If ϕ is a typed meta-formula, then (ϕ) is a typed meta-formula.

7.2.7 DEFINITION: A *meta-antecedent* is a typed meta-formula restricted to the meta-predicate symbols `positive`, `negative`, `unknown`, `abstract`, `concrete`, `exact`, and `success`.

7.2.8 DEFINITION: A *meta-consequent* is a typed meta-formula restricted to the meta-predicate symbols `success`, `goal`, `abstract`, `concrete`, and `exact`. Disjunctions and negations are not permitted. The second induction rule is therefore omitted and the third one is replaced by:

2′. If ϕ and ψ are typed meta-formulae, then ϕ & ψ is a typed meta-formula.

Apparently, the syntax of the meta-level language is little more complex than the syntax of the object-level language. Having defined meta-antecedents, and meta-consequents I can now define meta-rules:

7.2.9 DEFINITION: If ϕ is a meta-antecedent and ψ is a meta-consequent, then $\phi \rightarrow \psi$ is a *meta-rule*.

For the meta-rules I also adopted the IF-THEN notation instead of the connective $\rightarrow$. A meta-level language is defined as follows:

7.2.10 DEFINITION: A *meta-level language* given by a (meta-level) alphabet consists of the set of meta-rules constructed from the symbols of the alphabet.

7.2.11 DEFINITION: A *meta-level scenario* `<name,meta-rule-set>` is a finite set of meta-rules that has a unique *name*.

Examples of meta-rules are:

```
1   IF positive(p'(X')) & unknown(q'(X',f'(X')))
    THEN goal(r'(X'))
2   IF negative(p'(X')) | negative(q'(Y')) & abstract(r'(Z'))
    THEN success(s')
```

The primed symbols are names of expressions in the object-level language that have a meaning with respect to the process information state. In the sequel, I omit the priming provided that it does not cause confusion. The meta-rules reason about these expressions. The declarative semantics of the meta-level language will be given in the next section. The procedural semantics is presented in §7.4. Finally, in §7.5 I discuss the global interaction between the two languages.

7.3 Declarative aspects of meta-level languages

7.3.1 Declarative semantics

In this section, the truth value of meta-formulae and meta-rules is discussed. As pointed out before, an object-level language allows for expressing what (negative or positive) facts are to be considered true. A meta-level language allows for reasoning about the implications and consequences of these facts [Weyhrauch, 1980]. The *domain* of a meta-level language is a process information state which consists of two parts. First of all, it consists of a meta-level description of an *object information state*, i.e., the epistemic information on the (object-level) facts about the design object model that are currently known or unknown. Secondly, it consists of a set of *parameters* that describe other aspects of the current state of the design process. They include a set of goals that need to be satisfied and a set of goals that have been satisfied. Furthermore, the parameters represent the process state of object-level facts. E.g., the anatomical description of a part of the design object model is *abstract*.

The *reflection principle* describes a transformation between an object information state and its meta-level interpretation in a process information state.

7.3.1 DEFINITION: Let Γ be an object information state, let Π be a set of process parameters, and $\mathfrak{R}$ describes a reflection principle, then a process information state $\mathfrak{I}$ is obtained as follows.

$$\mathfrak{I}(\Gamma, \Pi) = \mathfrak{R}(\Gamma) \cup \Pi$$

Here the union symbol is used to denote gluing of information states. The reflection principle provides a meaning for the meta-terms occurring in the meta-language. Therefore, I assume a fixed mapping from each meta-constant and meta-function in the domain of the meta-level language to an object-level proposition symbol. It is defined as follows by making use of the interpretation-mapping:

7.3.2 DEFINITION: Let L be an object-level language, let M be the meta-level language related to L, let $t' \in M$ be a simple meta-term of type τ' and let $t \in L$ be an object-level term of type τ. Then the definition of a *interpretation-mapping* from t' to t is:

1. If t' is a meta-constant symbol, then it is mapped to a constant symbol t.
2. If t' is a meta-variable, then it is mapped to a variable t.
3. If t' is a meta-term of the form $f'(s'_1, \cdots, s'_n)$, then it is mapped to a meta-term t of the form $f(s_1, \cdots, s_n)$.

7.3.3 DEFINITION: Let L be an object-level language, let M be the meta-level language related to M. A *naming link* is then defined by the mapping $nl : M \rightarrow L$ as follows:

1. If c is a meta-constant symbol being a meta-term of type objectAtom, then there exists a (object-level) proposition symbol p_c.
2. If f is a n-ary meta-function symbol of type $\tau'_1 \times \cdots \times \tau'_n \rightarrow$ objectAtom, then there exists an n-ary (object-level) predicate symbol p_f of type $\tau_1 \times \cdots \times \tau_n$, where each simple meta-term t'_i of type τ'_i being an argument of f is mapped to an (object-level) term t_i of type τ_i using the interpretation-mapping.

Notice that, although an interpretation of a meta-language may vary with respect to the domain specific meta-variables, meta-constant symbols, and meta-functions, it consists of a fixed set of generic meta-predicate symbols.

Before the definition of an interpretation of the meta-level language is given, I clarify some notational aspects. The set Γ represents an object information state, that can be divided into three subsets. The expression $\Gamma_+ \subseteq \Gamma$ denotes the set of literal formulae that are true in the object information state. Similarly, the expression $\Gamma_- \subseteq \Gamma$ denotes the set of literal formulae that are false and $\Gamma_? \subseteq \Gamma$ denotes the set of literal formulae that are unknown in the object information state. Therefore, the expression $\phi \in \Gamma_-$ means that ϕ occurs as a negative fact in Γ. The following definition gives an interpretation of a meta-level language which is related to an object-level language by a reflection principle. The definition consists of three parts. In the first part (A), I give the domain of the interpretation. In the second part (B), the meta-predicate symbols are given, that have a fixed meaning with respect to the object information state, and the third part (C) gives the meta-predicate symbols whose meaning depends on the process information state.

7.3.4 DEFINITION: Let L be an object-level language and let $nl(c) = \phi_c$ be a naming link from M to L. Then, an *interpretation* $\Im$ of a meta-level language M with respect to L and nl is defined as follows:

A1. It consists of a set (possibly empty) of literal facts Γ, which describes an *object information state* of L.

A2. It consists of a tuple $\Pi = <\Pi_d, \Pi_g, \Pi_s>$, that contains the *process parameters* of the interpretation where Π_d is a set of process state descriptors of object-level literal facts, Π_g is a set of asserted goal names, and Π_s is a set of satisfied goal names.

B3. For the meta-predicate symbols positive, negative and unknown in M an assignment to {*true*, *false*} is given as follows:

	$\phi_c \in \Gamma_+$	$\phi_c \in \Gamma_-$	$\phi_c \in \Gamma_?$
positive(c)	true	false	false
negative(c)	false	true	false
unknown(c)	false	false	true

C4. Let q be one of the meta-predicate symbols abstract, concrete and exact. Then the truth value of q(c) is true if $\phi_c \in \Gamma_+$ and q(c) $\in \Pi_q$; otherwise, its truth value is false.

C5. The truth value of goal(c) is true, if $\phi_c \in \Gamma_?$ and c $\in \Pi_g$; otherwise, its truth value is false.

C6. The truth value of success(c) is true if $\phi_c \in \Gamma_+$ and c $\in \Pi_s$; otherwise, its truth value is false.

The entries (A1) and (A2) define the domain of an interpretation, namely a *process information state* denoted by $\Im(\Gamma, \Pi)$. The truth values of the meta-predicate symbols defined by (B3) are independent of the (other) process parameters. They can directly be obtained from the object information state. The truth values of the meta-predicate symbols that describe other aspects of the process state of a design object are given by (C4-C6). They depend both on an object information state and a process parameter. The knowledge of the existence of a positive fact may sometimes be insufficient to proceed the design process. It is therefore convenient to add extra information to such a fact in the form of a process parameter. It expresses in more detail the design process state of a fact. Therefore, the meta-atom abstract(p) holds true in an interpretation if p is a positive fact and if there is a process parameter that tells that the description of p is abstract.

Finally, the meta-predicate symbols of (C5) and (C6) actually control the design process by asserting the names of design goals, that need to be satisfied, and by declaring the names of goals that have been satisfied. When the set of asserted goal names and the set of satisfied goal names are the same, then there are no further goals to be satisfied and the design process is finished. Before that happens, the design process proceeds by transitions of one process information state to another as shown below (here $\Im_i$ stands for $\Im(\Gamma_i, \Pi_i)$):

$$\Im_1 \Rightarrow \Im_2 \Rightarrow \cdots \Rightarrow \Im_i \Rightarrow \cdots \Rightarrow \Im_n$$

The interpretations $\Im_1 \cdots \Im_{n-1}$ describe partial models Γ_i of a design, but finally in the state $\Im_n$ the design Γ_n has been completed. This process is presented in detail in §7.4.1 about meta-level inference.

The truth value of a meta-formula can now be defined in a straightforward manner. The truth values for meta-atoms have been given, thus only the connectives need to be introduced.

7.3.5 DEFINITION: Let $\Im$ be an interpretation of a meta-level language *M*. If ϕ and ψ are meta-formulae in *M*, then the meta-formulae $\sim\phi$, $\phi \,\&\, \psi$ and $\phi \mid \psi$ and the meta-rule **IF** ϕ **THEN** ψ in *M*, can be given a truth value, *true* or *false*, (wrt $\Im$) as shown in the following table:

ϕ	ψ	$\sim\phi$	$\phi \,\&\, \psi$	$\phi \mid \psi$	**IF** ϕ **THEN** ψ
true	true	false	true	true	true
true	false	false	false	true	false
false	true	true	false	true	true
false	false	true	false	false	true

An example of the implication of the given definitions may enlighten the reader. Recall the previous example of meta-rules in §7.2. I now give the same meta-rules but this time without priming:

```
1   IF positive(p(X)) & unknown(q(X,f(X)))
    THEN goal(r(X))
2   IF negative(p(X)) | negative(q(Y)) & abstract(r(Z))
    THEN success(s)
```

The first rule reads as follows: if the mapping of `p(X)` is `true` in an object information state and if the mapping of `q(X,f(X))` is `unknown` in the same state, then the goal `r(X)` is added to the set of goal names that need to be satisfied. The second rule reads: if the mapping of `p(X)` is false in an object information state or the mapping of `q(Y)` is `false` in the same state and the mapping of `r(Z)` is both `true` and its process parameter is `abstract` in an process information state, then the goal `s` has been satisfied. More about meta-rule interpretation will be said in the next section on the procedural semantics of the meta-level language.

7.3.2 Meta-level inference

After giving the declarative semantics of the meta-level language, the reasoning mechanism needs to be explained and I need to declare how meta-rules are executed. Similar to the previous chapter, the procedural interpretation of a meta-level language consists of a control mechanism that derives conclusions from a set of meta-rules and a set of procedures that compute the variable bindings of these rules. In this section, meta-level inference is discussed. The next section gives the computational mechanisms. The inference mechanism applied at the meta-level uses the same derivation rules as at the object-level. They are not repeated here. The soundness of the meta-level language follows directly from its definition, since it is based on a subset of classical logic. The conclusions drawn by meta-level rules are completely different from object-level rules. The purpose of object-level inference is to extend the known facts about the design object description. Meta-level inference aims at extending the set of process parameters of the process information state. The conclusions drawn by meta-level rules are i) assertions of

goals that need to be satisfied in order to solve the design problem, ii) statements about the design process state of object-level facts, or iii) conclusions that a goal has been satisfied. These three categories are separately discussed below.

The assertion of `goal(g)` causes the activation of a scenario with the name `solve-g` (the definition below gives a more precise definition of the mechanism). I first explain the mechanism informally. Suppose I have the following meta-rule:

IF `positive(p(a))` **THEN** `goal(refinement(a))`

and `p(a)` is a positive fact in the object information state. Now the conclusion is that in order to proceed the design the goal `refinement(a)` needs to be satisfied. This is achieved by transferring control to either a meta-level scenario or an object-level scenario whose name is `solve-refinement`[8]. As soon as the goal has been satisfied control is given back to the current scenario. Knowing that the previous goal has been satisfied, the meta-level interpreter can choose a next goal using other meta-rules.

Recall the definitions of meta-level and object-level scenarios. Both kinds of scenario receive –upon activation– either an object information state or a process information state. In the sequel, an object information state is called a *world*. A world is constructed by selecting a subset of literal facts from a fact-base that contains all object-level literal facts being asserted during the design process. Furthermore, it contains the object description that are mentioned in the subset. Hence, a world is a subset of the fact-base and the object-base. In the sequel, I leave the object-base out of consideration. A scenario is activated through the assertion of a `goal` meta-predicate symbol by a meta-rule. The argument of a `goal` statement is an object-level atom whose predicate symbol is a goal name and whose arguments are names of constant symbols in the object-level language. These constant names are used to build the world of the activated scenario. If no arguments are given the scenario's world focuses on the entire object-information state. The following definition gives the mechanism for stating a goal that must be satisfied with respect to some world.

7.3.6 DEFINITION: Let Γ be a fact-base and let Π be a tuple of process parameters. Then, the meta-atom $goal(g(c'_1, \ldots, c'_n))$ being derived from a meta-rule is asserted through the application of the following sequence of steps:

i. A world Ω is created as follows: for each literal fact $p \in \Gamma$, if one or more of $c_1 \cdots c_n$ occurs in p, then $p \in \Omega$.

ii. A scenario Λ_g with the name `solve-g` is activated focusing on the world Ω.

---

[8] If a goal causes the activation of an object-level scenario, the involved reasoning process strictly speaking concerns object-level inference rather than meta-level. For sake of simplicity, I treat 'object-level goals' and 'meta-level goals' on even terms.

iii. The goal name `g` is added to the set Π_g of asserted goal names of Π that need to be satisfied. This causes a state transition of Π to a next process information state Π'.

In other words, the assertion of a goal statement causes deactivation of the current scenario and activation of a new scenario. The aim of the new scenario is to satisfy the asserted goal. The construction of a world that is a subset of the fact-base is called the *world mechanism*. This mechanism is best illustrated by an example. Suppose a fact-base consists of the following literal facts:

`p(a)`	`q(a,b)`	`r(a,b,c)`
`p(b)`	`q(b,c)`	`r(a,c,c)`
`p(c)`	`q(a,c)`	`r(c,c,c)`

and the meta-atom `goal(g(a))` is asserted. Upon activation of a scenario named `solve-a`, the following world is constructed from the above fact-base:

`p(a)`	`q(a,b)`	`r(a,b,c)`
		`r(a,c,c)`
	`q(a,c)`	

The world contains only literal facts that have `a` as an argument. Object names other than `a` are imported to the world, if they occur in an atom where `a` occurs as well, e.g. `q(a,c)`. As a consequence, there can be no unary literal fact whose term is unequal to `a`. The purpose of the world mechanism is to make the derivation of an antecedent more efficient since fewer literal facts have to be examined.

It may be useful to implement an enhanced version of the world mechanism that focuses on a specific part of the design object model. In that case, the world ought to contain literal facts that have `a` or a component of `a` as an argument. The components of an object are described by the built-in predicate symbol `hasPart` (see §6.5.3). So far, such an extended world mechanism has not yet been implemented.

When a newly activated scenario is a meta-level scenario, its world is part of its process information state, i.e., it is the object information state to which the process information state refers. When a newly activated scenario is an object-level scenario, then the world is simply the object information state of that scenario. Fig. 7.2 depicts the world mechanism. A world is strictly 'read-only'. It can not be modified by a scenario; it can only be viewed.

The derivation of a meta-atom `success(g)` expresses the satisfaction of a goal that has been asserted earlier in the design process. The name of the goal is indicated by the argument `g`. The derivation procedure proceeds as follows:

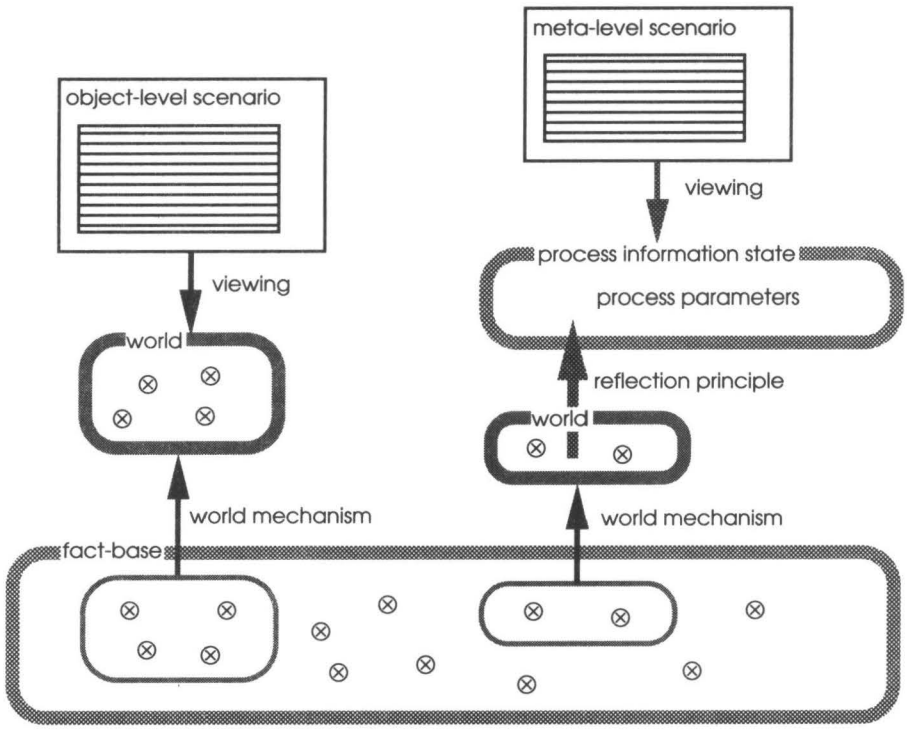


Fig. 7.2 The world mechanism creates a view on a fact-base on which a scenario focuses. The symbol ⊗ denotes a literal fact.

7.3.7 DEFINITION: Let Λ_g be the active scenario and let Π be a tuple of process parameters. Then, the meta-atom `success(g)` being derived from a meta-rule of Λ_g is asserted through the application of the following sequence of steps:

i. The goal name `g` is added to the set Π_s of goal names of Π that have been satisfied. This transforms Π into a next process information state Π'.

ii The scenario Λ_g is deactivated and control is given to the 'source' of the goal.

The word 'source' may sound a little cryptic. The use of this word is due to the fact that the goal that has been satisfied may either be asserted by another meta-level scenario, or it may be the top-level goal of the design process. In the former case, the scenario that asserted the goal statement is reactivated. In the latter case, the design problem has been solved and the fact-base contains a complete description of the design object model.

The second category of conclusions inferred by meta-level rules are statements about the process state of a positive fact. They assert that the anatomical description of (a part of) the design object is abstract, concrete or exact. These conclusions are stored as process parameters in the process information state. It proceeds as follows:

7.3.8 DEFINITION: Let `s` be one of the meta-predicate symbols `abstract`, `concrete`, or `exact` and let Π be a tuple of process parameters. Then the meta-atom `s(c)` being derived from a meta-rule is added to the set Π_d of process state descriptors of Π. This transforms Π into a next process information state Π'.

As an example, I give a meta-rule that illustrates the above presented concepts. It reads as follows: if `p'(a')` maps to a literal fact `p(a)` in an object-level information state and its process parameter is `abstract` in a process information state and the goal `basicGeometry` has been satisfied, then the process parameter of `p(a)` is also concrete. Notice that for convenience the symbols are not primed:

```
IF abstract(p(a)) & success(basicGeometry(a))
THEN concrete(p(a))
```

Notice that the process parameters are mutually independent. Therefore, a fact may have –as shown by the example– more than one parameter being set. Although it may seem natural that when a concrete description of an artifact has been established, its description is also abstract, this is not necessarily the case in ADDL unless stated explicitly.

7.4 Procedural aspects of meta-level languages

The computational mechanism underlying a meta-level language is similar to the mechanism presented in §6.4.3 and §6.4.4. A *meta-resolution tree* is constructed and traversed in an analogous manner. The mechanism is simplified because, (i) it only has to deal with the classical truth values *true* and *false* and (ii) the unification algorithm is solely applied at the object-level. During traversal, meta-variables are bound to meta-constant symbols as follows. Meta-constant symbols have a mapping to object-level constant symbols by means of the naming link. The application of the unification algorithm at the object-level results in a binding of object-level variables to constant symbols. These constant symbols are on their turn linked by a mapping from meta-constant symbols. These are thus the bindings of the corresponding meta-variables. This mechanism is illustrated in Fig. 7.3 where a meta-variable `X'` is bound to a meta-constant `c'`.

The derivation procedure, that infers the antecedent of a meta-rule, traverses a meta-resolution tree until it has found a truth value for the root of the tree. The procedure is biased towards *true*. Therefore, it only concludes that the truth value of the tree is *false* when it has been searched exhaustively by a backtracking

Fig. 7.3 The bindings of the meta-variables are computed at the object-level. A double arrow indicates a naming link and a single arrow stands for the binding of a variable.

algorithm. A meta-resolution tree is constructed very similar to an object-level tree. Nearly the same procedures as given in §6.5 are used. However, a few modifications are needed.

In the first place, the second parameter, `t` of the procedure for adding a leaf, `addLeaf (n,t)`, is a *literal meta-formula* rather than a literal formula. Secondly, the procedure `makeTree (s)` given in §6.5 for constructing a resolution tree for a meta-antecedent is modified by replacing in the first case-statement `literal formula` by `literal meta-formula`. The procedure described by the following pseudo code is used to construct a meta-resolution tree.

PROCEDURE: Let `s` be a meta-antecedent represented by a sequence of tokens `s`, let `t` be a token, and let `n` and `st` be nodes. Then a *meta-resolution tree* being the tree of nodes `n` is built up as follows:

```
makeTree (s)
   n = nil;
   t = nextToken (s);
   do {
      switch (t) {
      case 'literal meta-formula':
         n = addLeaf (n,t); break;
      case '&':
      case '|':
         n = root (n);
         n = addNode (n,t); break;
      case '(':
         st = makeTree (s);
         n = addTreeLeaf (n,st); break; }
      t = nextToken (s); }
   while (t != nil && t != ')');
   return root (n); }
```

Traversal of a meta-resolution tree progresses analogous to traversal of a resolution tree. When a node becomes false the algorithm will backtrack over previous nodes.

The leaves of the tree contain literal meta-formulae. Their truth value is obtained as follows. The meta-term of a (unary) literal meta-formula is mapped to an object-level predicate symbol by the resolution principle. If the meta-term is a meta-function, then the simple meta-terms are likewise mapped to object-level terms. The outcome is an object-level atom whose truth value can be computed using the method for a positive leaf as is presented in §6.5.2. The behaviour of this method depends on the *direction of search*. It applies the unification algorithms and it fills in an *instantiation set*. The truth value of the literal meta-formula can then easily be determined. It depends on the truth value of the (object-level) atom and the process parameters of the process information state. It is defined as follows:

7.4.1 DEFINITION: Let `l` be a leaf with the literal meta-formula `m(c)`, let `I` be an instantiation set let `d` be the direction of search, and let Π be a process information state. Then, the truth value of `l` is obtained by performing the following sequence of steps.

i. Map `c` to an object-level atom p_c.
ii. Compute $V_{p_c}^{I,d}$, the truth value of p_c with respect to the world of Π.

Step (ii) uses the definition of an interpretation of a meta-language to compute the truth value of a literal meta-formula. This approach is successful because a meta-language has a fixed set of meta-predicate symbols. Each individual meta-predicate symbol has its private method for computing the truth value.

The truth values of an and-node and an or-node occurring in a meta-resolution tree are acquired as follows:

7.4.2 DEFINITION: Let `n` be an and-node and let n_l and n_r be its left successor and its right successor, respectively, let `I` be an instantiation set and let `d` be the direction of search. The truth value of `n` with respect to `I` is obtained by performing the following steps:

i. If `d` ≡ `backward`, then go to step iv.
ii. Compute V_l^I, the truth value of the left successor of `n`.
iii. If $V_l^I \equiv$ `false` then V_n^I = `false` and stop.
iv. Compute V_r^I the truth value of the right successor of `n`.
v. If $V_r^I \equiv$ `false`, then `d` = `backward` and go to step ii.
Otherwise V_n^I = `true`.

7.4.3 DEFINITION: Let `n` be an or-node and let n_l and n_r be its left successor and its right successor, respectively, and let `I` be an instantiation set. I call σ the state of `n`. The truth value of `n` with respect to `I` is obtained by performing the following steps:

i. If `d` ≡ `backward` and $\sigma \equiv$ `right`, then go to step iv.
ii. Compute V_l^I, the truth value of the left successor of `n`.
iii. If $V_l^I \equiv$ `true`, then σ = `left`, V_n^I = `true`, and stop.

iv. Compute V_r^I the truth value of the right successor of `n`.

v. If $V_r^I \equiv$ `true`, then σ = `right`, V_n^I = `true`.
Otherwise V_n^I = `false`.

The meta-level only deals with the two classical truth values *true* and *false*. The computational methods are therefore slightly simpler. Computations that involve variable bindings are strictly performed at the object-level. The truth value of a meta-resolution tree can be computed as follows:

7.4.4 DEFINITION: Let `T` be a meta-resolution tree and let `n` be the root of the tree and let `d` be the direction of search. The truth value of `T` is obtained by computing the truth value of `n` with `d` = `forward`.

The behaviour of the computational mechanism regarding the derivation procedures of a consequent are fully described by the declarative description of meta-level inference. They need no further explanation here.

7.5 Global interaction between the two levels

An ADDL program consists of a number of meta-level and object-level scenarios, and a prototype library. The selection of scenarios is controlled by a top-level meta-level scenario that is invoked by the designer. The top-level scenarios states design goals that are to be satisfied by either meta-level or object-level scenarios. The meta-level scenarios extend and transform the set of process parameters of a process information state. The object-level scenarios extend a fact-base that describes the design object status. The fact-base and the set of process parameters are initially empty[9].

Fig. 7.4 shows the application of two meta-level scenarios (`top` and `sub`) and three object-level scenarios (`ob1`, `ob2` and `ob3`). For simplicity I omit the world mechanism from the figure. The process information state $\Im(\Gamma, \Pi)$ described by the fact-base Γ and the set of process parameters Π, is initially associated with `top`. During the design process the process information state is subject to a number of state transitions leading from $\Im(\Gamma_0, \Pi_0)$ to $\Im(\Gamma_4, \Pi_3)$. Since a process information state is defined as a function of Γ and Π, viz. $\Im(\Gamma, \Pi) = \Re(\Gamma) \cup \Pi$, the registration of the individual modifications to Γ and Π is sufficient to obtain a new process information state.

The first meta-rule of `top` asserts a goal that causes the activation of `ob1`. The application of the rules of `ob1` results in a new state of the fact-base (Γ_1). The rules are applied until the goal has been satisfied. Then, control is given back to `top`, which causes a state transition from Π_0 to Π_1. The next goal causes the activation

[9] This does not imply that there is no information in the process information state. Obviously, the process information state contains the information that literal facts are unknown.

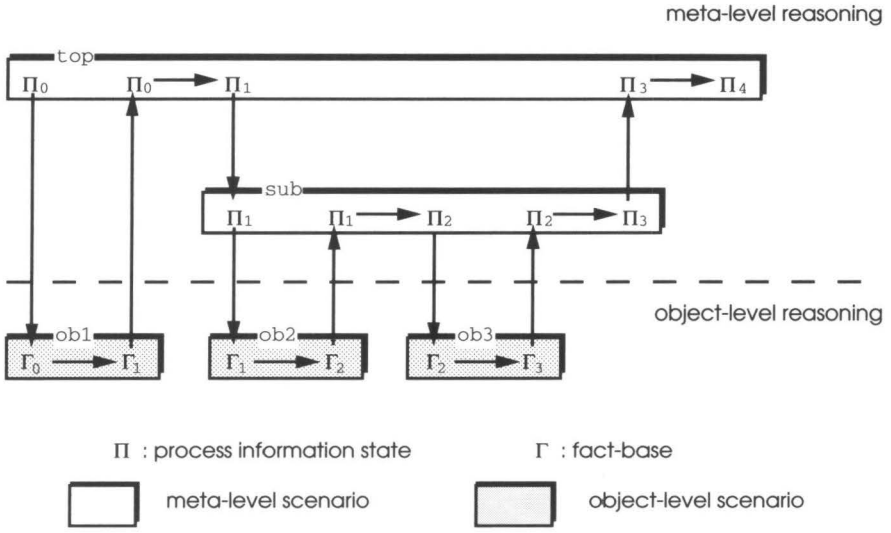


Fig. 7.4 Multi-level reasoning in ADDL.

of the meta-level scenario `sub`, which subsequently activates the object-level scenarios `obj2` and `obj3`. After the application of the rules of the first, the state of the fact-base becomes Γ_2. This reasoning process continues along the same line and the final states of the set of process parameters and the fact-base are Π_4 and Γ_3.

So far, the interface and interaction between the meta-level and the object-level has been discussed. Now, I give some example-scenarios which implement the behaviour as described by Fig. 7.4. These scenarios have a slightly different name, the string `solve-` is 'prepended' to the names given in the figure. Thus, a scenario named `solve-top` satisfies a goal named `top`. Notice that the knowledge represented by the scenarios is not particularly relevant to a certain problem. Their purpose is to clarify the flow of control between scenarios. Prior to the presentation of a scenario its signature Σ is given.

The following is the signature of a meta-level scenario with the name `solve-top`. The symbols of the signature that refer to symbols of the related object-level language appear with an accent. In the sequel, I omit the priming of symbols in signatures of meta-level scenarios. A short-hand notation for types is useful to reduce the otherwise lengthy types of (meta-) functions and (meta-) predicate symbols. Thus, the abbreviation `SY` stands for the type `symbol`. In Σ`(solve-top)`, I introduce (among others) the meta-constant symbols `ob1'` and `top'`. They are both goal names. The second goal name is currently being satisfied

by the meta-level scenario `solve-top`, that asserts the first goal name as a new (sub) goal. Furthermore, the signature introduces the meta-function symbols `p'` and `sub'`. The first refers to a predicate symbol of an object-level language. The second is a goal name asserted by `solve-top`.

Σ(solve-top)

Type	**Notation**	**Parent**
objectAtom	A	object'
symbol'	SY'	primitive'
Meta-constant	**Type**	**Comments**
ob1', top'	A	refer to an object-level proposition symbol
Meta-function	**Type**	**Comments**
p', sub'	SY'→A	refer to an object-level predicate symbol
Meta-predicate	**Type**	
abstract, exact, goal, success, positive, unknown	A	

Meta-rules

1 **IF** unknown(p(ω)) **THEN** goal(ob1)
2 **IF** positive(p(X)) **THEN** abstract(p(X)) & goal(sub(X))
3 **IF** exact(p(X)) **THEN** success(top)

It is the top-level scenario that corresponds to the meta-level scenario `top` in Fig. 7.4. It controls the overall problem solving process. Its first rule asserts the meta-atom `goal(ob1)` which causes the activation of an object-level scenario named `solve-ob1`. Since `ob1` has no arguments, it does not specify a particular world. The default action is that the world encompasses the entire fact-base (being empty).

After the application of `solve-top`'s first rule, the meta-level interpreter transfers control to `solve-ob1` and `solve-top` becomes *inactive*. Note that there is a difference between a scenario that is becoming inactive and a scenario that is terminating. In the former case, the interpreter maintains a scenario's state, while in the latter case it merges a scenario's set of hypotheses with the fact-base or the set of process parameters depending on the kind of inference (object or meta). The signature of `solve-ob1` looks as follows:

Σ(solve-ob1)

Type	**Notation**	**Parent**
symbol	SY	primitive

Constant	Type
symbol	S

Predicate	Type
typeFor	SY×SY
p	SY
ob1	

Rules

1 **IF** typeFor(X,symbol) **THEN** p(X) & ob1

The binary built-in predicate `typeFor` instantiates an object. It instantiates its first argument to an object of a type denoted by its second argument. argument and by appending a number to this symbol it creates a unique new symbol. In this case, it generates the symbol 'symbol1', because it is the first application of the function to the symbol 'a'. The scenario's rule simply asserts two facts to the set of hypotheses, viz. `p(symbol1)` and `ob1`. The scenario causes a state transition of the fact-base from Γ_0 to Γ_1 consisting of the singleton `{p(symbol1)}`.

After application of `solve-obj1`, the set of process parameters transforms from Π_0 to Π_1 containing `success(ob1)` since it is now known that `goal(ob1)` has been satisfied. Now, `solve-top` becomes active again. It applies its second rule, which concludes that the description of `p(symbol1)` is abstract and which asserts a new goal `goal(sub(symbol1))`. It causes the activation of the meta-level scenario `solve-sub`:

Σ(solve-sub)

Type	Notation	Parent
objectAtom	A	object
symbol	SY	primitive

Meta-function	Type	Comments
p, ob2, ob3, sub	SY→A	refers to a predicate symbol

Meta-predicate	Type
abstract, concrete, exact	
goal, success	A

Meta-rules

1 **IF** abstract(p(X)) **THEN** goal(ob2(X))
2 **IF** success(ob2(X)) **THEN** concrete(p(X)) & goal(ob3(X))
3 **IF** success(ob3(X)) **THEN** exact(p(X)) & success(sub(X))

The world of `solve-sub` consist of the literal fact `p(symbol1)`. It is created by selecting the literal facts that are concerned with the object `symbol1`. The

scenario's first rule asserts the goal `goal(ob2(symbol1))`. If this goal has been satisfied (by the scenario `solve-ob2` below), the condition of the second rule succeeds. The second rule states that the anatomical description of `p(symbol1)` is concrete. Obviously, the implicit notion is that this has been achieved by the scenario `solve-ob2` when solving the goal `goal(ob2(symbol1))`. Furthermore, it asserts a new goal `goal(ob3(symbol1))`. When this goal has been satisfied (by the scenario `solve-ob3` below), the third rule concludes that i) the anatomical description of `p(symbol1)` has been made and ii) that the aimed goal of the scenario has been reached.

The first and the second rule of `solve-sub` cause the activation of the object-level scenarios `solve-ob2` and `solve-ob3`. The first has the following signature:

Σ(solve-ob2)

Type	**Notation**	**Parent**
symbol	SY	primitive

Constant	**Type**
symbol	SY

Predicate	**Type**
p, ob2	SY
typeFor, q	SY×SY

Rules

1 **IF** p(X) & typeFor(Y,symbol)
THEN p(Y) & q(X,Y) & ob2(X)

The world of `solve-ob2(symbol1)` consists of `p(symbol1)` and the built-in predicate `typeFor(Y,symbol)` instantiates an object of type symbol, viz. `symbol2`. Therefore, the set of hypotheses of `solve-ob2` becomes

```
{ p(symbol2), q(symbol1,symbol2), obs(symbol1) }.
```

The set is merged with the fact-base after termination of `solve-ob2`. The set of process parameters is extended with `success(ob2(symbol1))`.

The signature of `solve-ob3` is as follows:

Σ(solve-ob3)

Type	**Notation**	**Parent**
symbol	SY	primitive

Constant	**Type**
symbol	SY

Predicate	**Type**
p, ob3	SY
typeFor, q	SY×SY
r	SY×SY×SY

Rules

1 **IF** q(X,Y) & typeFor(Z,symbol)
THEN p(Z) & r(X,Y,Z) & ob3(X)

The application of solve-ob2 results in the extended fact-base Γ_2 which in turn is extended by solve-ob3 obtaining Γ_3. The last one consists of the following facts (enlisted in the order of assertion and amplified with the fact-base that was obtained at that time):

```
p(symbol1)                          Γ1
p(symbol2)
q(symbol1,symbol2)                  Γ2
p(symbol3)
r(symbol1,symbol2,symbol3)          Γ3
```

Notice that the predicate symbols that refer to a satisfied goal name are actually a kind of built-in predicate symbols that 'promote' themselves to process parameters. For example, the conclusion ob3(symbol1) in the rule above appears as the meta-level atom success(ob3(symbol1)) in the set of process parameters.

After the application of the last rule of solve-sub, the interpreter gives control back to solve-top. An exact anatomical description of p(symbol1) has been made and the top-level goal has been satisfied. The set of process parameters cumulates as the design process proceeds. The obtained set Π_4 consists of the following items (again in order and with the set of process parameters being reached):

```
goal(ob1)
success(ob1)                        Π1
abstract(p(symbol1))
goal(sub(symbol1))
goal(ob2(symbol1))
success(ob2(symbol1))               Π2
concrete(ob2(symbol1))
goal(ob3(symbol1))
success(ob3(symbol1))               Π3
exact(p(symbol1))
success(sub(symbol1))               Π4
success(ob1)
```

Notice that the history of the set of process parameters of the process information state Π_4 corresponds to the flow of multi-level reasoning depicted in Fig. 7.4.

7.6 Discussion

In [van Harmelen, 1989] a classification of meta-level architectures is given. The author distinguishes among three types of architectures, viz. *object-level* inference systems, *mixed-level* inference systems and pure *meta-level* inference systems. Systems that belong to the first category have their main activity at the object-level. The reasoning takes place at the object-level. The meta-predicates are only used to define a fixed order in which the object-level rules are searched. The other extreme are pure meta-level inference systems where the reasoning mainly takes place at the meta-level. The object-level search space is minimized. The selection of an object-level rule is fully determined at the meta-level.

A combination of the previous two are mixed-level inference systems. Reasoning takes equally place at both levels. The ADDL architecture is an example of a mixed-level architecture. The strategic decisions are taken at the meta-level. These decisions are then carried out at the object-level. A system that is based on such an architecture is also called a *subtask management* system. The meta-knowledge is used to subdivide the design task in a number of *subtasks*. Each of the subtasks is then either subdivided into other subtasks or solved by an object-level scenario. After completion of an object-level scenario control is given to the meta-level scenario that either states a new subtask or gives control to a higher level meta-level scenario.

The meta-level language has only a fixed number of meta-predicate symbols. However, this set is easily extendible by the system programmer. The current set is sufficient for the kind of design systems that have been implemented so far. These systems have in common that they operate on a rather small design problem with a limited number of solutions. As a consequence the number of scenarios solving these problems is quite small as well (approximately twenty). I have the strong impression that if a larger more complicated design problem is tackled, the number of scenarios will grow significantly. Hence, the control becomes more complex and the meta-level scenarios play a more important role. The implementation of a system for designing a testing device for tyres confirmed this opinion [Maurice, 1991; Veerkamp and ten Hagen, 1991].

8

Implementation

8.1 Introduction

The current implementation of ADDL consists of a compiler, an interpreter, and a run-time environment. The compiler embodies a lexical analyzer, a parser and a code generator. The interpreter consists of a meta-level interpreter and an object-level interpreter. The development of the compiler started in 1987 by Monique Megens who worked as a Master's student at CWI. She implemented a lexical analyzer and a parser for ADDL (at that time called IDDL) [Megens, 1987]. Since then the syntax of the language has changed drastically. The major parts of the lexical analyzer and the parser are still in use, though adapted to the modified specifications.

The implementation of ADDL evolved over four years and is still in progress. It is written in Smalltalk-80 [Goldberg and Robson, 1983; Goldberg, 1984] and the ADDL code is compiled to Smalltalk code. The Smalltalk programming environment is used not only because it is an object-oriented language allowing for easily modelling of ADDL objects. Smalltalk's major advantage is its suitability for rapid prototyping. During the last few years, several versions of ADDL have been operational. The easy modification and reuse of Smalltalk code allows for designing the language while an experimental version of the current state of design is operational.

Another advantage of Smalltalk is its extremely useful debugging facility. On the one hand it simplifies the detection of programming errors while implementing the ADDL environment. On the other hand it greatly benefits to the debugging of ADDL code itself, since the ADDL code runs within the Smalltalk environment. The ADDL programmer can make full use of the Smalltalk debugger without knowing

too much about Smalltalk itself. Throughout the chapter, I give some sample Smalltalk methods that explain parts of the implementation. Therefore, the next section provides a short introduction to Smalltalk. The remainder of this chapter aims at giving some insight into the implementational aspects of ADDL. First of all, the ADDL compiler is described in §8.3. Then, in §8.4 the interpreter is presented. A prototype of an experimental CAD system is given in §8.5. Finally, §8.6 concludes this chapter with a discussion about the implementation.

8.2 Introduction to Smalltalk

Smalltalk-80 has an extremely simple syntax. Basically, it amounts to sending a message to an object as follows:

```
anObject doSomething.
```

The object `anObject`, called the *receiver*, performs some action and exits with a return value. Messages can have one or more arguments when the method selector ends with a colon. The following two messages have one and two arguments respectively:

```
anObject doSomethingWith: anotherObject.
anObject doSomethingWith: object1 and: object2.
```

Note that smalltalk expressions are terminated by a period. It is not uncommon that messages are nested. The token `:=` denotes an assignment and the token `↑` exits with the succeeding object as return value. Local variables are declared between bars (`|`) and comment appears between double quotes (`"`). Lastly, code between square brackets is called a *block*. When a block is passed as an argument to a message, the evaluation of the block is postponed. It is executed by the receiver of the message as soon as the block is used. The following Smalltalk method **`lex2`** contains all of the concepts introduced above:

```
lex2    "instance method of LexicalAnalyzer"
"[I][F]=> either VARIABLE or IF"
   | char |
   char := self nextChar.
   (AlphaNumeric includes: char)
      ifTrue: [↑self lex9].
   ↑self lex3
```

The keyword `self` refers to the object that executes the above method. The object `AlphaNumeric` is a global instance of class `Set` that contains all alphanumeric characters. The above method asks for the next character and stores it in the local variable `char`. If this character is included in the set of alphanumeric characters, then the result of sending the message `lex9` to itself is returned. Otherwise, the method returns the result of the message `lex3`.

8.3 The ADDL compiler

Fig. 8.1 depicts the organization of the ADDL compiler in a block diagram. The lexical analyzer scans the stream of ADDL source code and separates it into *tokens*. The tokens are keywords such as **IF**, **THEN**, predicate symbols and so on. The token stream is the input of the next module, the *parser*.

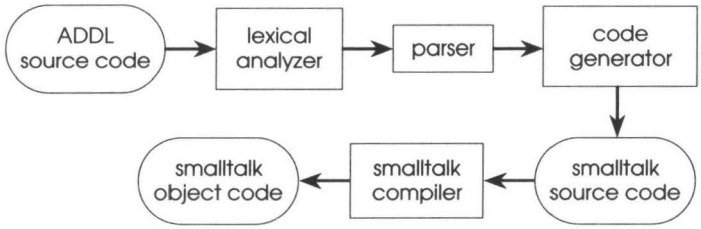


Fig. 8.1 Organization of the ADDL compiler.

The parser groups tokens together in accordance with their syntax into a *parse tree*. For instance, an atom is a syntactic structure consisting of the tokens: predicate symbol, left parenthesis, list of terms, and right parenthesis. The leaves of the tree are the tokens. Fig. 8.2 shows the part of a parse tree concerning an atom. During the phase of code generation the parse tree is traversed and the syntactic structures are translated into Smalltalk source code. The Smalltalk source code is compiled by the Smalltalk compiler into Smalltalk object code.

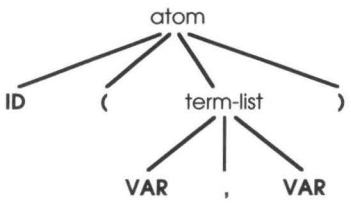


Fig. 8.2 Part of a parse tree concerning the atom `p(X,Y)`.

The ADDL source code consists of meta-level scenarios, object-level scenarios, local operations and prototype definitions. The prototype library has its own interface to Smalltalk without the need of a compiler. The interface to the prototype library is presented in §8.4. The three modules of the ADDL compiler are discussed in the following three sections. The Smalltalk compiler will *not* be discussed.

8.3.1 The lexical analyzer

While there are three separate parsers and code generators for meta-level scenarios, object-level scenarios, and local operations, there is only a single lexical analyzer. The lexical analysis of ADDL code is done by picking up tokens from the stream of characters representing a scenario or a function. A strategy similar to *Lex* –a lexical analyzer generator– is employed [Lesk and Schmidt, 1986]. As soon as a token is recognized by the lexical analyzer, it is added to a stream of tokens. A token is represented by a name such as `OPERATOR` and a value which stores the token string such as '+'. The lexical analyzer recognizes tokens by means of *rules*. Each rule contains a *regular expression* that matches the string belonging to a token. The rules and token names are given in Table 8.1. I assume that the reader is familiar with regular expressions first studied by Kleene [Kleene, 1956].

The lexical analyzer is implemented with the aid of *transition diagrams*, which are derived from the regular expressions. A transition diagram consists of an initial state, normal states, and accepting states. Transition from one state to another is done on a certain input character or a certain set of input characters. E.g. Fig. 8.3 shows the transition diagram derived from the specification of a variable, the specification of `IF` and the specification of `THEN`. From the initial state `0` there is a transition to state `1` on the letter `I`. There is a transition to state `4` on the letter `T` and there is a transition to state `9` on all other letters. A transition labeled ε denotes that the concerned token is recognized. E.g. if from state `2` the input is not a letter or a digit the token `IF` is matched. Otherwise state `9` will be reached.

Transition diagrams can easily be translated to Smalltalk code. A Smalltalk *class* `LexicalAnalyzer` contains for each state in the diagram, a corresponding instance method. For example, the state `2` is translated to the following code:

```
lex2    "instance method of LexicalAnalyzer"
"[I][F]=> either VARIABLE or IF"
   | char |
   char := self nextChar.
   (AlphaNumeric includes: char)
      ifTrue: [↑self lex9].
   ↑self lex3
```

which examines the next character from the character stream. If this character is an alphanumeric character –a letter or a digit– then there is a state transition to state `9`. Otherwise, there is a state transition to state `3`, which is an accepting state. An accepting state is implemented as follows.

```
lex3    "instance method of LexicalAnalyzer"
"[I][F][]=> IF"
   self addToken: #IF.
   ↑self char
```

Rule	Token name
\"([" / "]?[^"]?)*\"	*skipped comment*
"+"|"-"|"/"|"*"|"**"	OPERATOR
"IF"	IF
"THEN"	THEN
[A-Z][a-zA-Z0-9]*	VARIABLE
:=	ASSIGN
:[a-z][a-zA-Z0-9]*	FNAME
[a-z][a-zA-Z0-9]*	ID
[0-9]+(.[0-9]+(e[+-]?[0-9]+)?)?	NUMCONSTANT
'([' / ']?[^']?)*'	STRINGCONSTANT
$[^]	CHARCONSTANT
"#"	HASH
"&"	AND
"|"	OR
"~"	NOT
"("	LPS
")"	RPS
"["	LSB
"]"	RSB
","	COMMA
"="	IS
"{"	LCB
"}"	RCB
";"	SEMI
[\n\t]	*white space*

Table 8.1 The names of the tokens that are recognized by the lexical scanner are shown in the second column of the table. The first column shows the corresponding regular expression.

The recognized token is added to the token stream and the current character is returned. This character is the first character of a new token. White space –i.e. spaces, tabs and new lines– is skipped. The complete transition diagram of the lexical analyzer is depicted in Appendix 1.

8.3.2 The parser

The parser reads the token stream generated by the lexical analyzer. The stream is converted to a parse tree using grammar rules if the stream obeys the grammar

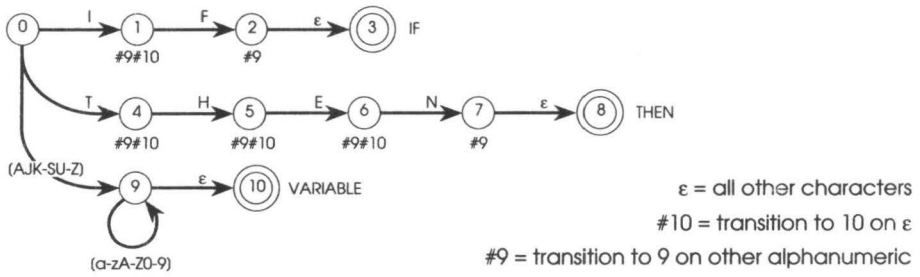


Fig. 8.3 Transition diagram for tokens starting with a capital letter. The double circles denote terminal states. The single circles denote normal states. The states marked `#N` have transitions to state `N` on the given condition. The transition marked ε is on all remaining characters.

rules, i.e. if the stream is syntactically correct. Otherwise, the parser generates an error message. The grammars of object-level scenarios, meta-level scenarios and local operations are given in Appendix 2. The parser is written with the use of *Yacc* (Yet Another Compiler Compiler) [Johnson, 1986]. The grammar rules used by Yacc consist of *terminals* and *non-terminals*. Terminals are the token names written in capital letter. Non-terminals refer to rules. An example of a grammar rule used by Yacc is:

```
antecedent  :  /* 7*/ atom
            |  /* 8*/ NOT atom
            |  /* 9*/ antecedent AND antecedent
            |  /*10*/ antecedent OR antecedent
            |  /*11*/ LPS antecedent RPS
```

which is a direct translation of the definition of an antecedent given in Chapter 6. The other syntax definitions were equally easy translatable to Yacc grammar rules.

A Yacc generated parser is a LALR(1) parser [Aho and Ullman, 1977], which stands for Look Ahead, Left-to-right scanning, and Rightmost derivation construction. It constructs the parse tree using a shift-reduce parsing technique. The parse tree is constructed in a bottom-up style. This all means that the parser reads the tokens from the token stream one by one (i.e it *shifts* the tokens) until it recognizes –with one token look-ahead– a grammar rule which it can *reduce*. When Yacc is invoked with the '-v' option, it produces a human-readable description of the parser. It consists of all the states of the parser with a description of their involved actions. The complete sets of states of the three ADDL parsers are also given in Appendix 2. For example, the descriptions of `state 6` and `state 13` are as follows:

```
state 6
        rule :  IF_antecedent THEN consequent

        NOT  shift 11
        LPS  shift 12
        ID  shift 13
        .  error

        antecedent  goto 9
        atom  goto 10

state 13
        atom :  ID_LPS termlist RPS
        atom :  ID_     (17)

        LPS  shift 20
        .  reduce 17
```

The line(s) following the state number indicate the grammar rules being processed when the state is encountered. They have no influence on the performed actions. The '_' character is an indication of which tokens has been parsed so far.

I implemented a translator that automatically converts a set of states into Smalltalk code. It turned out very convenient since the ADDL grammar changed several times during its development. The Smalltalk code produced from the above two states looks as follows:

```
state6    "instance method of ObjectScenarioParser"
"rule :  IF_antecedent THEN consequent"
   | node |
   node := self getNextNode.
   node nodeName = #NOT
      ifTrue: [↑self shift: node forState: 11].
   node nodeName = #LPS
      ifTrue: [↑self shift: node forState: 12].
   node nodeName = #ID
      ifTrue: [↑self shift: node forState: 13].
   ↑self error: 'antecedent expected'

goFrom6    "instance method of ObjectScenarioParser"
   | node |
   node := self symbolAt: parsePointer.
   node isNil
      ifTrue: [↑self error: 'antecedent expected'].
   node nodeName = #antecedent
      ifTrue: [self pushState: 9. ↑self state9].
   node nodeName = #atom
      ifTrue: [self pushState: 10. ↑self state10].
   ↑self error: 'antecedent expected'.
```

```
state13     "instance method of ObjectScenarioParser"
"atom :  ID_LPS termlist RPS"
"atom :  ID_     (17)"
   | node |
   node := self makeNewNode.
   node nodeName = #LPS
     ifTrue: [↑self shift: node forState: 20].
   self reduce: Rule17.
   ↑self goFromCurrentState
```

A brief explanation of the Smalltalk code is now given. The method `makeNewNode` reads the lookahead token from the token stream and returns a new node of the parse tree. The name of the node is equal to the name of the token. The method `shift:forState:` adds its first argument as a node to the parse tree and it pushes the second argument –being the new state– on the stack. Furthermore, it performs the method to activate the new state.

The method `reduce:` creates a node with the name of the left hand side of the rule and inserts it in the parse tree. The handled states are popped from the stack. The children of the new node are the nodes which are created while the right hand side of the rule is being parsed. The method `goFromCurrentState` sends the `goFrom#` message where `#` stands for the state on top of the stack. The method `symbolAt:` returns the node which represents the left hand side of the previously reduced rule.

The following example may illustrate the construction of a parse tree. Suppose the parser is applied to the following fragment of ADDL code which is part of an object-level scenario:

```
IF p(X) & q(a,b) THEN  . . .
```

then the (partial) parse tree shown in Fig. 8.4 is produced. Recall that the tree is built in a left-first depth-first manner. On the edges of the tree the corresponding actions are shown. By performing the shift actions, a leaf with the recognized token is added to the tree. A node is inserted to the tree, when a rule is reduced. The recognized nodes which appear at the right hand side of the rule are replaced by the new node and become its children.

The creation of the corresponding Smalltalk code is the next step which is taken by the compiler. The parse tree is traversed and meanwhile the code is generated. Note that this could have been done simultaneously with the construction of the parse tree which would have made the ADDL compiler faster. However, it would also have made it more complex and therefore less easy to modify. The code generation is the subject of the next section.

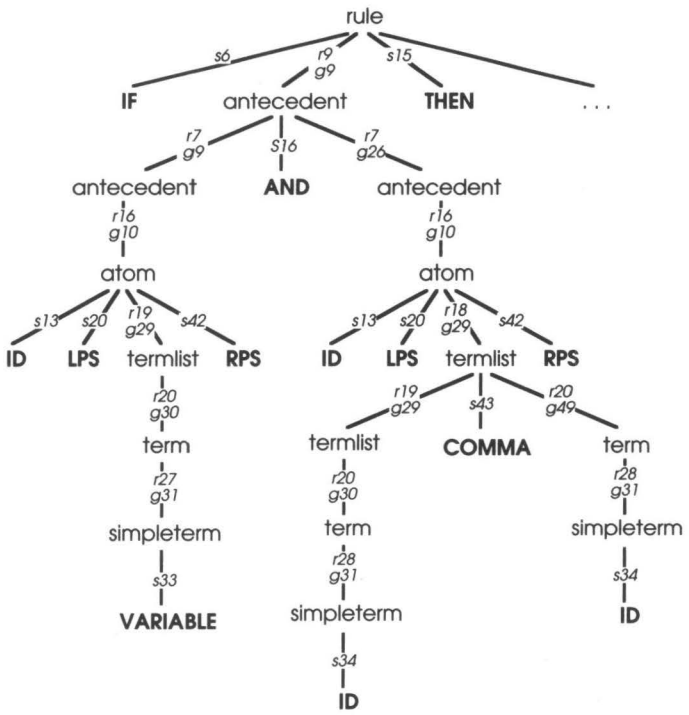


Fig. 8.4 The parse tree of a fragment of ADDL code. The italic entries refer to the actions which built the tree. An entry *s#* means a shift to state *#* on the token at the leaf. An entry *r#* means reduce rule number *#* and *g#* means go to state *#*.

8.3.3 The code generator

During the last phase of the ADDL compiler Smalltalk code is generated which is equivalent to the ADDL code. Both meta-level scenarios and object-level scenarios are compiled to instances of the class `ADDLScenario`. Operations are compiled to instance methods of the class `ADDLObject`. When a scenario is compiled, an instance of either the class `MetaScenario` or the class `ObjectScenario` is made depending on the kind of scenario. Both classes are sub-classes of `CompiledScenario` which is a sub-class of `ADDLScenario`. Their class hierarchy is as follows:

```
Object ()
   ADDLScenario ()
      CompiledScenario ('name' 'rules' 'operations' 'factBase'
            'world' 'derivations' 'ruleSelectionMethod' )
         MetaScenario ()
         ObjectScenario ()
```

A compiled scenario has a *name*, a collection of *rules* and *operations*, a *fact-base* which contains all literal facts describing the design object model, a *world* which is a sub-set of the fact-base, a set of *derivations* which contains the literal facts derived from the scenario's rules and a *rule selection method* which indicates the order in which the rules are chosen. Each rule of a scenario is translated to an instance method of `ADDLScenario`. Suppose a scenario with the name `foo` contains four rules. The selectors of the methods generated from the rules are **`addlfooNO1`** till **`addlfooNO4`**. The body of the method consists of code which represents the rule. The methods are performed by the ADDL interpreter. I say a few words about the ADDL interpreter now. It is discussed in detail in the next section.

The execution of a scenario is accomplished by a number of methods defined in the class `CompiledScenario`. There is a clear distinction between the methods which define the behaviour of the interpreter and the methods which are compiled rules. Therefore, to avoid confusion `CompiledScenario` is made a sub-class of `ADDLScenario`. The compiled rules are collected in `ADDLScenario` and all other methods are found in its sub-classes.

The code generation is done by traversing the parse tree in left-first depth-first manner. For each node with name `nodeName` in the parse tree there is a method whose selector basically looks like **`writeNodeName:on:`** having two arguments. The first argument is a node in the parse tree, the second argument is a string of code which has been produced so far. The method appends code which is specific for that node to the string. E.g. the node `rule` has following method associated with it:

```
writeRule: aNode on: aString     "CodeGenerator"
"rule  :  IF antecedent THEN consequent
       ;"
   aString addAll: '
| root aSet value |
root := (.
   aChild := aNode children at: 2.
   variables := OrderedCollection new.
   aChild isSimpleFormula
      ifTrue:
         [self writeSimpleAntecedent: aChild on: aString]
      ifFalse:
         [self writeAntecedent: aChild on: aString].
   aString addAll: ').
```

```
aSet := InstantiationSet for: '.
   self writeVariableListOn: aString.
   aString addAll: 'value := root computeTruthFor: aSet
   lookingAt: self world.
value
ifTrue: [value := '.
   aChild := aNode children at: 4.
   aChild isSimpleFormula
      ifTrue:
         [self writeSimpleConsequent: aChild on: aString]
      ifFalse:
         [self writeConsequent: aChild on: aString].
   ↑aString addAll: '].
↑value'
```

The relevance of describing in detail how the code is actually generated is questionable. Therefore, I just give the result of the code generation. The actual interpretation of the rules is of more importance and is discussed in the next section. A compiled rule of either an object-level or a meta-level scenario named `solveFoo` fits within the following frame:

```
addlsolveFooNO#     "instance method of CodeGenerator"
   | root aSet value |
   root := compiled antecedent.
   aSet := InstantiationSet for: #(array of variables).
   value := root computeTruthFor: aSet lookingAt: self world.
   value
      ifTrue: [value := compiled consequent].
   ↑value
```

The *compiled antecedent* amounts to a series of messages which recursively builds up the resolution tree in accordance with the algorithms presented in Chapter 6 and Chapter 7. When using these algorithms, the antecedent:

```
p(X) & q(a,b)
```

compiles to the following code:

```
(Node
   andNodeLeft:
      (Atom positive: 'p' arguments: #('VX'))
   andNodeRight:
      (Atom positive: 'q' arguments: #('Ia' 'Ib'))
```

The resolution tree of the above antecedent contains a single and-node and two positive leaves. To create an or-node the message `orNodeLeft:orNodeRight:` is sent. A negative leaf is created by the message `negative:arguments:`.

During the construction of the resolution tree, each newly encountered variable is registered in an *array of variables*. These variables are used for the instantiation of an instantiation set. Next, the truth value of the root is computed

and the variable bindings are added to the instantiation set. The local variable `value` becomes `true`, `false` or `unknown` in case of an object-level rule. It becomes either `true` or `false` in case of a meta-level rule.

The *compiled consequent* is constructed from the two messages:

```
then:instantSet:
then:and:instantSet:
```

The first message is used when there appears only a single atom in the consequent. The second one is used when the consequent is a conjunction of two or more atoms. Both messages infer the desired results from their arguments. A consequent consisting of a conjunction of more than two atoms results in a nesting of these messages as is illustrated by the following examples. The consequent

```
p(X)
```

compiles to

```
self
   then: (Atom name: 'p' arguments: #('VX'))
   instantSet: aSet
```

while the consequent

```
p(X) & q(Y) & r(Z)
```

compiles to

```
(self
   then: (self
      then: (Atom name: 'p' arguments: #('VX'))
      and: (Atom name: 'q' arguments: #('VY'))
      instantSet: aSet)
   and: (Atom name: 'r' arguments: #('VZ'))
   instantSet: aSet)
```

The compiled rules, which are discussed so far, occur in an object-level scenario. Compiled meta-level rules are very similar to compiled object-level rules. The major difference lies in the way the way the rules are interpreted, which is discussed in the next section. As a matter of fact, compiled meta-level rules fit within the same frame as compiled object-level rules. Differences are i) that the resolution tree is built up by instances of the class `MetaNode`, which is a sub-class of `Node` and ii) that the leaves of the tree are constructed from meta-atoms instead of atoms.

Suppose the following rule is the first rule of a meta-level scenario called **`solveMeta`**:

```
IF   positive(p(X)) & unknown(q(X))
THEN goal(foo)
```

which compiles to the following Smalltalk method (the code has been pretty-printed to improve readability):

```
addIsolveMetaNO1    "instance method of ADDLScenario"
   | root aSet value |
   root := (MetaNode
      orNodeLeft: (MetaAtom
         name: #positive
         argument: (Atom positive: #p arguments: #(#VX)))
      orNodeRight: (MetaAtom
         name: #unknown
         argument: (Atom positive: #q arguments: #(#VX)))).
   aSet := InstantiationSet for: #(#X).
   value := root computeTruthFor: aSet lookingAt: self world.
   value
      ifTrue:
         [value := self
            then: (MetaAtom
               name: #goal
               argument: (Atom constant: #foo))
            instantSet: aSet].
   ↑value
```

The methods which generate the compiled rules are distributed over two classes. One being a sub-class of the other. The protocol `smalltalk writing` of the class `ObjectScenarioCodeGenerator` contains the methods which produce compiled object-level rules. These methods are:

```
writeAntecedent:on:
writeAtom:on:
writeConsequent:on:
writeElements:on:
writeFunction:on:
writeNegativeAtom:on:
writePredconstant:on:
writeRule:on:
writeRules:on:
writeScenario:collectIn:
writeSimpleAntecedent:on:
writeSimpleConsequent:on:
writeSimpleterm:on:
writeTerm:on:
writeTermlist:on:
```

while the protocol `smalltalk writing` of its sub-class `MetaScenarioCodeGenerator` contains the methods which generate compiled meta-level rules. The only methods of this protocol are

```
writeAntecedent:on:
writeConsequent:on:
```

```
writeConstant:on:
writeMetaAtom:on:
writeSimpleAntecedent:on:
writeSimpleConsequent:on:
```

The remaining methods are inherited from `ObjectScenarioCodeGenerator` being its super-class.

The code generation of local operations is straightforward. Each operation compiles to a Smalltalk method. The compiled method is a literal translation of the ADDL code. It is stored as a method in the class `ADDLObject`.

8.4 The ADDL interpreter

The execution of an ADDL program is invoked by a design goal which is stated by a designer. The ADDL interpreter activates a scenario which tries to solve the goal. Ultimately, the goal is satisfied by a sequence of active scenarios. The application of the design knowledge represented by the rules of these scenarios, result in a design object description which satisfies the initial goal. The execution of both object-level and meta-level scenarios is done by the interpreter which is written in Smalltalk-80. A (compiled) scenario is activated by sending it the message `executeYourSelf`. It selects one of the rules of the active scenario and evaluates it. The inferred results are administered and unless its goal has been satisfied, the next rule is selected. The rule selection is done by using a rule selection method.

The interpreter is equipped with a belief revision system. Whenever, an inconsistency is detected with regard to the design object description, the interpreter tries to recover. A false assumption and all its dependencies are removed from the system and the interpreter continues its process. The multi-world mechanism allows for the activation of two or more scenarios concurrently. For this mechanism only the frame-work has been implemented. It is not yet fully operational due to conflicts of priorities. It is considered a topic for further research. All the above aspects of the ADDL interpreter are discussed in the following sections.

8.4.1 Scenario execution

An outline of the execution of a scenario is depicted in Fig. 8.5. When a scenario is activated, the interpreter creates a *world* which represents the scenario's view on the fact-base. A world is a sub-set of (or identical to) the fact-base. It is used to infer the antecedents of the scenario's rules from. Therefore, a world narrows down the view of a scenario looking at only a portion of the entire fact-base. It is created by the built-in predicate symbol `goal` (see Chapter 7). Consequently, it may be impossible to infer an antecedent with respect to a certain world, while it would have been possible to infer it with respect to the entire fact-base.

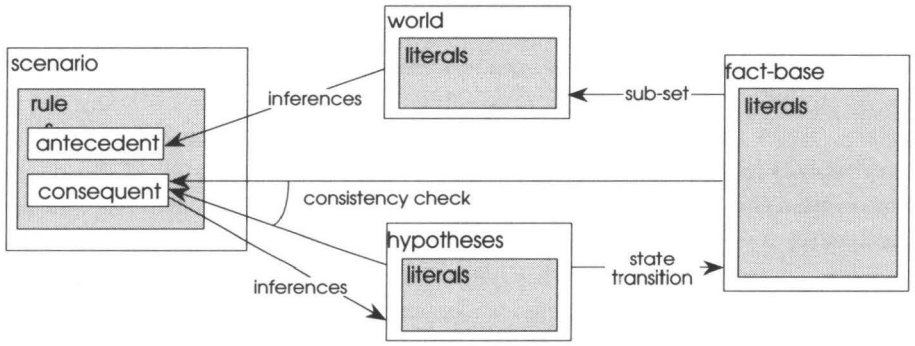


Fig. 8.5 Schematic overview of the execution of a scenario. The world is created upon activation of the scenario. The state transition takes place upon exiting the scenario. An arrow indicates a flow of information.

The purpose of the world mechanism is to optimize and to control the object-level inference. As an example of this mechanism, suppose a designer is designing a table with four legs. The knowledge to attach a leg to the table top is represented by a single object-level scenario. A meta-level scenario can then activate this scenario four times. Each time, the scenario's world is focussed on a different part of the table represented by a different sub-set of the fact-base.

The literal facts which are derived from both the rules and the world, are stored in a temporary place called the set of hypotheses. Each derived literal fact is checked upon consistency with both the fact-base and the set of hypotheses. Note that a consistency check with the world instead of the fact-base is insufficient, since a literal fact may be consistent with a world but inconsistent with the fact-base. The price paid for the consistency check is not such a burden, because the derivation procedures for the antecedent are much more time consuming than those for the consequent (see Chapter 6). Finally, the set of hypotheses is merged with the fact-base when the execution of the scenario terminates. By means of such a state transition a new extended fact-base is created.

The execution of a scenario is triggered by sending it the message `executeYourSelf`. Its state becomes `active` and it starts processing its rules as long as its state remains active. The rules are processed by selecting one of the scenario's rules. The selection mechanism is discussed in the next section. The selected rule is evaluated with respect to the scenario's world. The conclusions drawn from the rule are registered in the set of hypotheses. The rules are evaluated by applying the compiled methods as described §8.2.3. The methods which are used i) to construct a resolution tree and ii) to traverse it in either a forward or a backward direction, are a direct translation of the computational mechanisms

presented in §6.5.2, §6.5.3 and §7.4.2. Here, the rule interpretation is only presented at a global level in order to prevent redundant repetition.

A rule returns a *value*, which can be `normal`, `success`, or `failure`. If the value is `normal`, the state of the scenario remains `active` and the next rule is selected. If the value is `success` or the value is `failure` the state of the scenario becomes `inactive` and the execution terminates. In the first case, the goal, that the scenario aimed at, has been satisfied and control is given back to the parent scenario after the set hypotheses has been merged with the fact-base. In the second case, all literal facts derived from the scenario are canceled and the satisfaction of the goal has failed. Control is given to the user-interface, which provides an error message and tries to recover in dialogue with the designer[10]. The following instance method of the class `CompiledScenario` embodies the execution of either an object-level or meta-level scenario:

```
executeYourself    "instance method of CompiledScenario"
   | state rule |
   state := #active.
   [state = #active]
      whileTrue: [
         rule := rules perform: ruleSelectionMethod.
         state := self apply: rule].
   state = #failure
      ifTrue: [↑ADDL scenarioFailed: self].
   ↑self stateTransition.
```

Suppose `solve-a` is a compiled scenario, which is to be executed. The world of `solve-a` has been created by the interpreter, before the above message is sent to it. The body of the method is self explanatory. When the message `stateTransition` is sent to `solve-a`, the set of hypotheses is merged with the fact-base and the method returns `#success`. The parent scenario which caused the activation of `solve-a` becomes the active scenario.

8.4.2 Rule selection and application

This section describes the inner loop of the execution of a scenario. A rule is chosen using some selection method and it is applied returning a state. Unlike Prolog which has a single static search strategy for selecting clauses [Clocksin and Mellish, 1981], ADDL provides a mechanism to allow for multiple rule selection strategies [Veerkamp, Pieters Kwiers, and ten Hagen, 1991]. Furthermore, it allows for dynamically changing the current rule selection strategy. There are several ways to control the selection of rules. The most straightforward mechanism is to

[10] The interpreter is not yet capable of doing "error recovery" in case of an inconsistency. The fact is simply reported to the designer, who can modify the knowledge-base in order to resolve the inconsistency.

search the collection of rules in a top-to-bottom manner until an applicable rule has been found. This process is repeated either until no more rules can be applied or until the scenario's goal has been satisfied. There are many possible variations on this mechanism, such as searching the rule in a circular fashion, selecting each rule only once, or selecting only a single rule, etc. More advanced rule selection mechanisms base the search on the contents of the rules, such as selecting the rule with the most complex antecedent, with the least number of variables, or with the most complex consequent, etc. In this section, I present the framework which allows for multiple rule selection strategies, and I give some of the implemented strategies. Furthermore, I show how the system programmer can add a new rule selection method. Finally, I show how the rule selection mechanism can be modified run-time.

The rules of a scenario are collected in the instance variable named `rules` of the class `CompiledScenario`. It is an instance of the class `Rules` which is a sub-class of `OrderedCollection`. The collection consists of the compiled rules of a scenario and is has an instance variable `index` which points to the the last-chosen rule. Initially the index is zero. Each compiled scenario has an instance variable `ruleSelectionMethod` which contains the message selector of a rule selection method. This message is sent to the collection of rules of the scenario and returns the next rule. It is embodied by the following line of code of **`executeYourself`**

```
rule := rules perform: ruleSelectionMethod
```

which performs the method which is indicated by the current rule selection method. A rule selection method is set in the scenario header of a scenario. Suppose the following fragment of ADDL code is the header of a scenario with the name `foo`:

```
foo( aRuleSelectionMethod )
IF  . . .
```

The key-word `aRuleSelectionMethod` refers to the rule selection method of `foo`. The scenario compiler checks whether it is a valid (i.e. an implemented) rule selection method and it instantiates the instance variable `ruleSelectionMethod` to the appropriate method selector. It gives an error message if it is not a valid selector. When a rule selection method is omitted from the scenario header, the instance variable is set to the default rule selection method. The default rule selection method chooses each rule once in a top-to-bottom fashion. It is outlined as follows:

```
defaultRuleSelectionMethod
   index := index + 1.
   index > self size
      ifTrue: [↑nil].      "there are no more rules"
   ↑self at: index
```

Another rule selection method, which also operates in a top-to-bottom manner, applies each rule repeatedly until it fails. This method is similar to the clause selection strategy of Prolog. The name of the rule selection method is `eachUntilFail`. It is implemented as follows:

```
eachUntilFail
   (index = 0 or: [(self at: index) failed])
      ifTrue: [index := index + 1].      "take next rule"
   index > self size
      ifTrue: [↑nil].      "there are no more rules"
   ↑self at: index
```

A third rule selection method which has been implemented, behaves as an *exclusive or* over the rules. It applies each rule from top to bottom, until it finds a rule which succeeds. The scenario terminates after application of that rule. It consists of the following code:

```
exclusiveOr
   ((index > 0 and: [(self at: index) succeeded])
               or: [index > self size])
      ifTrue: [↑nil].      "only a single rule is applied"
   index := index + 1.
   ↑self at: index
```

As shown by those three examples, it is relatively easy to add a new rule selection method to the existing methods. It amounts (i) to adding a new method selector to the list of implemented rule selection methods, and (ii) to writing the proper Smalltalk code belonging to the new method. The easiness of writing new code depends of course on the complexity of the selection strategy one wants to use.

The current rule selection method of an active scenario can be replaced by another method. This is done by the object-level built-in predicate symbol `directive` as being introduced in §6.5.3. When the expression `directive(aRuleSelecticnMethod)` is encountered in a rule of the active scenario and `aRuleSelectionMethod` is a valid selector of a rule selection method, then from that moment on the rules are selected according to this method. Changing the rule selection method of a scenario only affects the rule selection of the current life-cycle of the scenario. When the scenario terminates and it is activated another time the rule selection method will just be the original one that the scenario has been compiled with.

The actual application of rules takes place by performing the methods of compiled rules as described in §8.2.3. It is triggered by the message:

```
self apply: rule
```

which applies the rule and administrates its results. Each rule has an instance variable `state` which contains the *return state* of the rule after it has been applied. It can be `true`, `false`, `unknown`, `success`, or `failure`. Each of them is explained below.

`true`: A rule returns `true` if its antecedent can successfully be derived from the scenario's world and the conclusions are consistent with the fact-base. The goal of the scenario has not (yet) been reached.

`false`: A rule returns `false` if its antecedent can *not* be derived from the scenario's world. The rule is not applicable and hence it fails.

`unknown`: A rule returns `unknown` if it is undecidable whether it is possible to derive its antecedent from the scenario's world. The rule may succeed a next time, when the object information state is more complete.

`success`: A rule returns `success` if its antecedent can be derived and the scenario's goal can be concluded from the rule's consequent.

`failure`: A rule returns `failure` if, though the antecedent can be derived, the conclusions of the consequent are inconsistent with the fact-base. In fact, this means that the knowledge-base is in contradiction with the object information state. This inconsistency needs to be removed in order to proceed the design.

The message `apply:` returns `#active` in the first three cases. It returns `#success` in the fourth case and `#failure` in the last case. In the last two cases, the execution of the scenario halts.

After successful application of a rule the variables bindings which are registered in an instantiation set are stored in an instance variable of the compiled scenario, called `ruleBindings`. It is a dictionary with the selectors of the compiled rules as keys. Their values contain information about the application of rules. The information contains the variable bindings of each successful application of a rule during the life-time of a scenario. It prevents a rule from being applied more than once with the same variable bindings. Furthermore it is used to record the behaviour of scenarios. The `ruleBindings` are reset each time a scenario is activated.

8.4.3 Belief revision

In the previous section, it has been stated that a rule returns a failure if the fact-base is inconsistent with a derived conclusion. Such an inconsistency occurs if a *positive* fact is being asserted to the fact-base while it already exists as a *negative* fact, or vice versa. Another, less fatal, inconsistency can occur when an object's attribute receives a value. If I recall the built-in predicate symbol `value`, the following object-level atom is an example of an assignment of a value `8` onto an

attribute `x` of an object `a`

```
value(a:x,8)
```

The above is a valid expression if the attribute `x` was `nil` prior to the assignment. However, there is an inconsistency if the attribute has already been set at an earlier state of the design process and that value is different from the new one. Such an inconsistency is allowed because I consider the old value an *assumption* which has been replaced by the new value. Such a mechanism is called *belief revision*, since an attribute value is believed to be true until it has been determined that another value suits better.

The belief revision mechanism is not simply a matter of replacing an old value by a new one. Some assumptions may be 'based on' a value that has been or will be revised. In case of a revision they can be based on a non-existing assumption. Below I explain the meaning of 'based on' in this context. Such assumptions *depend* on the revised assumption. All dependents of a revised assumption need to be removed when a revision takes place. Those dependents may on their turn have assumptions that depend on them, and so on. Therefore, the revision mechanism may result in a chain of assumptions which are removed. Technically, removal of an assumption amounts to setting the attribute value of a dependent to `nil`.

To aid this mechanism, a dependency graph of assumptions is maintained during the course of the design process. Each time an assumption is made, it is registered on which assumptions it is based. Suppose the following expression is encountered

```
value(a:x,t)
```

where `a` is the name of a composite object, `x` is one of its attributes and `t` is a term from an object-level language. Then the value of `t` may be obtained by using some attribute values which are assumptions. The value of the attribute `x` depends on these assumptions. The dependencies are determined while the antecedent of a rule is derived. The mechanism is illustrated by the following example of a local operation and a rule:

```
:plus[L] = { self:a + L:b }

IF   p(X,Y,Z) & equal(N,X:plus[Y])
THEN value(Z:c,N+1)
```

Suppose the variables `X`, `Y`, and `Z` are unified to the composite objects `p1`, `p2`, and `p3`, then the attribute value `z` of `p3` depends on the attribute values `a` of `p1` and `b` of `p2`. If either of the two attributes `a` or `b` is revised, then the value of `c` is set to `nil`.

Belief revision as described above takes place entirely at the object-level. It is concerned with the revision of attribute values of objects. At the meta-level, belief

revision deals with the retraction of literal facts. During the design process, a user may remove a literal fact from the fact-base because he is not satisfied with the current state of the design. Such an action causes a total revision of the fact-base since each literal fact which has been asserted after the removed literal fact must be removed as well. For example, the following meta-level expression removes the fact `p(a)` and all its successors from the fact-base[11].

```
retract(p(a))
```

To enable this mechanism each asserted literal fact has a *time-stamp* attached to it. A time-stamp is a natural number which increases as the design proceeds. When a retraction occurs, all literal facts which have a time-stamp greater than the time-stamp of the retracted literal fact are removed from the fact-base. After a retraction the time is reset to the moment of assertion of the retracted literal fact. The design proceeds from the next rule after the one which asserted the retracted literal fact.

8.5 The programming environment

Since ADDL has been developed in Smalltalk, its programming environment is implemented in the same language. It consists of i) a *scenario browser*, which aids the ADDL programmer in writing scenarios, ii) a *prototype browser* in which the prototype definitions are given, and iii) an experimental ICAD system, in which the scenarios are executed. The following three sections present the three tools briefly.

8.5.1 The scenario browser

The *scenario* browser is very similar to the Smalltalk system browser. It allows for the ADDL programmer to browse through *categories* of scenarios. Scenarios can be created, edited and removed. The scenario browser consists of four views, as shown in Fig. 8.6. Each scenario is organized in a category for convenience of the programmer. The top-left view contains a list of scenario categories. The programmer can select a category by clicking the mouse on one of the items. The name of a category is highlighted when it is selected. By selecting a category, the names of the scenarios which belong to that particular category are shown in the bottom-left view. These are either meta-level scenarios or object-level scenarios depending on the state of the switch in the middle-left view. In Fig. 8.6 the switch *object* is turned on.

In the right view, the code which represents a scenario is viewed. It consists of two sub-views. The upper view shows the rules of a scenario, while the bottom view give its (optional) local operations. Fig. 8.6 shows the code which belongs to

[11] The meta-predicate symbol `retract` has not been introduced in Chapter 7, since it is purely used for control and it does not add to the process information state.

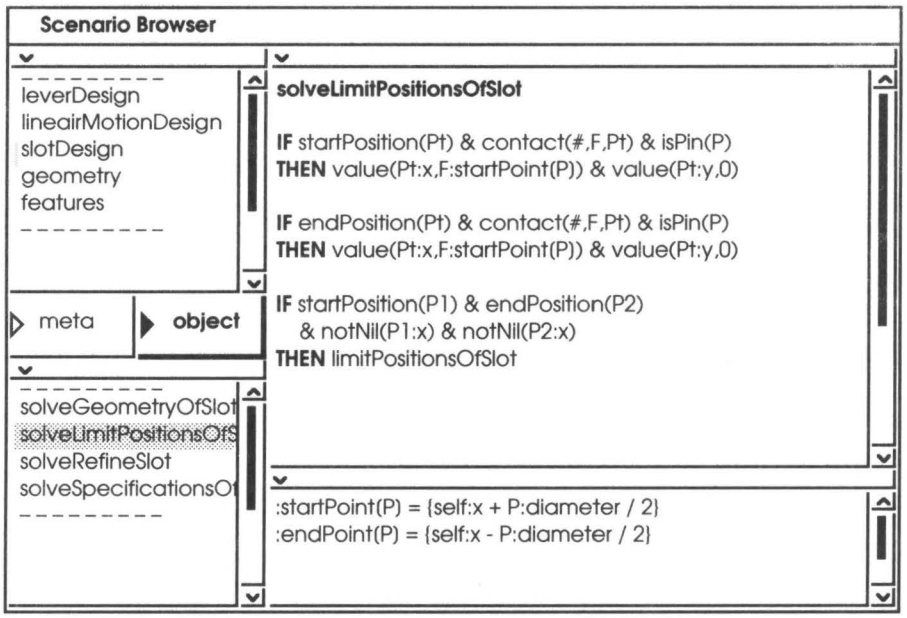


Fig. 8.6 Scenario Browser that shows an object-level scenario of the category **slotDesign**.

the object-level scenario `solveLimitPositionsOfSlot` of the category `slotDesign`. In each of the views (except for the switch view), different pop-up menus are active offering commands which are appropriate to its contents.

The commands for the category view are listed in the order in which they appear in the menu. In this view (and others), the menu has fewer options when no category has been selected. The commands marked with * are only active when a category has been selected. Each command is presented below.

`file out`* All scenarios which belong to the selected category are stored in a file whose name is prompted. The file can be read-in by another ADDL environment or can be used as a back-up.

`print out`* All scenarios which belong to the selected category are pretty-printed in a file whose name is prompted.

`add category` A category name is prompted. The new category name is added immediately above the currently selected category (if one is selected) or at the bottom of the list.

`rename`* A new name is prompted. The current selection is replaced by the new name both in the list and in all scenarios under the selected category.

`remove*` The selected category name is removed from the list. All scenarios under the selected category are removed from the system. For safety reasons, it is first asked whether the scenarios should really be removed.

`update` The category listing is brought up to date. It may be necessary after filing in a new category or adding one in another scenario browser.

The commands for the view which shows a list of scenario names (in short *scenario view*) are exactly the marked commands of the category view. They are only active if a scenario name is selected. Obviously, the commands operate on the selected scenario names instead of the selected category.

The *code* view allows a programmer to edit scenarios. It shows the rules and local operations of a selected scenario, if a scenario is selected. Otherwise, it shows a template scenario. The commands offered by the pop-up menu are the default commands of a Smalltalk code view. They are `again`, `undo`, `copy`, `cut`, `paste`, `accept` and `cancel`. The first five commands aid the programmer in editing the code. There meanings are obvious. The command `accept` invokes the ADDL compiler. If the code is correct, it is stored and the code view presents a pretty-printed version of the code to the programmer. Otherwise, it prompts an appropriate error message to the programmer. The command `cancel` removes all changes introduced to the code and restores the original contents of the code view.

8.5.2 The prototype browser

The aim of the prototype browser is to define and edit the prototypes of composite objects. The outlook is comparable to the scenario browser. It consists of a *category* view, a *prototype* view, a *definition* view, a *operation* view, a *code* view, an *attribute* view, and a *value* view. Fig. 8.7 depicts an example of a prototype browser. The operation `distance:` of the prototype `point` of the category `geometry` is highlighted.

The commands of the category view are the same as those of the category view of the scenario browser. Likewise, the commands of the prototype view are the marked commands of the category view. The definition view enables a programmer to create and modify prototype definitions. The possible modifications are changing the name of the parent and adding, deleting and modifying the attribute names. The operations of a prototype definition can be accessed through a separate interface, viz. the operation view and the code view.

The attribute view and value view show the attribute names of a prototype definition and their respective (default) values. The attribute view has only a single command and the value view has no commands. The attribute view allows a designer to modify the value of an attribute. This value is then used as a default value upon instantiation of an object of the involved type. The operation view lists

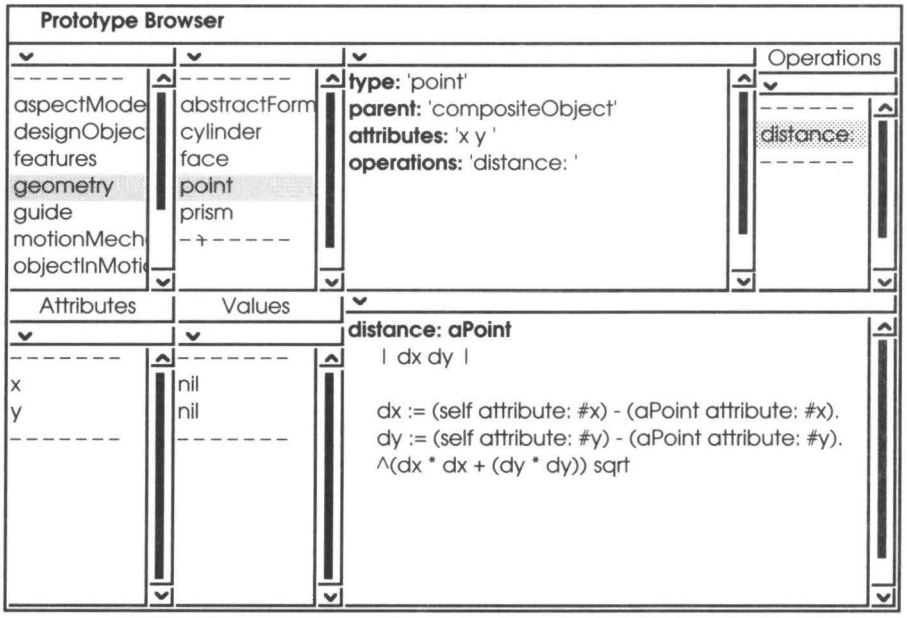


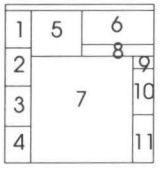
Fig. 8.7 Prototype Browser focussed on the prototype **point** of the category **geometry**.

the operations that are defined for a prototype. The possible commands are the same as for the prototype view. The source code of the operation can be edited in the code view, which gives the programmer the same commands as the code view of the scenario browser.

Currently, the source code of operations is Smalltalk code. An accepted operation is implemented as an instance method of the class `ADDLObject`. The interpreter checks whether an operation is valid to a certain object before it applies the standard Smalltalk message passing mechanism. The class `ADDLObject` also defines the instance method `attribute:` which allows the programmer to access the attribute values of a composite object. A future implementation will have a separate operation compiler which allows for operations written in ADDL code similar to that of operations local to a scenario. For example, the expression "`aPoint attribute: #x`" will then look like '`aPoint:x`'. So far, the straightforward implementation of the operations in Smalltalk gives me more programming liberties.

8.5.3 The experimental ICAD system

The current ICAD system is merely constructed to test the ADDL interpreter than to act as a designer's tool. It is used to run scenarios and to show intermediate and final results of the execution. A future version must have a user-interface that better suits the designer's requirements. Then, the designer can play an active role in the design process choosing scenarios dynamically. Now, the only role being played is that of a spectator watching the system doing the design. Fig. 8.8 gives an overview of the structure of the experimental system. It consists of eleven views which are schematically depicted.



1. category view
2. scenario view
3. objects view
4. object view
5. facts view
6. process view
7. wireframe view
8. dialogue view
9. button view
10+11. gauge views

Fig. 8.8 Overview of the experimental system.

Views #1 and #2 are means to select a scenario name and state it as a goal. Other views show the current state of the design object description. There is a view on the object-base and on an individual object. There is a view on the fact-base or on a portion of it. Furthermore, there is a view on the process parameters. Finally, a wireframe view gives a geometric representation of the design object. These are the view #3 til #8 in the picture. The remaining views are interfaces to the user of the system. One view prompts the user for answer upon queries asked by built-in predicate symbols. Another view is a button that must be pushed if the user wants to continue. The last two views are gauges which allow the user rotating the wireframe model (views #9 till #11).

The category view of Fig. 8.9 highlights the category `leverDesign` of which the scenario `solveLevel0` is selected in the scenario view. This view offers three commands, viz. `goal`, `update` and `reset`. The first command starts a design session. It asserts a top-level goal to the process information state, which is then going to be solved by the system. The update command assures that the list of scenarios is up-to-date. This may be necessary when the user is simultaneously editing and executing scenarios. The reset commands sets all view that focus on the design object model to an empty state.

The *objects view* presents a list of names of all instantiated objects at a particular moment of the design process. It represents the contents of an object-base. The user may select an object name, as a result of which the object's internal state is presented in the *object view*. The object view shows the type of an object,

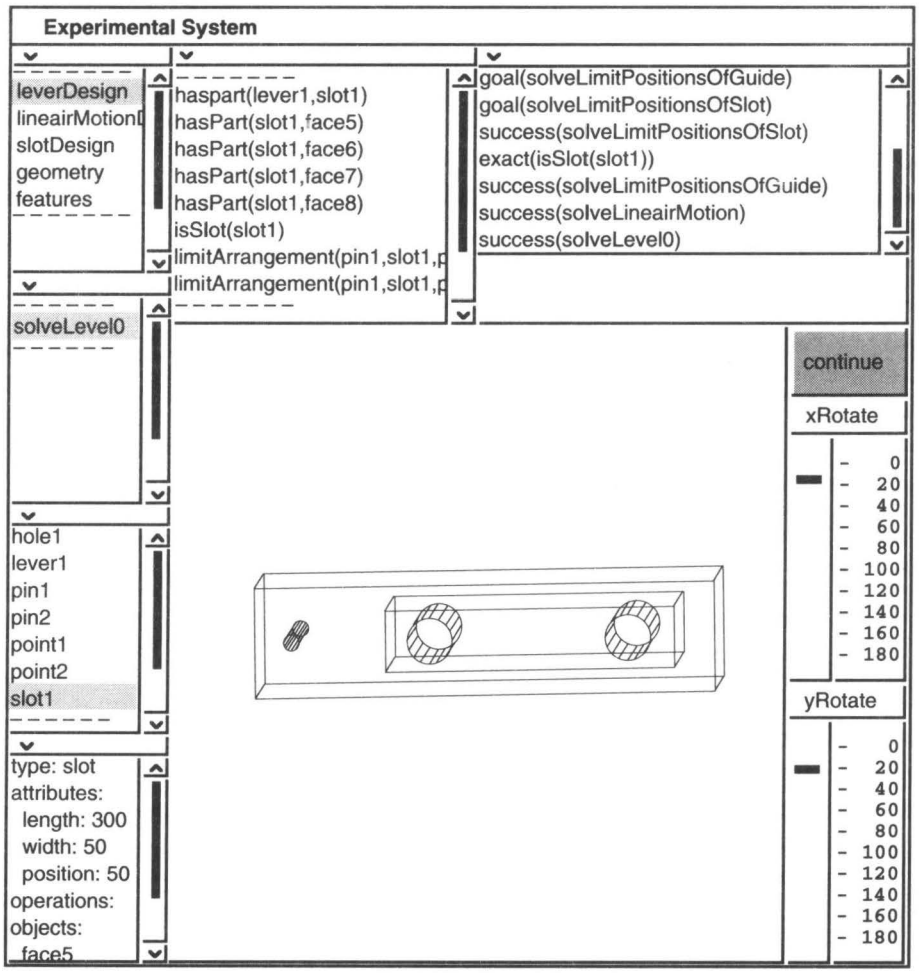


Fig. 8.9 Experimental system focussed on the prototype **point** of the category **geometry**.

its attributes and their values, its operation names, and a set of object names being part of the selected object. The objects view has a single command `inspect` which opens a view on the selected object. It enables the user to take a closer look at the object.

The view on the fact-base has a conditional and an unconditional state. If no object name has been selected in the objects view, the fact-base view shows the entire set of literal facts appearing in the fact-base. If an object name *is* selected, then it shows only those literal facts which have the selected object name in their

argument list. Therefore, it can be regarded as a world which is focussed on a particular object. The commands of the fact-base view are `inspect` and `sort`. The first is equivalent to the inspect command of the objects view. The second command sorts the literal facts of the viewed set alphabetically. The *process* view shows the asserted process parameters in chronological order. This view is the only view which is synchronously updated as the design proceeds. Therefore, as soon as a new process parameter is asserted to the process information state, it is shown to the user. The other views only update their contents, when an update signal has been given by the system.

The *wireframe view* provides a geometric representation of the design object description by means of a wireframe model. There is a prototype called `geometricModel` which has a single attribute `model` and which responds to six operations, viz. `close`, `create`, `dimension:`, `left:`, `top:`, and `update:`. The first five operations deal with opening and closing the view on the model. When an update message is sent to the object, it displays all objects which are part of it as defined by the `hasPart` predicate symbol. It only displays those objects for which a display operation has been defined. The user can rotate the wireframe model along the x-axis and the y-axis using the *gauge views*. An update message halts the design process. By pushing the *continue button* the user can proceed the design process.

The *dialogue view* enables the system to interact with the user in a plain manner. Each query posed by a user-interface predicate symbols opens a distinct dialogue box in the dialogue view. The box vanishes after the user has given an answer.

8.6 Discussion

This chapter presented the implementational aspects of ADDL. If I compare the amount of time spent on the three major activities, i.e., the compiler, the interpreter, and the experimental IIICAD system, the implementation of the interpreter was by far the most time consuming. During the development of ADDL, its specification changed due to renewed insights obtained by experience and by discussions with other researchers. Especially, during my stay in Japan and afterwards because of my discussions with Jan Treur, the implementation went through a number of major changes. Some of them caused minor adjustments; others caused major revisions. This section does not discuss the specified but unimplemented features of ADDL. The reader can find this discussion if § 10.4.

For instance, the decision to omit a disjunctive conclusion caused only minor changes to the compiler and the interpreter. However, the move to a meta-level architecture required the implementation of an entire new compiler and interpreter. These two changes are obvious to the user, since they lead to a modified syntax. The switch from unification by instantiation pair lists to

unification by backtracking, is less obvious to the ignorant user, although it has made the unification sound and it increased the efficiency by a factor two.

The use of Smalltalk as implementation language has been a great help for introducing the improvements to ADDL. Its flexible programming interface made it possible to alter the code dynamically. Its ability to reuse code greatly aided in adding functionality, such as meta-level reasoning, to the system. People who are mainly concerned about the performance of applications, criticize Smalltalk for being slow. I think that the time that you win during the development of an application, greatly outweighs a minor loss of run time performance. This is certainly the case for projects such as the IIICAD project, where the system specifications are highly contingent and are due to many revisions. In the event that the system is fully crystallized, it can easily be moved to a language like C++ [Stroustrup, 1986] in order to improve performance. However, it is argued that as soon as C++ obtains the same functionality as Smalltalk (in terms of class hierarchy) it will show performance comparable to Smalltalk. It may be the price you pay for the flexibility of object-oriented programming.

9

An Example Design System in ADDL

9.1 Introduction

This chapter gives an application of ADDL by showing the implementation of an example design system. Parts of the material presented here have also been used in [Xue *et al.*, 1990]. The class of design problems tackled by the system involves the design of a linear motion mechanism introduced in Chapter 3. I show that it is feasible in ADDL to build a meta-model, which represents the solution for a certain category of design problems in a general way. In a recent paper [Veerkamp *et al.*, 1990], I introduced the meta-model mechanism as a representation of the qualitative behaviour of a design object. The meta-model mechanism plays a dual role in the design system.

First of all, to create an *aspect model* (see Chapter 2), a designer must know physical laws relevant to the aspects being modeling. Since modeling involves the creation of representations of the design object in terms of specific physical phenomena, these representations are called aspect models, the system must know about the behaviour relevant to the aspect. Different aspect models derived from the same design object description are not independent. In order to make consistent models of the design object, relationships among aspect models must be known. Knowledge about physics in the meta-model is indispensable to maintain the consistency among aspect models.

Secondly, knowledge about available structural components and physical phenomena is necessary to perform conceptual design. At this stage of the design process, the functional specification is mapped onto an abstract anatomical structure. Such a mapping is achieved by means of a behavioural model. The system breaks down the specifications into behaviours of the design object, and

determines structures which embody the behaviour. Knowledge about structural components and physical phenomena is used by the system to accomplish the mapping. The result of the mapping is a meta-model which represents the qualitative behaviour of the design object.

The meta-model in the example design system describes the qualitative behaviour of a linear motion mechanism. In order to include new designs, the addition of specific solution dependent scenarios is the only thing a knowledge engineer has to do. The names of solution dependent scenarios in the system start with `solveSlot`, or `solveShaft`, e.g. `solveSlotLimits`. The meta-model 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 meta-model description.

At least three different approaches can lead to the design of a linear motion mechanism. A first approach uses a slot and a pin, another uses a shaft and a slider, and a third approach uses a rail and a table to construct a linear motion mechanism (see Fig. 9.1). All three approaches employ the same meta-model description in ADDL. The aspect models which are created on the meta-model differ, e.g. each type of solution has its own geometric and kinematic models.

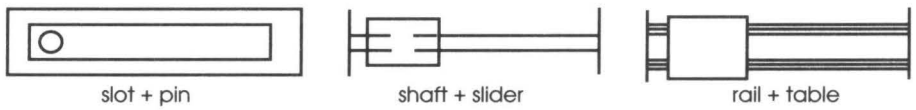


Fig. 9.1 Three possible approaches to construct a linear motion mechanism.

To aid the design of a linear motion mechanism, a number of scenarios are specified and implemented in ADDL. There are two categories of scenarios, viz. *meta-level* and *object-level* scenarios. The former have the knowledge about *how* to design, they direct the design process and describe what kind of actions must be performed concerning the current state of the design object representation. The latter have the knowledge *what* to design, they model the design object and add new information to the design object representation obtaining a more precise description [Takeda *et al.*, 1990]. Meta-level scenarios evaluate the process information state and assert *design goals* and other process parameters. Either (other) meta-level scenarios or object-level scenarios can be activated to satisfy these goals. The meta-level and object-level interpreters take care of this mechanism (see Chapter 7 for a detailed discussion on this subject). Each stage of the design process (i.e., conceptual, fundamental, and detailed) has its own subset of both categories of scenarios associated with it.

This chapter is subdivided into four parts. In §9.2, I present the example design system. The scenarios which control the the conceptual, fundamental and detailed stages of the design process are given in §9.3 till §9.5. Finally, §9.6 concludes this chapter.

9.2 A linear motion design system

To solve the linear motion design problem a simple design system is implemented in ADDL. It is not intended to be applied to an actual design problem. The system merely shows a very small part of a large intelligent CAD system. In this respect I want to stress that in a full system the number of rules in a scenario will be greater. The overall idea will nevertheless be the same. The system shows how the design process is directed by means of meta-level scenarios, and it shows how a design object representation is developed by using object-level scenarios.

The system's runtime environment consists of i) meta-level scenarios that control the design process, ii) object-level scenarios that manipulate the design object description, iii) a process information state consisting of design process information, and iv) an object information state consisting of a fact-base containing literal facts that describe the design object's structure and an object-base with attributes that describe the design object's data. The first two represent the static knowledge of the system, while the last two represent the dynamic information (see Fig. 9.2). The process information state and the object information state grow as the design proceeds. Meta-level scenarios augment the process information state by asserting goals, declaring the success of goals, and adding process parameters. Object-level scenarios augment the object information state by asserting atomic statements to the fact-base, by adding objects to the object-base, and by assigning values to objects' attributes [Takeda *et al.*, 1990].

	state description	knowledge
meta	process information state	meta-level scenarios
object	object information state	object-level scenarios

Fig. 9.2 Static and dynamic aspects of ADDL

9.2.1 The knowledge base

The system's knowledge base consists of five meta-level scenarios and eight object-level scenarios. Actually the number of object-level scenarios is greater, but I omit the scenarios which give a design solution different from the slot and pin solution. The top-level goal of the system is `linearMotion`, which is solved by a

meta-level scenario called `solveLinearMotion`. This scenario contains four rules of which the first three assert sub-goals. Each of these sub-goals are solved by either meta-level or object-level scenarios. A meta-level scenarios may on its turn generate new sub-goals, and so on. Fig. 9.3 shows the goal structure as specified by the meta-level scenarios. It is equivalent to the goal structure presented in Chapter 3 though the goal names here are more concise for reasons of implementation.

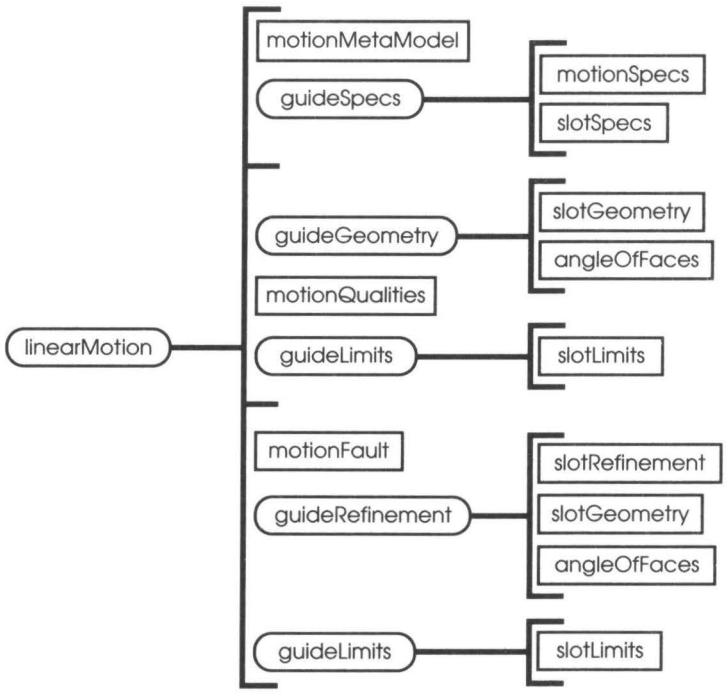


Fig. 9.3 Goal structure as specified by the meta-level scenarios.

A set of asserted goals describes a process information state. The success of a goal or sub-goal is also registered in that process information state. Thus, the example design system described in this chapter aims at satisfying the goal:

```
goal(linearMotion).
```

The meta-level interpreter always tries to solve the most recently asserted goal or conjunction of goals first (depth first strategy).

The system employs the type hierarchy shown in Fig. 9.4. There is a meta-level and an object-level type hierarchy. The meta-level hierarchy consists of the primed types of the object-level hierarchy plus the type `objectAtom`. In the

sequel, I omit for convenience the priming of meta-level types unless when it causes confusion. I use the following short-hand notations for the types: `object:O`, `composite:C`, `objectAtom:A`, `primitive:P`, `pivot:PV`, `guide:GU`, `point:PT`, `face:FA`, `objectInMotion:OM`, `lever:LE`, `sliderDevice:SD`, `motion:MN`, `slot:SL`, `shaft:SH`, `pin:PI`, `slider:SR`, `symbol:SY`, `type:TY`, `string:ST`, `number:NU`, and `array:AR`. E.g. the type `guide` has the super-types `composite` and `object`, and it has the sub-types `slot`, and `shaft`. Each constant belongs to one of these types and each function evaluates to a type. All predicate symbols are defined over these types. For each scenario in the system, Appendix 3 gives a signature containing the types, constants, functions and predicate symbols being used in the scenario.

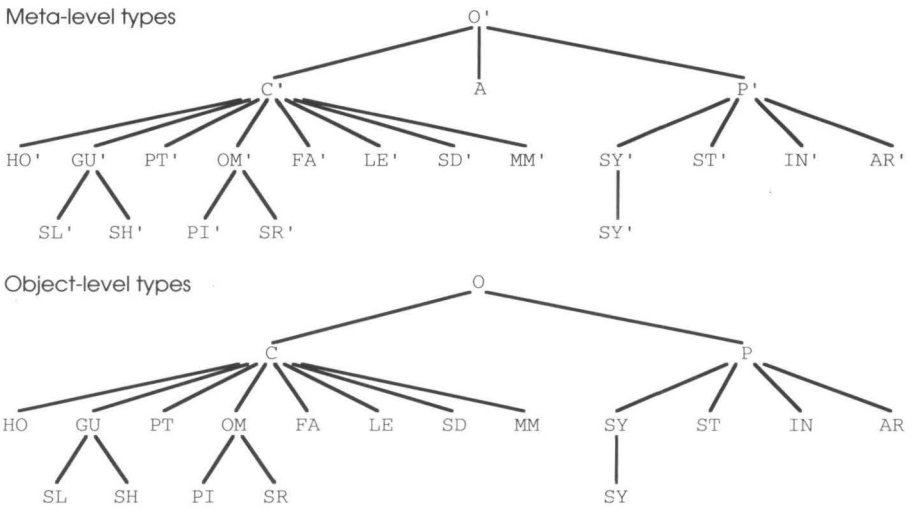


Fig. 9.4 Meta-level and object-level type hierarchy.

The signature of a scenario `s` is denoted by Σ`(s)`. Certain predicates in the system (e.g. `equal`, `greaterEqual`, `value`, `uiNumber`, `isNil`, etc.) are evaluable predicates. When one of these predicates is encountered in a rule, the procedure attached to it will be evaluated (see Chapter 6 for precise descriptions). The design problem, that this chapter deals with, is a part of larger design problem. Only those scenarios, which exclusively deal with the linear motion mechanism are presented in this chapter. Therefore, the process actually steps into a certain process information state of the overall design. The signature that describes that state is Σ`(process-information-state)`. The column '**`Meta-constants`**' contains references to object-level proposition symbols. The column '**`Process parameters`**' contains the process information state being reached so far.

Σ(process-information-state)

Type	Short	Meta-constants
objectAtom	A	level0, linearMotion
Meta-predicate	**Type**	**Process parameters**
goal	A	goal(level0), goal(linearMotion)

The fact-base initially has the signature Σ(fact-base). The column '**Constants**' gives the constants which are present in the fact-base at the initial state of the system. The column '**Literals**' presents the literal facts being present at that state. Since the process information state and the fact-base grow during the design process, its signature will grow as well. The extended signatures remain within the maximum signatures presented in Appendix 3.

Σ(fact-base)

Type	Short	Constants
lever	LE	lever1
pivot	PV	pivot1
face	FA	face1, face2, face3, face4
composite	C	
Predicate	**Type**	**Literals**
isLever,		isLever(lever1),isPivot(pivot1)
isPivot,		isFace(face1),isFace(face2)
isFace	C	isFace(face3),isFace(face4)
adjacent	FA×FA	adjacent(face1,face2),adjacent(face2,face3)
		adjacent(face3,face4),adjacent(face4,face1)
hasPart	C×C	hasPart(lever1,face1),hasPart(lever1,face2)
		hasPart(lever1,face3),hasPart(lever1,face4)
		hasPart(lever1,pivot1)

9.2.2 Overall design process

The overall design process is controlled by the meta-level scenario solveLinearMotion. The design process is subdivided into three stages, conceptual, fundamental, and detailed design. During the first stage of design, the system constructs an abstract anatomical description of the design object. Then, during fundamental design, it further models the description modeled until a more concrete description is obtained. If this description is not precise enough, it is refined during the detailed stage. The following rules and the signature in Appendix 3 specify the knowledge-base and the language of the scenario respectively.

```
Name(solveLinearMotion)
Meta-rules
1   IF      unknown(isGuide(ω))
    THEN    goal(motionMetaModel)      "conceptual design"
    &       goal(guideSpecs)
2   IF      abstract(isGuide(ω))
    THEN    goal(guideGeometry)        "fundamental design"
    &       goal(motionQualities)
    &       goal(guideLimits)
3   IF      concrete(isGuide(ω))
    &       ~exact(isGuide(ω))
    THEN    goal(motionFault)          "detailed design"
    &       goal(guideRefinement)
    &       goal(guideLimits)
4   IF      exact(isGuide(ω))
    THEN    success(linearMotion)
```

The scenario 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 object in motion can move unobstructed inside the guide of the motion, i.e., the design meets the requirements as imposed by the designer. The process parameter `exact` denotes this fact. This meta-predicate symbol is asserted when the description of the involved object is an exact anatomical description. Therefore, the condition `exact(isGuide(ω))` evaluates to true when there exists an *exact* anatomical description of a guide. The goal `goal(linearMotion)` and hence the design of a linear motion mechanism succeeds, if this condition is met. Below, I treat the first three rules in more detail.

The first rule of `solveLinearMotion` reads as follows: if the definition of an object of type `guide` is specified as unknown in the process information state then assert the conjunction of goals:

```
goal(motionMetaModel) & goal(guideSpecs).
```

When these two goals have been satisfied, the system has created an initial abstract anatomical model of the design object. Furthermore, the specifications for the linear motion mechanism have been given in dialogue with the designer. The variable ω in the function `isGuide(ω)` is an *pseudo* variable that denotes that the programmer does not care to which constant the argument of the function is actually bound. This mechanism is similar to the "don't care" symbol used in Prolog [Clocksin and Mellish, 1981].

Next, the second rule is applied if there exists an object of type `guide` in the fact-base and if an *abstract* anatomical description of that object has been constructed. Note that the query `abstract(isGuide(ω))` succeeds if either a guide or an object which is defined as a subtype of a guide is found (see the discussion in Chapter 6 on the object definition built-in predicates). For example, the unification of `isGuide(ω)` and `isSlot(slot1)` succeeds because `slot` is a subtype of `guide`. The rule asserts the conjunction of goals:

```
goal(guideGeometry) & goal(motionQualities)
& goal(guideLimits).
```

The purpose of these goals is to express that a concrete anatomical model of the design object should be built by defining the objects' geometrical structures and by defining kinematic properties. Determining the limit positions of the motion mechanism is the goal of `guideLimits`. These positions are used to check whether the object in motion is inside the guide.

The third rule is applied when a *concrete* anatomical structure of the guide has been described that does *not* satisfy the requirements, i.e., the description is not yet *exact* (the meta-predicate symbol `concrete` and `exact`). The third rule asserts the conjunction of goals:

```
goal(motionFault) & goal(guideRefinement)
& goal(guideLimits).
```

The goal of `motionFault` is finding inconsistencies between the geometric and kinematic aspect models. The properties of the guide are adjusted dependent on the kind of inconsistency and a new geometric model will be obtained. The second goal (`guideRefinement`) aims at achieving these two issues. Note that the last goal is the same as that of the previous rule. If the object has an exact description (`exact`), then the design has been completed. Otherwise, the third rule is applied again trying to find a different solution.

Each of the first three rules of `solveLinearMotion` represents a certain stage of the design process. The first rule expresses when conceptual design should be done, the second rule does the same with respect to fundamental design, and the third rule stands for detailed design. The *backtrack* rule selection method controls the execution of a rule, i.e., if the condition of a rule fails, the previous rule is tried (if its condition still holds). 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 `isGuide(ω)` is unknown. In other words, an object of type `guide` cannot be found in the meta-model. By the same token, the third rule is applied as long as the object in motion does not have an exact anatomical description. In the following three sections I explain each of these three design stages, and I show the state of the meta-model at the end of each stage.

9.3 Conceptual design of lever and pin

In this section, I show how the conceptual design of a linear motion mechanism is performed. At this stage the process information state contains the following process parameters (the length of the dashed line indicates the level of control):

```
goal(linearMotion),
---goal(motionMetaModel) & goal(guideSpecs).
```

The bottom line of the meta-facts contains a conjunctions of two goals. The meta-level interpreter schedules the lastly asserted (conjunction of) goal(s) at highest priority. The conjunction will successively be solved by the object-level scenario `solveMotionMetaModel` and the meta-level scenario `solveGuideSpecs`. The former establishes an initial meta-model of the linear motion mechanism, and the latter gives two new design goals to give the specifications for the motion mechanism and for the slot and pin.

9.3.1 SolveMotionMetaModel

The object-level scenario `solveMotionMetaModel` has the following rules.

Name(solveMotionMetaModel)
Rules

```
1   IF      isLever(M) & typeFor(S,slot) & typeFor(P,pin)
    THEN    isSlot(S) & hasPart(M,S) & isPin(P)
    &       isMotion(M)
2   IF      isSliderDevice(M) & typeFor(S,shaft)
    &       typeFor(Sl,slider)
    THEN    isShaft(S) & hasPart(M,S) & isSlider(SL)
    &       isMotion(M)
3   IF      isObjectInMotion(O) & isGuide(G)
    &       typeFor(P1,point) & typeFor(P2,point)
    THEN    isPoint(P1) & isPoint(P2) & linearMotion(O,P1,P2)
    &       limitArrangement(O,G,P1) & limitArrangement(O,G,P2)
    &       motionMetaModel
```

The scenario `solveMotionMetaModel` creates an abstract anatomical representation of the design object. It asserts the entities which construct the linear motion mechanism. The first rule is applied when a guide is not yet known and when the device which supports the motion mechanism is a lever. The rule asserts a slot, being part of the lever, and a pin to the fact-base. Furthermore, it defines the lever as an object of a multiple type by asserting

```
isMotion(lever1).
```

By doing so, the lever inherits the attributes of both the type `lever` and `motion`.

In another design, when the mechanism involved is a 'slider-device', the fact-base will contain the literal fact `isSliderDevice()`. As a result, the condition of the second rule will succeed and the second rule will be applied. In that case, a shaft and a slider are asserted. In this chapter, I discuss the situation in which the first rule is applied. The third rule of `solveMotionMetaModel` stores the general qualitative properties of a linear motion mechanism in the fact-base. These properties are independent of a chosen solution and are used to check and maintain the consistency of the design object model. The rule states that there are two points:

```
isPoint(point1) & isPoint(point2)
```

and the object in motion (in this case the pin) makes a linear motion between these points:

```
linearMotion(pin1,point1,point2)
```

Furthermore, in both points there exists a limit arrangement between the object in motion and the guide of the motion (in this case the slot):

```
limitArrangement(pin1,slot1,point1)
limitArrangement(pin1,slot1,point2)
```

These limit arrangements are used to determine at a later stage of the design whether the object in motion is inside the guide of the motion. When these facts are asserted to the fact-base, the abstract anatomical description is made. It asserts through the proposition symbol `motionMetaModel` that the scenario's goal has been reached. The scenario terminates but the original goal consists of a conjunction of two goals. Therefore, the meta-level scenario `solveGuideSpecs` will subsequently become active.

9.3.2 SolveGuideSpecs

This meta-level scenario derives that a conjunction of two goals is relevant, the first goal deals with the specifications for the motion mechanism, the second one deals with the specifications for either a slot or a shaft dependent on the chosen solution. The scenario's rules are given below.

Name(solveGuideSpecs)
Meta-rules

1	**IF**	positive(isSlot(ω))
	THEN	goal(motionSpecs) & goal(slotSpecs)
2	**IF**	positive(isShaft(ω))
	THEN	goal(motionSpecs) & goal(shaftSpecs)
3	**IF**	positive(isGuide(G)) & success(motionSpecs)
	THEN	abstract(isGuide(G)) & success(guideSpecs)

The scenario has three rules but only one of the first two rules will be applied, in this case the one that is applicable to the slot and pin solution. When the first rule is applied the process information state is extended by a conjunction of two goals. Its contents is now:

```
goal(linearMotion),
---goal(motionMetaModel) & goal(guideSpecs),
---success(motionMetaModel),
------goal(motionSpecs) & goal(slotSpecs).
```

The last rule of the scenario asserts that the anatomical description of a guide is abstract if there is a positive fact `isGuide(G)` in the object information state and the goal named `motionSpecs` has been satisfied. Note however, that both the positive fact `isSlot(slot1)` and `isShaft(shaft1)` match against the query `isGuide(G)` since they are both subtypes of `guide`. But first, the scenario `solveMotionSpecs` becomes active.

9.3.3 SolveMotionSpecs

In this object-level scenario the specifications for the linear motion mechanism are given. These specifications are constrained as shown in Table 3.2 of Chapter 3. The scenario has the following rules and operations.

Name(solveMotionSpecs)
Rules

```
1   IF      isMotion(M) & isNil(M:start)
    &       uiNumber('start of motion',S,M:halfWidth,M:innerRange)
    THEN    value(M:start,S)
2   IF      isMotion(M) & isNil(M:end)
    &       uiNumber('end of motion',E,M:startWidth,M:motionRange)
    THEN    value(M:end,E)
3   IF      isMotion(M) & notNil(M:start) & notNil(M:end)
    THEN    motionSpecs
```

Local operations

```
:halfWidth = { self:width * 0.5 }
:innerRange = { self:range - self:width * 1.5 }
:startWidth = { self:start + self:width }
:motionRange = { self:range - self:width * 0.5 }
```

This scenario is active until the requirements for both the starting and ending point of the motion are specified. The first rule asks the designer to supply a value for the start of the motion. It must be greater than or equal to half the width of the lever and it must be smaller than or equal to the distance of the right face of the lever minus one and a half times the width of the lever. The second rule does the

same for the end of the motion. This value must be greater than or equal to the starting point plus the width of the lever and it must be smaller than or equal to the distance of the right face of the lever minus half the width of the lever.

Finally, the scenario succeeds if the condition of the third rule is satisfied, viz. both the start and the end of the motion mechanism are known. The meta-level interpreter gives control to the scenario associated with the next goal namely, the object-level scenario `solveSlotSpecs`.

9.3.4 SolveSlotSpecs

In this scenario the designer gives the specifications for the slot and pin. These specifications are constrained as is shown in Table 3.3 of Chapter 3. The following rules and operations specify the scenario.

```
Name(solveSlotSpecs)
Rules
-------------------------------------------------------------------
1   IF      isSlot(S) & isNil(S:position) & isLever(L)
    &       uiNumber('position of slot',X,10,L:maxPosition)
    THEN    value(S:position,X)
2   IF      isSlot(S) & isNil(S:length)
    &       notNil(S:position) & isLever(L)
    &       uiNumber('length of slot',X,L:width,L:maxLength[S])
    THEN    value(S:length,X)
3   IF      isSlot(S) & isNil(S:width) & isLever(L)
    &       uiNumber('width of slot',X,10,L:widthMinusTol)
    THEN    value(S:width,X)
4   IF      isPin(P) & isNil(P:diameter)
    &       isSlot(S) & notNil(S:width)
    &       uiNumber('diameter of pin',X,10,S:width)
    THEN    value(P:diameter,X)
5   IF      isSlot(S) & notNil(S:length) & notNil(S:width)
    &       isPin(P) & notNil(P:diameter)
    THEN    slotSpecs
Local operations
-------------------------------------------------------------------
:maxPosition = { self:range - self:width - 10 }
:maxLength[S] = { self:range - S:position - 10 }
:widthMinusTol = { self:width - 20 }
```

The rules of `solveSlotSpecs` determine in dialogue with the designer the attribute values of the slot and the pin. These specifications are constrained to the effect that the slot is always positioned inside the lever. The constraints do not check whether the slot is a valid solution to the linear motion mechanism. The

meta-model mechanism validates the consistency of the design. Therefore, the attribute values given in this scenario are assumptions which might be revised during the design process. This happens during the detailed design phase.

The first rule determines the position of the slot which is constrained by the length of the lever, and the minimum length of the slot itself, which is equal to the width of the lever. The second rule gives the length of the slot, constrained by its position, and the range of the lever. The width of the slot is provided by the third rule, constrained by the width of the lever minus a certain tolerance. The fourth rule gives the diameter of the pin, which must be smaller or equal to the width of the slot. When all attributes are set, the last rule will be applied, and the scenario will terminate.

9.4 Fundamental design of lever and pin

The completion of the first rule of the top-level scenario and the application of the second rule result in the following set of process parameters of the process information state

```
goal(linearMotion),
---goal(motionMetaModel) & goal(guideSpecs),
---success(motionMetaModel),
------goal(motionSpecs) & goal(slotSpecs),
------success(motionSpecs), success(slotSpecs),
------abstract(isGuide(slot1)),
---success(guideSpecs),
---goal(guideGeometry) & goal(motionQualities)
   & goal(guideLimits).
```

At the beginning of fundamental phase of the design process the meta-model consists of an abstract anatomical description of the design object. During the course of fundamental design the meta-model 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 of the linear motion are determined by a kinematic aspect model. The aspect models are created by the scenarios `solveGuideGeometry` and `solveGuideLimits` respectively. However, the length of the motion can only be determined if I know the starting and the ending position of the motion. Therefore, when the geometry of the motion mechanism is set up, these positions are detected, for there is a contact between the object in motion and the guide of the motion in these positions. This is done by the scenario `solveMotionQualities`.

9.4.1 SolveGuideGeometry

The scenario `solveGuideGeometry` is very similar to `solveGuideSpecs`. It asserts a conjunction of three goals, the first to build the geometry of the guide, the

second to determine the relative angle of all faces, and the last to create a wire-frame modeler to show the actual geometry. The scenario of the latter will not be presented, because it is rather technical and it does not contribute relevant information to the reasoning process. The rules of the (meta-level) scenario are:

```
Name(solveGuideGeometry)
Meta-rules
_____________________________________________________________
1   IF      positive(isSlot(ω))
    THEN    goal(slotGeometry) & goal(angleOfFaces)
    &       goal(slotWireframe)
2   IF      positive(isShaft(ω))
    THEN    goal(shaftGeometry) & goal(angleOfFaces)
    &       goal(shaftWireframe)
3   IF      positive(isGuide(G))
    &       (success(slotGeometry) | success(shaftGeometry))
    THEN    concrete(isGuide(G)) & success(guideGeometry)
```

The scenario consists of three rules. Regarding the first two rules either the first or the second is applied depending on the type of linear motion mechanism. In this particular case the first rule is applied asserting the conjunction of goals

```
goal(slotGeometry) & goal(angleOfFaces).
& goal(slotWireframe)
```

The first goal is to build a geometric model of the slot and pin construction. The purpose of the second goal is determining for each face the angle which it makes whit the x-axis. The third goal activates a geometric modeler which shows a wire-frame representation of the current state of the design object. The last rule of the scenario asserts that the anatomical description of a guide is concrete, if either the goal `slotGeometry` or `shaftGeometry` has been satisfied. Furthermore, it conclude that the goal of the scenario has been reached.

9.4.2 SolveSlotGeometry

The object-level scenario `solveSlotGeometry` builds a geometric representation of the slot and pin according to specifications given by the designer. A slot has a rectangular shape consisting of four adjacent faces. The faces have an anti-clockwise orientation. A face has three attributes, an x-coordinate, a y-coordinate, and an angle. For simplification, I use a 2-dimensional model. I assume that both the lever and slot lay within a single surface, i.e., the x-y plane. The x- and y-coordinates specify the starting-point of a face. The ending-point of the face is specified by the starting-point of the face it is adjacent to. A pin has only a single face whose geometry is specified by the centre of the pin and the diameter. The geometry of both the slot and pin has been given in Fig. 3.5 of Chapter 3. The scenario has the following rules and operations.

```
Name(solveSlotGeometry)
Rules
1   IF      isSlot(S) & isFace(F1) hasPart(S,F1)
    &       adjacent(F1,F2) & adjacent(F2,F3)
    &       adjacent(F3,F4) & adjacent(F4,F1)
    THEN    value(F1:x,S:position) & value(F1:y,S:halfWidthUp)
    &       value(F2:x,S:position) & value(F2:y,S:halfWidthDown)
    &       value(F3:x,S:posLength) & value(F3:y,S:halfWidthDown)
    &       value(F4:x,S:posLength) & value(F4:y,S:halfWidthUp)
    &       slotGeometry
2   IF      isSlot(S) & typeFor(F1,face) & typeFor(F2,face)
    &       typeFor(F3,face) & typeFor(F4,face)
    THEN    isFace(F1) & isFace(F2) & isFace(F3) & isFace(F4)
    &       hasPart(S,F1) & hasPart(S,F2) & hasPart(S,F3)
    &       hasPart(S,F4) & adjacent(F1,F2) & adjacent(F2,F3)
    &       adjacent(F3,F4) & adjacent(F4,F1)
3   IF      isPin(P) & typeFor(F,face)
    THEN    isFace(F) & hasPart(P,F)
Local operations
:halfWidthUp = { self:width / 2 }
:halfWidthDown = { (self:width / 2):negated }
:posLength = { self:position + self:length }
```

The scenario consists of two parts. The first part (i.e., the first rule) determines the x- and y-coordinates of the slot's faces. The second part (i.e, the second and third rule) initializes the slot's and pin's geometry. This structure makes the scenario generally applicable. If the geometry has not yet been initialized, the condition of the first rule will not hold. The second and third rule will firstly be applied. After that the condition of the first rule does hold and the first rule will as yet be applied. Otherwise, the first rule will immediately be applied and the scenario terminates successfully without applying the second and third rule.

The second rule asserts four adjacent faces to the fact-base. These faces are part of the slot. The pin has a single face asserted by the third rule. The coordinates of the faces of the slot are specified by the first rule in the following way. It detects four faces which are part of the slot and which are oriented in an anti-clockwise fashion. If two faces are adjacent, then they share a vertex, viz. the starting-point of one face is the ending-point of the other. The coordinates of the vertices of the faces are determined by the angle points of the slot. The position of the slot, the width of the slot and the length of the slot ascertain these coordinates uniquely.

9.4.3 SolveAngleOfFaces

The object-level scenario `solveAngleOfFaces` can generally be applied to determine the angle of the faces of an object whose geometry is defined in terms of more than two faces. Its rules are:

```
Name(solveAngleOfFaces)
Rules
1   IF      isFace(F1) & isNil(F1:angle) & adjacent(F1,F2)
    THEN    value(F1:angle,F1:angle[F2])
2   IF      isFace(F1) & adjacent(F1,F2)
    &       ~isNil(F1:angle) & ~isNil(F2:angle)
    THEN    angleOfFaces
```

The first rule of the scenario takes two adjacent faces and determines the angle of the former with the x-axis by using its starting-point and the starting-point of the latter. The goal is solved when there are no more adjacent faces with an unknown angle. Extending the meta-model of the linear motion mechanism is the next step to be performed.

9.4.4 SolveMotionQualities

When the geometry of the slot and the pin has been defined, the system can describe the qualitative behaviour of the linear motion mechanism in detail. Remember that I defined the limit arrangements for the starting and the ending position of the motion. These limit arrangements are defined by a contact between the face of the pin and a face of the slot. The scenario `solveMotionQualities` detects the faces which have such a contact. Furthermore, it defines the starting- and ending-position of the motion. Its rules follow.

```
Name(solveMotionQualities)
Rules
1   IF      limitArrangement(O,G,P) & linearMotion(O,P,ω)
    &       isFace(F1) & hasPart(O,F1) & isFace(F2)
    &       hasPart(G,F2) & equal(F2:angle,270)
    THEN    startPosition(P) & contact(F1,F2,P)
2   IF      limitArrangement(O,G,P) & linearMotion(O,ω,P)
    &       isFace(F1) & hasPart(O,F1) & isFace(F2)
    &       hasPart(G,F2) & equal(F2:angle,90)
    THEN    endPosition(P) & contact(F1,F2,P)
3   IF      startPosition(ω) & endPosition(ω)
    THEN    motionQualities
```

The first rule of the scenario defines the starting position of the motion by satisfying the following condition. If there is a linear motion of the pin between

points P1 and P2, there is a limit arrangement involving the pin and the slot in P1, there is a face F1 being part of the pin, there is a face F2 being part of the slot, and the angle between F2 and the x-axis is 270 degrees anti-clockwise, then P1 is the starting-position of the motion, and there is a contact between F1 and F2 in this position, i.e.,

```
startPosition(point1)
contact(face9,face5,point1).
```

The second rule defines the ending-position P2 in a similar way. In this case, however, the angle of the face of the slot must be 90 degrees. It results in

```
endPosition(point2)
contact(face9,face7,point2).
```

The last rule asserts that a concrete anatomical description of the guide has been established and that the scenario can succeed if both the starting and the ending position of the motion have been determined.

9.4.5 SolveGuideLimits

The meta-level scenario solveGuideLimits behaves like the meta-level scenario solveGuideGeometry. It asserts a goal to determine the actual coordinates of the starting- and ending-point of the linear motion of the pin guided by the slot. The rules are:

Name(solveGuideLimits)
Meta-rules

```
1   IF      positive(isSlot(ω))
    THEN    goal(slotLimits)
2   IF      positive(isShaft(ω))
    THEN    goal(shaftLimits)
3   IF      positive(isMotion(M)) & positive(isGuide(G))
    &       positive(startPosition(P1))
    &       positive(endPosition(P2))
    &       positive(smallerEqual(P1:x,M:start))
    &       positive(greaterEqual(P2:x,M:end))
    THEN    exact(isGuide(G))
4   IF      success(slotLimits) | success(shaftLimits)
    THEN    success(guideLimits)
```

Regarding the first two rules, only the first rule is applied. The second rule deals with a different kind of solution. The first rule asserts the goal

```
goal(slotLimits)
```

to the set of process parameters of the process information state. At this instant, its

contents is as follows:

```
goal(linearMotion),
---goal(motionMetaModel) & goal(guideSpecs),
---success(motionMetaModel),
------goal(motionSpecs) & goal(slotSpecs),
------success(motionSpecs), success(slotSpecs),
------abstract(isSlot(slot1)),
---success(guideSpecs),
---goal(guideGeometry) & goal(motionQualities)
   & goal(guideLimits),
------goal(slotGeometry) & goal(angleOfFaces)
      & goal(slotWireframe),
------success(slotGeometry), success(angleOfFaces),
------success(slotWireframe),
------concrete(isSlot(slot1)),
---success(guideGeometry),
---success(motionQualities),
------goal(slotLimits),
```

The goals `guideGeometry` and `motionQualities` have been solved. The next goal to be solved is `slotLimits`.

The third rule of `solveGuideLimits` applies kinematic knowledge. It verifies whether the linear motion mechanism meets the requirements given by the designer. The limit positions specify the starting- and the ending-point (S_p and E_p respectively) of the motion. The design fulfills the specifications if the x-coordinate of the starting-point is smaller than or equal to the starting of the motion S_0 specified by the designer, and x-coordinate of the ending-point is greater than or equal to the end of the motion E_0 specified by the designer, i.e.,

$$S_p \leq S_0 \quad \wedge \quad E_p \geq E_0 .$$

If this condition is fulfilled, the rule asserts that an exact anatomical description of the slot or the shaft has been made, i.e., the relevant attributes have all received a value *and* these values meet the requirements as imposed by the designer. The design is complete as far as the guide is concerned. The last rule finally states that the goal of the scenario has been reached, if either the goal `slotLimits` or `slotLimits` has been satisfied.

At this state of the design process the description of the design object is obtained by using a geometric aspect model and a kinematic aspect model. If the description satisfies the requirements imposed by the designer, it has been found that the pin can move inside the slot without being obstructed and the design is *exact*. However, when this condition is not fulfilled, the obstruction must be removed, i.e., the design must be improved. The implication is that the geometry of the slot is changed, and a revision process must take place.

9.4.6 SolveSlotLimits

The object-level scenario `solveSlotLimits` represents a kinematic aspect-model of the linear motion mechanism. It calculates the coordinates of the starting- and ending-points of the motion. These limit positions are then used to check whether the design satisfies the requirements, i.e., the pin can move unimpededly inside the slot. The scenario's rules and operations are:

Name(solveSlotLimits)
Rules

```
1   IF      startPosition(Pt) & contact(ω,F,Pt)
    &       isNil(Pt:x) & isNil(Pt:y) & isPin(P)
    THEN    value(Pt:x,F:startPoint[P]) & value(Pt:y,0)
2   IF      endPosition(Pt) & contact(ω,F,Pt)
    &       isNil(Pt:x) & isNil(Pt:y) & isPin(P)
    THEN    value(Pt:x,F:endPoint[P]) & value(Pt:y,0)
3   IF      startPosition(Pt1) & notNil(Pt1:x)
    &       endPosition(Pt2) & notNil(Pt2:x)
    THEN    slotLimits
```

Local operations

```
:startPoint[P] = { self:x + P:diameter / 2 }
:endPoint[P] = { self:x - P:diameter / 2 }
```

Since the system has asserted that there is a contact between the face of the pin and a face of the slot in two limit arrangements, the scenario `solveSlotLimits` is able to compute the limit positions of the motion mechanism. The first rule in `solveSlotLimits` calculates the coordinates of the starting-position of the motion as follows. The centre of the pin in a limit position determines the starting of the motion. In such a limit position, there is a contact between the face of the pin and one of the faces of the slot. Therefore, the x-coordinate of the starting-position is equal to that of the slot's face plus half the diameter of the pin (see Fig. 9.5).

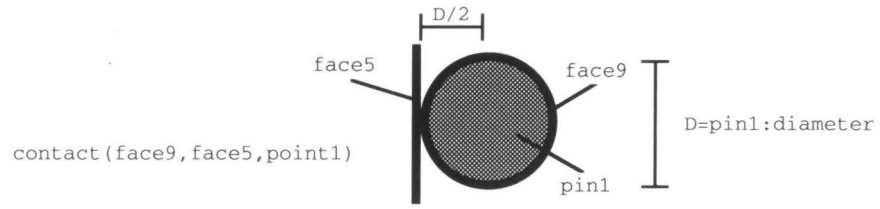


Fig. 9.5 Kinematic model of an object in motion.

The second rule of the scenario computes the coordinates of the ending-position of the motion in the same fashion. There is a contact between another face of the slot and the pin in the second limit position. In this case, the x-coordinate of the ending-position equals to the x-coordinate of the slot's face minus half the diameter of the pin. The y-coordinate of the starting and the ending position is set to zero, since the pin moves along the centre line of the lever. Finally, the scenario succeeds, when the last rule is applied, viz. when the x-coordinates of both the starting- and ending-position are known.

9.5 Detailed design of lever 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 estimated specifications of the designer using heuristic knowledge. The obtained geometry might result in an incorrect stroke length. The system is able to detect such an inconsistency in the meta-model, since the meta-model integrates knowledge about both models. In this section, I show how the cause of the inconsistency is detected through a kinematic model and how it is repaired by changing the geometric model of the design object (see Fig. 9.6).

Fig. 9.6 Relation between a kinematic and a geometric aspect model.

At the detailed stage of the design process, the process information state contains the following set of process parameters

```
goal(linearMotion),
---goal(motionMetaModel) & goal(guideSpecs),
---success(motionMetaModel),
------goal(motionSpecs) & goal(slotSpecs),
------success(motionSpecs), success(slotSpecs),
------abstract(isSlot(slot1)),
---success(guideSpecs),
---goal(guideGeometry) & goal(motionQualities)
   & goal(guideLimits),
------goal(slotGeometry) & goal(angleOfFaces)
      & goal(slotWireframe),
```

```
------success(slotGeometry), success(angleOfFaces),
------success(slotWireframe),
------concrete(isSlot(slot1)),
---success(guideGeometry), success(motionQualities),
------goal(slotLimits), success(slotLimits),
---success(guideLimits),
---goal(motionFault) & goal(guideRefinement)
& goal(guideLimits).
```

The system behaves as follows. First of all, the stroke fault of the motion is calculated. Secondly, the geometry of the slot is adjusted, and thirdly the new limit positions of the motion are determined and the consistency of the design is verified.

9.5.1 SolveMotionFault

The object-level scenario `solveMotionFault` identifies the cause of the inconsistency in the design. It represents a kinematic aspect model of the motion mechanism. Dependent on the nature of the inconsistency it will suggest to either shift the left most face of the slot to the left, the right most face of the slot to the right, or both (see Fig. 9.7). The rules and operations of `solveMotionFault` are shown below.

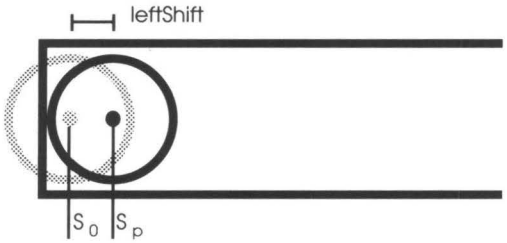


Fig. 9.7 Detection of inconsistency.

Name(solveMotionFault)
Rules

1	**IF**	isMotion(M) **&** startPosition(P) **&** greater(P:x,M:start)
	THEN	leftShift(P:leftMinus[M])
2	**IF**	isMotion(M) **&** endPosition(P) **&** smaller(P:x,M:end)
	THEN	rightShift(P:rightMinus[M])
3	**IF**	leftShift(ω) **|** rightShift(ω)
	THEN	motionFault

Local operations

```
:leftMinus[M] = { self:x - M:start }
:rightMinus[M] = { M:end - self:x }
```

The first rule of the scenario is applied when the pin is obstructed by the left most face of the slot. Consequently, that face is shifted to the right by $S_p - S_0$, e.g.

```
leftShift(20).
```

The second rule is applied when the pin is obstructed by the right most face, resulting in a shift to the left by $E_p - E_0$, e.g.

```
rightShift(20).
```

The scenario succeeds if at least one of the first two rules is applied. Solving the detected inconsistency is the next goal. It is done by adjusting the geometry of the slot either by a shift to the right, or a shift to the left, or both.

9.5.2 SolveGuideRefinement

The meta-level scenario `solveGuideRefinement` is similar to `solveGuideLimits`. It asserts a goal to adjust the geometry of the slot dependent on the detected inconsistency in the design. The scenario's rules are:

Name(solveGuideRefinement)
Meta-rules

```
1   IF      positive(isSlot(ω))
    THEN    goal(slotRefinement) & goal(angleOfFaces)
2   IF      positive(isShaft(ω))
    THEN    goal(shaftRefinement) & goal(angleOfFaces)
3   IF      success(slotRefinement) | success(shaftRefinement)
    THEN    success(guideRefinement)
```

The scenario's first rule is applied resulting in the assertion of the following conjunction of goals to the set of process parameters:

```
goal(slotRefinement) & goal(angleOfFaces)
```

When either the goal `slotRefinement` or the goal `shaftRefinement` has been satisfied, the third rule infers that the goal `guideRefinement` has been satisfied as well.

9.5.3 SolveSlotRefinement

The object-level scenario `solveSlotRefinement` revises the designer's original specifications of the slot to obtain a consistent description of the linear motion mechanism. Dependent on the cause of the fault the scenario changes the position of the slot and/or the length. As a result, the slot's geometry, i.e., the position and

the length of the faces, changes as well. The scenario's rules and operations are:

```
Name(solveRefineSlot)
Rules
1   IF      leftShift(L) & rightShift(R) & isSlot(S)
    THEN    value(S:position,S:leftShift[L])
    &       value(S:length,S:rightShift[L,R]) & slotRefinement
2   IF      leftShift(L) & isSlot(S)
    THEN    value(S:position,S:leftShift[L])
    &       value(S:length,S:rightShift[L]) & slotRefinement
3   IF      rightShift(R) & isSlot(S)
    THEN    value(S:length,S:rightShift[R]) & slotRefinement
Local operations
:leftShift[L] = { self:position - L }
:rightShift[R] = { self:length + R }
:rightShift[L,R] = { self:length + L + R }
```

There are three possible states of inconsistency to be solved by the scenario `solveSlotRefinement`, i) the slot's left most face impedes the motion, ii) the slot's right most face impedes the motion, or iii) both the slot's left most face and right most face impede the motion. Each of the first three rules of the scenario is applicable to one of the three respective cases. When the left most face obstructs the motion, the first rule moves the position of the slot to the right and accordingly adjusts the slot's length. The second rule is applied when the right most face obstructs the motion. It adjusts the length of the slot. In case both the left and right most face of the slot need to be adjusted, the third rule moves the position of the slot to the right and increases the length doubly. Since these three situations always occur separately, only one of the three rules is applied.

Either of the first three rules changes one or more attributes of the slot. I call such attributes object-facts. E.g. the first rule changes both the length and the position of the slot. Such a change causes a revision of the fact-base. All object-facts which depend on the changed facts are reset to nil. During the the design process the system keeps record of these dependencies (The mechanism is described in Chapter 5). The dependency graph of position and the length of the slot is shown in Fig. 9.8. Four object-facts depend on the position of the slot, (viz. `face5:x`, `face6:x`, `face7:x`, and `face8:x`). Two object-facts depend on the length of the slot, (viz. `face7:x`, and `face8:x`). Moreover, these object-facts have again other object-facts depending on them. E.g. `point1:x`, `face5:angle`, and `face6:angle` depend on `face5:x`.

The revision process activated by the change of the slot's attributes, causes a redoing of part of the design process. Since the x-coordinates of the slot depend

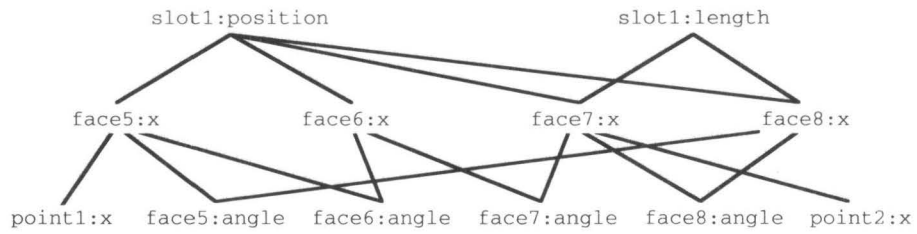


Fig. 9.8 Dependency graph of attributes of objects.

on the position and length of the slot, they need to be recalculated. This is done by the the fourth and the fifth rule of `solveSlotRefinement` dependent on the kind of refinement. The fourth rule is used when there is a shift to the left, and the fifth rule is applied when there is a shift to the right.

The scenario `solveSlotRefinement` detects the proper face to be adjusted, since the meta-model 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 meta-model the system would not have been able to create a geometric model independent of certain properties which are determined by a kinematic model. The meta-model 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 meta-model has qualitative knowledge about the relation between kinematic motion and geometry.

After completion of the scenario, the process information state has been extended with the following process parameters:

```
------success(slotLimits),
---success(guideLimits),
---goal(motionFault) & goal(guideRefinement),
   & goal(guideLimits),
---success(motionFault),
------goal(slotRefinement) & goal(slotGeometry)
      & goal(angleOfFaces).
```

The next scenario which becomes active is `angleOfFaces`. This scenario is presented in §9.4.3. It determines unknown angles of faces caused by the revision process. After completion of this scenario, the scenario `solveGuideRefinement` succeeds as well, and control is given to `solveGuideLimits`, presented in §9.4.5. The new limit positions of the motion mechanism are determined, and if the pin is

inside the slot, i.e., the mechanism fulfills its requirements, the an exact anatomical description is made and the design is complete. The final contents of the process information state is the following set of process parameters.

```
goal(linearMotion),
---goal(motionMetaModel) & goal(guideSpecs),
---success(motionMetaModel),
------goal(motionSpecs) & goal(slotSpecs),
------success(motionSpecs), success(slotSpecs),
------abstract(isSlot(slot1)),
---success(guideSpecs),
---goal(guideGeometry) & goal(motionQualities)
   & goal(guideLimits),
------goal(slotGeometry), goal(angleOfFaces),
      goal(slotWireframe),
------success(slotGeometry), success(angleOfFaces),
------success(slotWireframe),
------concrete(isSlot(slot1)),
---success(guideGeometry),
---success(motionQualities),
------goal(slotLimits), success(slotLimits),
---success(guideLimits),
---goal(motionFault) & goal(guideRefinement),
   & goal(guideLimits),
---success(motionFault),
------goal(slotRefinement) & goal(slotGeometry)
      & goal(angleOfFaces),
------success(slotRefinement), success(slotGeometry),
      success(angleOfFaces),
---success(guideRefinement),
---------goal(slotLimits), success(slotLimits),
---------exact(isGuide(slot1)),
---success(guideLimits),
success(linearMotion).
```

9.6 Discussion

This chapter shows how ADDL can be used to model a design object independent of a certain context. The meta-model mechanism provides such a modeling technique. Besides, it demonstrates the use of meta-level reasoning to control the design process. I can make two observations regarding the generation of assumptions upon which the meta-level interpreter makes strategic decisions. First of all, the assumptions are all made at the object-level. E.g., the object-level scenario `solveSlotSpecs` generates assumptions for the attributes of a slot. The reason for this is inherent in the use of the reflection principle. In its current version, ADDL only uses upward reflection. Therefore, assumptions generated at the object-level can be transformed to process parameters in the process

information state. The reverse, i.e., the transformation of meta-level assumptions to object-level literal facts appearing in the fact-base, is not possible.

The latter can be achieved though if ADDL uses *downward reflection* as well. For example, DESIRE employs the meta-level built-in predicate `possibleAssumption` that generates an assumption at the meta-level [Kowalczyk and Treur, 1990]. Downward reflection transforms its argument to the object-level. Thus the object-level literal fact `value(slot1:position,10)` can be generated by asserting the meta-atom

```
possibleAssumption(value(S:position,X))
```

and by a downward transformation to the fact-base (also called object information state). A future version of ADDL may also have downward reflection.

The second observation regards the role of the designer in the design process. The example design system employs the heuristic knowledge of the designer to generate assumptions about the slot's specifications. For example, the object-level scenario `solveSlotSpecs` queries the designer to supply the system with the specifications of the slot's attributes. The scenario itself has no embedded heuristic knowledge. In this approach, the designer is entirely responsible for making the assumptions. In another approach, the scenario `solveSlotSpecs` can generate the assumptions by applying its own heuristic knowledge embedded in the rules. In that way, the example design system takes a more mechanical approach in which the designer only observes the behaviour of the system and intervenes when the obtained result is not satisfactory. Both approaches can be implemented in ADDL. There was no particular reason for choosing the first one.

10

Discussion

10.1 Introduction

This dissertation presents the development of ADDL throughout the last five years. The ordering of the chapter reflects a shift from a merely abstract model being presented in Chapter 2 towards a concrete system in Chapter 9. It is no coincidence that this ordering also reflects a chronological description of the research on ADDL. However, similar to the design process model it only reflects the research in retrospect. In practice, it was a process of constantly revising the language specifications in accordance with the changed demands. Chapter 2 is a rewrite of my first publication [Veerkamp, 1989]. Chapter 3 contains a worked-out example that appeared in [Veerkamp *et al.*, 1990; Xue *et al.*, 1990]. Chapter 4 is a complete revision of the first paper about ADDL's specifications (at that time it was called IDDL) written by the entire IIICAD group [Veth, 1987]. Chapter 5 and Chapter 6 deal with the specifications of the object-level language, that appear in a (now) obsolete form in [Veerkamp *et al.*, 1989; Veerkamp, Pieters Kwiers, and ten Hagen, 1991] Chapter 7 and Chapter 8 are entirely new and contain recently published material [Treur and Veerkamp, 1992]. Finally, Chapter 9 discusses a design system which solves the class of design problems introduced in Chapter 3 and in [Veerkamp *et al.*, 1990; Xue *et al.*, 1990].

The attentive reader has already noticed that the design maxims are not fully covered by the language specifications. Especially the multiple worlds, which play a dominant role in the model, seem to have disappeared from the specifications. Yet, this is not utterly true. During the development of ADDL it became clear that an explicit representation of control and process knowledge is a prerequisite before even thinking about a multi-world mechanism. Therefore, the locus of attention

has primarily been on the meta-level reasoning mechanisms. Until I have fully specified and implemented the meta-level language, the system cannot support a multi-world mechanism. Hence, the meta-level language signifies a milestone that must be reached before research on the multi-world mechanism can go on. Having reached this milestone, it is appropriate to look back and examine the current state of the art and compare the language specifications with the design maxims, which is discussed in § 10.2. A comparison of several competitive systems is presented in § 10.3 and § 10.4 gives some directions for future research. Finally, § 10.5 concludes this dissertation.

10.2 Achievements

This dissertation deals with the specification and implementation of a knowledge representation language for design. In particular, it focuses on the description and control of design processes. In the introduction of this dissertation (Chapter 1), I stated that a CAD system must be i) *intelligent*, ii) *interactive*, and iii) *integrated*. Obviously, these requirements are ambitiously chosen. I shall discuss them one by one.

As far as intelligence is concerned, ADDL meets the requirements by offering a conceptual framework for representing both design process and design object knowledge. The emphasis in the development of ICAD systems or AI systems in general, has traditionally been on the representation of object knowledge. From the beginning of my research, the importance of an explicit representation mechanism for process knowledge has been stressed. Originally, in ADDL there was no clear separation between object and process knowledge. Scenarios could both assert literal facts and activate other scenarios [Veerkamp *et al.*, 1989; Veerkamp, Pieters Kwiers, and ten Hagen, 1991]. Only recently, I made a strict distinction between object-level and meta-level scenarios and I introduced the process parameters.

The work of the IIICAD partners at the Vrije Universiteit has strongly influenced the decision to make a separation between domain and control knowledge. In [Brumsen, Pannekeet, and Treur, 1992], they present an argument for such a separation. Because of this rather late change, the current implementation has less thoroughly been used than previous versions. It may thus be that, for instance, the number of different process parameters is too small for adequately controlling the design process in large applications. However, the current framework is such that it is fairly easy to extend the functionality. Besides, a system with explicitly and separately represented process knowledge is more modular, and thus easier to develop, debug and modify (see also [van Harmelen, 1989] pp. 14-).

Concerning the interactiveness of ADDL very little effort has been spent on that aspect of ADDL, because it is a research topic on its own carried out by another member of the IIICAD project. Built-in predicates currently keep a simple direct

dialogue with the user. In the near future, this dialogue will be held with the user-interface. In my opinion, defining special purpose scenarios that control the dialogue with the designer seems to be a promising approach. These scenarios may have a different syntax and may use other constructs than the 'normal' scenarios. These scenarios can run concurrently with the actual design process using the multi-world mechanism. The control of the dialogue can then be specified using *Manifold*, a specification language for parallel processes [Arbab and Herman, 1991; Soede *et al.*, 1991] currently under development at CWI.

The issue of integration must either be dealt with within ADDL or outside the language. Chapter 9 shows an example of the integration of multiple aspect models with a central qualitative model within ADDL, viz. a kinematic and geometric aspect model. Besides, it mentions a geometric modeler that generates a wireframe model of the design object description. The latter is an example of external aspect model. The wireframe modeler is written in Smalltalk. The interface to the modeler is achieved by applying functions to an object of the prototype `geometricModel` (see §8.4.3). It turned out to be relatively easy (less than a week programming) to adapt and connect an existing wireframe modeler to ADDL. So far, I have not yet tried to connect ADDL to an application outside Smalltalk. Though I do not foresee any insurmountable difficulties.

The above three requirements are implicitly reflected by some of the design maxims of Chapter 4. However, it does not address which design maxims have been met and which have not. In the introduction of this chapter, I have have already discussed the absence of the multi-world mechanism from the specifications. This discussion is continued in §10.4.1. Thus **DM** 17 has not been fulfilled. Furthermore, the design maxim on the dynamic modification of an object's internal structure (**DM** 27) has not been worked out. Dynamic modification can only be achieved if there are more advanced user-interface constructs. Concerning the inheritance mechanism in ADDL (**DM** 29) there is no inheritance of attributes and operations in the current implementation though the type hierarchy *is* used for the query mechanism. The remaining design maxims can be found in the current implementation of ADDL.

10.3 Comparison

As a programming environment, ADDL can be regarded from different points of view. First of all, it can be compared with expert system tools or knowledge representation languages in general, in the sense that ADDL suits for implementing expert systems for design problems. A second comparison is with some meta-level architectures. Thirdly, a comparison between ADDL and existing CAD development tools seems natural, though there are few systems that are conceived on a sound basis. Lastly, IDDL as it has been implemented at the University of Tokyo seems to be an obvious candidate for comparison. Evidently, each of these systems lacks

some functionality or has some other drawback compared to ADDL, since otherwise there was no proper justification for developing ADDL at all.

One of the essential features of ADDL is its underlying model. ADDL is not yet another general purpose knowledge representation language. It is especially designed for representing design knowledge. The ADDL specifications are inspired by a descriptive model of the design process, that describes *how* design is done[12]. The analysis of the domain and task of the language guides the choice for an appropriate data model, an adequate knowledge representation and an explicit control regime. It is thus justified to assert that ADDL is a special purpose programming language suitable for describing all aspects of design. Especially, the use of a meta-level language for controlling the design process and the both elegant and robust integration of the object-oriented and logic programming paradigm have contributed a lot to ADDL. The following sections compare ADDL with other approaches.

10.3.1 Expert system tools

In [Richer, 1986], Richer describes a set of criteria for evaluating expert system software tools and uses these criteria to evaluate several currently available commercial tools. With pleasure, I use some of his criteria for comparing ADDL with the given tools. The presented tools are: S.1 [Hayes-Roth, 1984], ART (Automatic Reasoning Tool) [Williams, 1984], KEE (Knowledge Engineering Environment) [Intellicorp, 1984] and *Knowledge Craft* [Buday, 1986].

The system S.1 is a derivative of the EMYCIN system [Davis, Buchanan, and Shortliffe, 1977]. Production rules represent the embedded knowledge. Meta-level, algorithmic or procedural knowledge represented in control blocks direct the problem solving process. It is started by a top-level control block. With respect to meta-level architectures, S.1 uses production rules at the object-level and control blocks at the meta-level. S.1's application domain is restricted to diagnosis and structured selection problem-solving strategies. It lacks, compared to ADDL, i) the expressive power of predicate calculus, ii) the flexibility of object-oriented programming and iii) a declarative description of the control knowledge.

Both the KEE system and Knowledge Craft integrate several AI methodologies into a single system. They support frame-based knowledge representation, rule-based reasoning, logic representations and object-oriented programming. They can both be viewed as extended frame-based systems. The rule-based and logic systems are not fully integrated with the frame-based system. Prototypes in ADDL are similar to frames in the sense that they consist of attributes and operations.

---

[12] See Chapter 2 and [Finger and Dixon, 1989; Takeda *et al.*, 1990] for a thorough discussion on *descriptive*, *cognitive*, *prescriptive* and *computable* models of the design process.

However, the relationships among frames in those frame-based languages are represented by frames themselves whereas the relationships among ADDL objects are represented by literal facts. The latter provides a better separation between objects and relations. Furthermore, both KEE and Knowledge Craft lack support for an explicit representation of control knowledge.

ART can be viewed as an extended rule-based system to which frames, logic, and Lisp programming are added. On the one hand it behaves like a deductive database that consists of a set of propositions. It makes an explicit distinction among positive, negative and unknown facts. Goals patterns define the condition for which the state of the database can be modified to a new state. The goals are stored in a goal base. A strategy pattern can be defined to control the kind of inference to satisfy the current goal. Schemata allow the user to describe objects and classes of objects. Viewpoints allow for an assumption based truth maintenance very similar to de Kleer's system [De Kleer, 1986a]. Compared to ADDL, ART lacks a true integration of its components and it does not have an explicit declarative representation of control knowledge.

A general characteristic of the above expert system tools is the combination of AI tools that have shown successful in past applications. On the one hand, this combination provides a good environment for software development. But on the other hand, the different tools are often loosely integrated resulting in bulky systems that are hard to comprehend. Furthermore, if the application domain is complex, such as in design, the systems offer little guidance as to what components of the system to use for the representation of what kind of knowledge. This issue is discussed in Chapter 1 of [Jackson, Reichgelt, and van Harmelen, 1989].

10.3.2 Meta-level architectures

None of the evaluated tools (except S.1 that has some facilities to represent control knowledge procedurally) has a built-in meta-level architecture. A comparison may therefore look unfair as far as the issue of control is concerned. The obvious reason is that there are no commercial meta-level reasoning tools available. Certainly not when the IIICAD project started. Suitable candidates for comparison are HERACLES, Heuristic Classification Shell, [Clancey and Bock, 1988], the *Socrates* system [Jackson, Reichgelt, and van Harmelen, 1989], and DESIRE, a framework for DEsign and Specification of Interacting REasoning modules [Langevelde, Philipsen, and Treur, 1992]. Besides, [van Harmelen, 1989] gives a good evaluation and comparison of meta-level architectures.

HERACLES is a task-specific language based on NEOMYCIN. It is yet another derivative of the MYCIN system. In NEOMYCIN the (medical) domain knowledge and the strategic or control knowledge (diagnostic procedure) are expanded and represented separately and explicitly. HERACLES *groundwork* (a kind of expert

system shell) is obtained by extracting the domain knowledge from NEOMYCIN. Its control knowledge consists of metarules, tasks, and a task interpreter. Metarules consist of a premise and an action. Tasks consist of ordered sequences of metarules and additional knowledge about how the task interpreter should apply them. The premises of metarules examine the domain rules and relations and the problem-solving history. The actions involve the application of domain rules and the request and assertion of data. HERACLES and ADDL have a similar architecture though the former is merely designed for diagnosis while the latter is appropriate for design tasks. Compared to ADDL, HERACLES lacks an object-level that can reason explicitly and independently and it has a less declarative reading. Furthermore, HERACLES controls the interpretation of individual rules at the object-level while ADDL controls the interpretation of groups of rules either at the meta-level or at the object-level (meta-level or object-level scenarios).

Socrates is a logic-based general purpose knowledge representation system that has been developed at the University of Edinburgh. It is a pure meta-level inference system, which connotes that the behaviour of the object-level is fully specified at the meta-level. Applied to designing, the implication is that all decisions are taken at the meta-level and the object-level is merely used for asserting facts and assigning attributes. However, in designing a strategic decision often leaves room for multiple elaborations at the object-level. Therefore, since all alternative solutions are completely specified at the meta-level in a pure meta-level inference system, the need for a meta-meta-level may be raised. It is often more elegant to infer strategic decisions at the meta-level and to infer the solution dependent facts at the object-level. Hence, a mixed-level inference system is preferred above a pure meta-level inference system when design systems are concerned.

The DESIRE framework is based on a mixed-level inference system. Its development takes place at the Vrije Universiteit in Amsterdam in cooperation with the IIICAD project. ADDL and DESIRE are therefore based on common grounds though the emphasis may be placed differently. ADDL is more focused on the area of design while DESIRE is a more generic framework. A recent paper [Treur and Veerkamp, 1992] discusses the specification and implementation of a design problem in both languages. The following is a quote from this paper:

> [...] the framework DESIRE [...] can be used to design and formally specify complex reasoning tasks and compositional architectures of knowledge-based reasoning systems that perform these tasks. A complex task can often be modeled by decomposing it into a number of subtasks. These subtasks can be described as *components* or *modules*. Within DESIRE, precisely defined notions of a module and of interactions between modules are used. This provides uniformity of specifications of both modules and interactions, which can serve as standardized building blocks and interfaces among them. By combining such building blocks, a compositional architecture is obtained that models the

given complex reasoning task (see [Tan and Treur, 1991; Treur, 1989; Brumsen, Pannekeet, and Treur, 1992; Treur, 1991a]).

Compared to ADDL, DESIRE lacks the object-oriented paradigm but provides a more formal approach as an added value. Furthermore, DESIRE lacks features specific to designing which are reflected by both object-level and meta-level built-in predicate symbols. However, there is no impediment to add these features to DESIRE.

10.3.3 Intelligent CAD systems

Currently, there are no commercially available design knowledge representation languages that offer the same functionality as ADDL. An attempt to that end is the ICAD System [ICAD, 1986]. It is a system that claims to be a powerful, knowledge-based modeling environment in which engineers and designers represent the knowledge used to design and manufacture complex mechanical products. Unlike traditional CAD systems, the ICAD System does not only deal with geometric information of a product but also with the rules that determine the product. According to the company's brochure, product designers use ICAD's object-oriented language to create *product descriptions*. They define *rules* for all of the product's design parameters such as size, shape and orientation; specify how it connects to other parts; designate what material it is made of, etc. Designers can include manufacturing limitations, cost factors and other constraints in addition to mechanical design rules.

From the above specifications, it is evident that the ICAD System only deals with object-level knowledge. The system is an environment in which the knowledge *how* to model a product is represented. It lacks an explicit representation of which process steps to undertake to arrive at a product's description. In other words, it does not have a separate level at which the knowledge how to control the design process is represented.

10.3.4 An integrated data description language

The Integrated Data Description Language (IDDL) is a sibling of ADDL. It is currently under development at The University of Tokyo [Tomiyama, Xue, and Ishida, 1991]. Both languages originate from the same roots [Veth, 1987]. The major difference between the two languages is the use of two separate languages for the meta-level and the object-level in ADDL. Since recently, IDDL also distinguishes between *action-level* scenarios and object-level scenarios. However, both scenarios use the same language and there is no notion of reflection. IDDL uses modal operators for expressing uncertainties of facts and other design process information [Hughes and Cresswell, 1972].

Another difference is the procedural reading of IDDL. While ADDL has a purely declarative reading, IDDL uses built-in predicate symbols with a procedural

connotation. Especially at the action-level built-in predicate symbols with procedural names such as `use`, `select` and `do` are used to execute scenarios and to invoke an inference engine. Furthermore, the built-in predicate symbols `fail` and `succeed` also suggest an involved action. IDDL has in addition to a forward reasoning engine also the possibility to perform backward reasoning which cannot be done in ADDL. Both languages use an assumption-based truth maintenance system based on the original ideas of De Kleer's [De Kleer, 1986a] but without bookkeeping of each individual state. The complete model of De Kleer is ruined by its complexity.

10.4 Future directions

This dissertation reflects the present implementation of ADDL. There have been less detailed versions in the past and there will be more extended versions in the future. However, each of these versions have been and will be based on the design process model presented in Chapter 2. When I compare the present implementation with this model, then there is a number of issues that can be subject to future research.

10.4.1 Multi-world evaluation

The multi-world mechanism as being described in this section is still in development. It turned out that the primary requirements which ADDL must meet, are too ambitiously chosen to be fulfilled by a single person four-year project. A frame-work for a multi-world mechanism has been set up, but the actual design and implementation is a matter for future research.

In Chapter 2, I have presented a descriptive model that (partly) describes design as a process that develops different aspects of an artifact simultaneously. This notion reenters in Chapter 4 in terms of multiple scenarios being active at the same time. It is therefore evident that an approach to control the multi-world mechanism at the meta-level should be adopted. A goal meta-predicate symbol with two or more goal names causes the activation of multiple scenarios. Each scenario views its own world. Three kinds of such goal meta-predicate symbols can be distinguished, viz. `syncGoal`, `asyncGoal`, and `splitGoal`. I briefly discuss them in the following three sub-sections.

Synchronous multiple goals. The assertion of a meta-predicate symbol `syncGoal` causes the activation of *synchronous* multiple scenarios which are indicated by the arguments of the meta-predicate symbol. These scenarios are called synchronous, because the meta-level interpreter waits for each of them to be satisfied before it proceeds to the next rule. When a meta-atom `syncGoal(`g_1`, ···, `g_n`)` is asserted, the meta-level interpreter activates n scenarios concurrently, each with its own world. The names of the scenarios are determined

by $g_1 \cdots g_n$ and the worlds by their arguments. After termination of one of these scenarios, the interpreter checks whether other scenarios activated by the same goal statement are still active. When all the scenarios have terminated successfully, the synchronous multiple goal has been satisfied.

This mechanism allows the designer to generate some concurrent views on the fact-base. Each of these views may model a different, but related, aspect of the design object. These views are related because they have to terminate together. Each of the scenarios generates its own set of hypotheses which are specific to the aspect being modeled. The consistency of each individual set of hypotheses can easily be checked against the fact-base. The mutual consistency among the different sets of hypotheses is checked when each of the related scenarios has terminated successfully. The goal has been satisfied if they are indeed consistent with each other.

Asynchronous multiple goals. The concept of an *asynchronous* multiple goal is in principle the same as the concept of a synchronous one. Multiple related scenarios are activated by means of the meta-predicate symbol `asyncGoal`. The arguments of the meta-atom denote the names of the scenarios. The difference between synchronous and asynchronous goals lies in the condition on which the goal is satisfied. For satisfying an asynchronous multiple goal only one of the activated scenarios needs to terminate. The other activated scenarios may remain active as *background* processes whose termination has no influence on the course of the design process. Scenarios, which are active in the background, may act as monitors which control the interaction with the user interface, or which show a geometric model of the current state of the design object, and so on.

During the lifetime of a background scenario, the world it is viewing may become obsolete, i.e. it may represent an antiquated state of the sub-set of the fact-base, which it is viewing. Furthermore, it may be required to merge the set of hypotheses with the fact-base. The atomic built-in (object-level) predicate symbol `update` causes a recreation of a scenario's world and it merges its set of hypotheses with the fact-base. As usual, they are checked for consistency with the fact-base. This way, the outcome of a scenario may become known although the scenario may remain active.

Independent multiple goals. The two meta-predicate symbols, presented above, are concerned with multiple views on the same fact-base. Using the meta-predicate symbol `splitGoal`, multiple copies of the fact-base are generated by asserting an *independent* multiple goal. The scenarios activated by synchronous and asynchronous multiple goals operate on a single design object description. Each of the scenarios, which is activated by an independent multiple goal, operates on its own description of the design object. Therefore, they produce

different solutions to the design problem. A special category of meta-predicate symbols is needed to control this mechanism. For instance, it may enable the designer to discard a less promising solution, or to focus solely on a specific solution letting other possibilities sleep for the time being.

The issues concerned with the multi-world mechanism are subject to future research. The multi-world mechanism is an obvious candidate for inclusion in a next implementation. As shown above, the meta-level language is a suitable basis for controlling the multi-world mechanism. Special built-in predicates assert synchronous or asynchronous multiple goals causing the concurrent activation of multiple scenarios. This seems to be a promising approach, especially since it does not affect the declarative semantics of the meta-level language. However, research to whether such an implementation meets the required functionality needs still to be carried out.

10.4.2 The user interface

The integration of the multi-world mechanism with the user-interface is of extreme importance because the effectiveness of the mechanism depends heavily on the interaction with the designer. The designer must always be aware of what the system is doing and must always have full control over the system. This leads automatically to the second candidate for future research: the user-interface. The built-in predicates that now maintain a dialogue of questions and answers may not be sufficient. An approach of having special purpose scenarios that direct the dialogue may be promising. Such scenarios may view (next to the process and object information state) a *user information state.* It represents the data that have been provided by the designer.

10.4.3 Feature modeling

A last candidate for future research deals with the application of *features.* In CAD, features are a well known mechanism for representing manufacturing operations [Cunningham and Dixon, 1988]. For instance, the attachment of a leg to a tabletop can be described by a feature. In ADDL features can be represented by prototypes. However, a feature can be applied in different ways. Therefore, a feature prototype has a meta-level scenario and a set of object-level scenarios associated with it. The meta-level scenario evaluates the context in which the feature needs to be applied. Based on this context the meta-level scenario asserts a goal that is satisfied by one of the object-level scenarios. The selected scenario contains the (object) knowledge how to apply the feature with regard to the context. Part of the research on this topic has already been carried out by Jan Rogier. The realization of the scenario-based feature application as described above amounts to the integration of his feature modeler and ADDL.

10.5 Conclusions

This dissertation 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 some descriptive models that cover all stages. These models served as a source of inspiration to extract design maxims from for the development of ADDL. These design maxims represent the requirements that a programming language for implementing a knowledge-based CAD system must fulfill. They serve as a basis for the formal language specifications of ADDL.

A third version of ADDL is now operational, and a first experimental ICAD has been built. The result of the experiments will be used to evaluate the language specifications. It might be possible that they must be adjusted to adapt to the changed needs. For the time being I have succeeded in developing a knowledge representation language that has the facilities to i) represent a complex object structure, ii) encode object-level rules that can reason about this structure and iii) control the object-level reasoning by means of meta-level rules.

11

Bibliography

Aho and Ullman, 1977

Aho, A.V. and Ullman, J.D. (1977). *Principles of Compiler Design.* Addison-Welsey, Reading, MA, 1977.

Aiello and Levi, 1988

Aiello, L. and Levi, G. (1988). 'The Uses of Metaknowledge in AI Systems'. In P. Maes and D. Nardi (Eds.), *Meta-level Architectures and Reflection*, pages 243-254. North-Holland, Amsterdam, 1988.

Akman *et al.*, 1988

Akman, V., ten Hagen, P.J.W., Rogier, J., and Veerkamp, P.J. (1988). 'Knowledge Engineering in Design'. *Knowledge-Based Systems* **1** (2): 67-77, 1988.

Apt, 1990

Apt, K.R. (1990). 'Logic Programming'. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science.* Elsevier Science Publishers B.V., Amsterdam, 1990.

Arbab and Herman, 1991

Arbab, F. and Herman, I. (1991). 'Manifold: A Language for Specification of IPC'. *Proc. EurOpen Autumn Conference.* 127-144, Budapest, September 1991.

Arbab, 1991

Arbab, F. (1991). 'Design Object Representation'. In H. Yoshikawa, F. Arbab, and T. Tomiyama (Eds.), *Intelligent CAD, III*, pages 31-41. North-Holland, Amsterdam, 1991.

Blake and Cook, 1987

Blake, E. and Cook, S. (1987). 'On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk'. In J. Bezivin, J.-M. Hullot, P. Cointe, and H. Lieberman (Eds.), *ECOOP'87 European Conference on Object-Oriented Programming*, pages 41-50. Springer-Verlag, 1987.

Blamey, 1986

Blamey, S. (1986). 'Partial Logic'. In D. Gabbay and F. Guenthner (Eds.), *Handbook of Philosophical Logic*, Volume III, pages 1-70. Reidel, Dordrecht, 1986.

Bobrow, 1984

Bobrow, H.G. (1984). 'Qualitative Reasoning about Physical Systems: An Introduction'. *Artificial Intelligence* **24** (3): 1-6, 1984.

Brumsen, Pannekeet, and Treur, 1992

Brumsen, H.A., Pannekeet, J.H.M., and Treur, J. (1992). 'A Compositional Knowledge Based Architecture Modelling Process Aspects of Design Tasks'. *Proc. Twelfth International Conference on Artificial Intelligence and Expert Systems*: 283-293, Avignon, 1992.

Buday, 1986

Buday, R. (1986). 'Carnegie: Schooled in Expert Systems'. *Information Week*: 35-37, 1986.

Bylander and Chandrasekaran, 1987

Bylander, T. and Chandrasekaran, B. (1987). 'Generic Tasks for Knowledge-Based Reasoning: the Right Level of Abstraction for Knowledge Acquisition'. *Int. J. of Man-Machine Studies*: 231-244, 1987.

Clancey and Bock, 1988

Clancey, W.J. and Bock, C. (1988). 'Representing Control Knowledge as Abstract Tasks and Metarules'. In L. Bolc and M.J. Coombs (Eds.), *Expert System Applications*, pages 1-77. Springer-Verlag, Berlin, 1988.

Clark and Tärnlund, 1982

Clark, K.L. and Tärnlund, S.-A. (1982). *Logic Programming*. Academic Press, London, 1982.

Clocksin and Mellish, 1981

Clocksin, W.F. and Mellish, C.S. (1981). *Programming in Prolog*. Springer-Verlag, Berlin, 1981.

Cook, 1987

Cook, W. (1987). *Self-Referential Models of Inheritance*. Brown University Report, 1987.

Cunningham and Dixon, 1988

Cunningham, J.J. and Dixon, J.R. (1988). 'Designing with Features: The Origin of Features'. In *proceedings of ASME Computers in Engineering 1988*, pages

237-243, 1988.

Davis, Buchanan, and Shortliffe, 1977

Davis, R., Buchanan, B., and Shortliffe, E. (1977). 'Production Rules as a Representation for a Knowledge-Based Consultation Program'. *Artificial Intelligence* **8**: 15-45, 1977.

Davis and King, 1977

Davis, R. and King, J. (1977). 'An Overview of Production Systems'. In E.W. Elcock and D. Michie (Eds.), *Machine Intelligence 8*, pages 300-332. Ellis Horwood Ltd., Chichester, 1977.

De Kleer, 1986a

De Kleer, J. (1986a). 'An Assumption Based TMS'. *Artificial Intelligence* **28**: 127-162, 1986.

De Kleer, 1986b

De Kleer, J. (1986b). 'Problem Solving with the ATMS'. *Artificial Intelligence* **28**: 197-224, 1986.

Finger and Dixon, 1989

Finger, S. and Dixon, J.R. (1989). 'A Review of Research in Mechanical Engineering Design. Part I: Descriptive, Prescriptive, and Computer-Based Models of Design Processes'. *Research in Engineering Design* **1** (1): 51-67, 1989.

Gero, 1990

Gero, J.S. (1990). 'Design Prototypes: A Knowledge Representation Schema for Design'. *AI Magazine* **11** (4): 26-36, 1990.

Goldberg and Robson, 1983

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

Goldberg, 1984

Goldberg, A. (1984). *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA, 1984.

Hayes, 1979

Hayes, P. (1979). 'The logic of frames'. In D. Metzing (Ed.), *Frame Conceptions and Text Understanding*, pages 46-61. Walter de Gruyter and Co., Berlin, 1979.

Hayes-Roth, 1984

Hayes-Roth, F. (1984). 'The Industrialization of Knowledge Engineering'. In W. Reitman (Ed.), *Artificial Intelligence Applications in Business.* Ablex, Norwood, NJ, 1984.

Hubka, 1987

Hubka, V. (1987). *Principles of Engineering Design.* Springer-Verlag, Berlin, 1987.

Hughes and Cresswell, 1972

Hughes, G.E. and Cresswell, M.J. (1972). *An Introduction to Modal Logic.* Methuen and Co. Ltd., London, 1972.

ICAD, 1986

ICAD, Inc. (1986). *The ICAD System: A New Way to Capture Design and Manufacturing Knowledge.* ICAD Company Brochure, Cambridge, MA, 1986.

Intellicorp, 1984

Intellicorp (1984). *The Knowledge Engineering Environment.* Intellicorp Company Brochure, Mountain View, CA, 1984.

Jackson, Reichgelt, and van Harmelen, 1989

Jackson, P., Reichgelt, H., and van Harmelen, F. (1989). *Logic-Based Knowledge Representation.* The MIT Press, Cambridge, MA, 1989.

Johnson, 1986

Johnson, S.C. (1986). 'Yacc: Yet Another Compiler Compiler'. *Unix Programmer's Manual, Supplementary Documents 1*: PS1:15, 1986.

Kernighan and Ritchie, 1978

Kernighan, B.W. and Ritchie, D.M. (1978). *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

Kleene, 1956

Kleene, S.C. (1956). 'Representation of Events in Nerve Nets'. In C.E. Shannon and J. McCarthy (Eds.), *Automata Studies*, pages 3-40. Princeton University Press, 1956.

Kowalczyk and Treur, 1990

Kowalczyk, W. and Treur, J. (1990). 'On the use of a formalized generic task model in knowledge acquisition'. In Wielinga, B.J., Boose, J., Gaines, B., Schreiber, G., and Someren, M. van (Eds.), *Current trends in knowledge acquisition*, pages 198-221. IOS Press, 1990.

Kumagai, 1976

Kumagai, S. (1976). *300 Selections of Automated Mechanisms.* Nikkan Industrial Newspaper, Tokyo, 1976.

Langevelde and Treur, 1991

Langevelde, I.A. van and Treur, J. (1991). *Tackling the Incompleteness of Chaining.* Report IR-274, Dept. of Math. and Comp. Sc., Vrije Universiteit Amsterdam, 1991.

Langevelde, Philipsen, and Treur, 1992

Langevelde, I.A. van, Philipsen, A.W., and Treur, J. (1992). 'Formal Specification of Compositional Architectures'. *Proc. European Conference on Artificial Intelligence, ECAI'92*, 1992.

Lesk and Schmidt, 1986

Lesk, M.E. and Schmidt, E. (1986). 'Lex – A Lexical Analyzer Generator'. *Unix Programmer's Manual, Supplementary Documents 1*: PS1:16, 1986.

Lieberman, 1986

Lieberman, H. (1986). 'Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Languages'. *OOPSLA '86, Special Issue of SIGPLAN Notices Notices* **21** (11), 1986.

Lloyd, 1987

Lloyd, J.W. (1987). *Foundations of Logic Programming.* Second, Extended edition, Springer-Verlag, Berlin, 1987.

Martelli and Montanari, 1982

Martelli, A. and Montanari, U. (1982). 'An Efficient Unification Algorithm'. *ACM Trans. on Prog. Lang. and Systems* **4** (2): 258-282, 1982.

Maurice, 1991

Maurice, I. (1991). *Gebruik van een ontwerpsysteem; meer grip op het ontwerpen?.* Stage verslag, Centrum voor Wiskunde en Informatica, Amsterdam, 1991. (in Dutch.)

Megens, 1987

Megens, M. (1987). *An Implementation of a Simple Design Description Language.* MSc. thesis, Centre for Mathematics and Computer Science, Amsterdam, 1987.

Meyer, 1988

Meyer, B. (1988). *Object-oriented Software Construction.* Prentice Hall, Hertfordshire, 1988.

Minker, 1988

Minker, J. (1988). *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, Inc., Lost Altos, CA, 1988.

Mostow, 1985

Mostow, J. (1985). 'Toward Better Models of the Design Process'. *AI Magazine* **6** (1): 44-57, 1985.

Pahl and Beitz, 1988

Pahl, G. and Beitz, W. (1988). *Engineering Design – A Systematic Approach.* Springer-Verlag, Berlin, 1988.

Paterson and Wegman, 1978

Paterson, M.S. and Wegman, M.N. (1978). 'Linear Unification'. *Journal of Computer and System Sciences* **16** (2): 158-167, 1978.

Perlis, 1988

Perlis, D. (1988). 'Meta in Logic'. In P. Maes and D. Nardi (Eds.), *Meta-level Architectures and Reflection*, pages 37-49. North-Holland, Amsterdam, 1988.

Reiter, 1978

Reiter, R. (1978). 'On Closed-World Databases'. In H. Gallaire and J. Minker (Eds.), *Logic and Data Bases*, pages 55-76. Plenum Press, New York, 1978.

Richer, 1986

Richer, M.H. (1986). 'An Evaluation of Expert System Development Tools'. *Expert Systems* **3** (3): 166-183, 1986.

Rogier, 1989

Rogier, J. (1989). 'The BiCad System: An Intelligent Product Modelling System for Architectural Design'. In Varol Akman, P.J.W. ten Hagen, and P.J. Veerkamp (Eds.), *Intelligent CAD Systems II – Implementational Issues*, pages 291-310. Springer-Verlag, Berlin, 1989.

Rogier, Veerkamp, and ten Hagen, 1989

Rogier, J., Veerkamp, P.J., and ten Hagen, P.J.W. (1989). 'An Environment for Knowledge Representation for Design'. In *Proc. Civil Engineering Expert Systems*, Madrid, 1989.

Rogier, 1991

Rogier, J. (1991). 'A Component Class for Design Objects'. In P.J.W. ten Hagen and P.J. Veerkamp (Eds.), *Intelligent CAD Systems III – Practical Experience and Evaluation*, pages 41-60. Springer-Verlag, Berlin, 1991.

Rosenbloom, Laird, and Newell, 1988

Rosenbloom, P., Laird, J., and Newell, A. (1988). 'Meta-Levels in SOAR'. In P. Maes and D. Nardi (Eds.), *Meta-level Architectures and Reflection*, pages 227-240. North-Holland, Amsterdam, 1988.

Rumbaugh *et al.*, 1991

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

Soede *et al.*, 1991

Soede, D., Arbab, F., Herman, I., and ten Hagen, P.J.W. (1991). 'The GKS Input Model in Manifold'. *Computer Graphics Forum* **10**: 209-224, North-Holland, 1991.

Sterling, 1988

Sterling, L. (1988). 'A Meta-Level Architecture for Expert Systems'. In P. Maes and D. Nardi (Eds.), *Meta-level Architectures and Reflection*, pages 301-311. North-Holland, Amsterdam, 1988.

Stroustrup, 1986

Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

Takala, 1987a

Takala, T. (1987a). 'Theoretical Framework for Computer Aided Innovative Design'. In H. Yoshikawa and E.A. Warman (Eds.), *Proc. IFIP WG 5.2 Working Conference on Design Theory for CAD*, pages 323-338. North-Holland, Amsterdam, 1987.

Takala, 1987b

Takala, T. (1987b). 'Intelligence beyond Expert Systems: A Physiological Model with Applications in Design'. In P.J.W. ten Hagen and T. Tomiyama (Eds.), *Intelligent CAD Systems I – Theoretical and Methodological Aspects*, pages 286-294. Springer-Verlag, Berlin, 1987.

Takeda *et al.*, 1990

Takeda, H., Veerkamp, P.J., Tomiyama, T., and Yoshikawa, H. (1990). 'Modeling Design Processes'. *AI Magazine* **11** (4): 37-48, 1990.

Tan and Treur, 1991

Tan, Y.H. and Treur, J. (1991). 'A Bi-modular Approach to Nonmonotonic Reasoning'. *Proc. World Congress on Fundamentals of Artificial Intelligence, WOCFAI-91*: 461-476, 1991.

Tan, 1992

Tan, Y.H. (1992). *Non-Monotonic Reasoning: Logical Architecture and Philosophical Applications.* Ph.D. Thesis, Vrije Universiteit Amsterdam, 1992.

Tomiyama and Yoshikawa, 1987

Tomiyama, T. and Yoshikawa, H. (1987). 'Extended General Design Theory'. In H. Yoshikawa and E.A. Warman (Eds.), *Proc. IFIP WG 5.2 Working Conference on Design Theory for CAD*, pages 95-125. North-Holland, Amsterdam, 1987.

Tomiyama and ten Hagen, 1987

Tomiyama, T. and ten Hagen, P.J.W. (1987). *The Concept of Intelligent Integrated Interactive CAD Systems.* CWI-Report CS-R8717, 1987.

Tomiyama, Xue, and Ishida, 1991

Tomiyama, T., Xue, D., and Ishida, Y. (1991). 'An Experience with Developing a Design Knowledge Representation Language'. In P.J.W. ten Hagen and P.J. Veerkamp (Eds.), *Intelligent CAD Systems III – Practical Experience and Evaluation*, pages 131-154. Springer-Verlag, Berlin, 1991.

Treur, 1989

Treur, J. (1989). 'A Logical Analysis of Design Tasks for Expert Systems'. *International Journal of Expert Systems* **2**: 233-253, 1989.

Treur, 1991a

Treur, J. (1991a). 'Interaction types and chemistry of generic task models'. *Proc. European Knowledge Acquisition Workshop, EKAW-91*, 1991.

Treur, 1991b

Treur, J. (1991b). 'On the Use of Reflection Principles in Modelling Complex Reasoning'. *International Journal of Intelligent Systems* **6**: 277-294, 1991.

Treur, 1991c

Treur, J. (1991c). 'A Logical Framework for Design Processes'. In P.J.W. ten Hagen and P.J. Veerkamp (Eds.), *Intelligent CAD Systems III – Practical Experience and Evaluation*, pages 3-20. Springer-Verlag, Berlin, 1991.

Treur and Veerkamp, 1992

Treur, J. and Veerkamp, P.J. (1992). 'Explicit Representation of Design Process Knowledge'. In J.S. Gero (Ed.), *Proc. of 2nd Int. Conf. on Artificial Intelligence in Design, AID'92*. Kluwer Academic Publishers, 1992.

Turner, 1984

Turner, R. (1984). *Logics for Artificial Intelligence*. Ellis Horwood, Inc., 1984.

van Dalen, 1985

van Dalen, D. (1985). *Logic and Structure, 2nd edition*. Springer-Verlag, Heidelberg, 1985.

van Harmelen, 1989

van Harmelen, F. (1989). 'A Classification of Meta-Level Architectures'. In P. Jackson, H. Reichgelt, and F. van Harmelen (Eds.), *Logic-Based Knowledge Representation*, pages 13-35. The MIT Press, Cambridge, MA, 1989.

van Klarenbosch, 1991

van Klarenbosch, H.E. (1991). *A Task-Based Intelligent User Interface*. CWI Report CS-R91??, Amsterdam, 1991.

Veerkamp *et al.*, 1989

Veerkamp, P.J., Akman, V., Bernus, P., and ten Hagen, P.J.W. (1989). 'IDDL: A Language for Intelligent Interactive Integrated CAD Systems'. In V. Akman, P.J.W ten Hagen, and P.J. Veerkamp (Eds.), *Intelligent CAD Systems II – Implementational Issues*, pages 58-74. Springer-Verlag, Berlin, 1989.

Veerkamp, 1989

Veerkamp, P.J. (1989). 'Multiple Worlds in an Intelligent CAD System'. In H. Yoshikawa and D. Gossard (Eds.), *Intelligent CAD, I*, pages 77-88. North-Holland, Amsterdam, 1989.

Veerkamp *et al.*, 1990

Veerkamp, P.J., Kiriyama, T., Xue, D., and Tomiyama, T. (1990). 'Representation and Implementation of Design Knowledge for Intelligent CAD – Theoretical Aspects'. *Proc. Fourth Eurographics Workshop on Intelligent CAD – The Added Value of Intelligence*, Compiegne, 1990.

Veerkamp, Pieters Kwiers, and ten Hagen, 1991

Veerkamp, P.J., Pieters Kwiers, R.S.S., and ten Hagen, P.J.W. (1991). 'Design Process Representation in ADDL'. In P.J.W. ten Hagen and P.J. Veerkamp

(Eds.), *Intelligent CAD Systems III – Practical Experience and Evaluation*, pages 155-168. Springer-Verlag, Berlin, 1991.

Veerkamp and ten Hagen, 1991

Veerkamp, P.J. and ten Hagen, P.J.W. (1991). 'Qualitative Reasoning about Design Objects'. *Preprints of the 5th International Conference on the Manufacturing Science and Technology of the Future*, Enschede, The Netherlands, June 11-13, 1991.

Veth, 1987

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

Waldron, 1991

Waldron, M.B. (1991). 'Design Processes and Intelligent Computer-Aided Design (CAD)'. In H. Yoshikawa, F. Arbab, and T. Tomiyama (Eds.), *Intelligent CAD, III*, pages 51-75. North-Holland, Amsterdam, 1991.

Wegner, 1990

Wegner, P. (1990). 'Concepts and Paradigms of Object-Oriented Programming'. *OOPS Messenger* **1** (1): 7-87, 1990.

Weyhrauch, 1980

Weyhrauch, R.W. (1980). 'Prolegomena to a Theory of Mechanized Formal Reasoning'. *Artificial Intelligence* **13**: 133-170, 1980.

Williams, 1984

Williams, C. (1984). *ART The Advanced Reasoning Tools – Conceptual Overview*. Inference Corporation, 1984.

Xue *et al.*, 1990

Xue, D., Kiriyama, T., Veerkamp, P.J., and Tomiyama, T. (1990). 'Representation and Implementation of Design Knowledge for Intelligent CAD – Implementational Aspects'. *Proc. Fourth Eurographics Workshop on Intelligent CAD – The Added Value of Intelligence*, Compiegne, 1990.

Yoshikawa, 1981

Yoshikawa, H. (1981). 'General Design Theory and a CAD System'. In H. Yoshikawa, T. Sata, and E.A. Warman (Eds.), *Man-Machine Communication in CAD/CAM, Proc. IFIP WG 5.2 Working Conference*, pages 35-. North-Holland, Amsterdam, 1981.

I

Lexical Analyzer

I.1 Transition diagram

The complete transition diagram of the lexical analyzer is depicted in Fig. I.1. For convenience, the states directly following state `0` are numbered `1-9a-n`. Thus, the methods **`lex2`** and **`lex3`** given in Chapter 8 are actually called **`lex4`** and **`lex41`**.

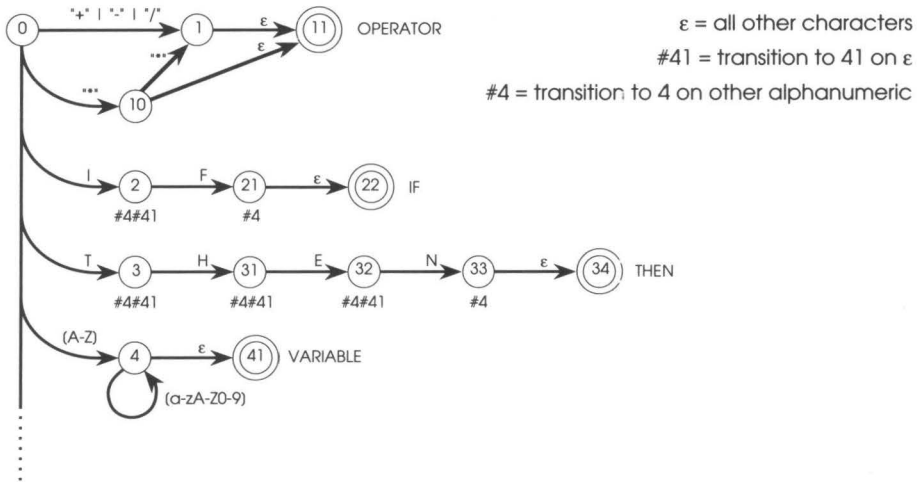


Fig. I.1 Transition diagram for all tokens appearing in ADDL.

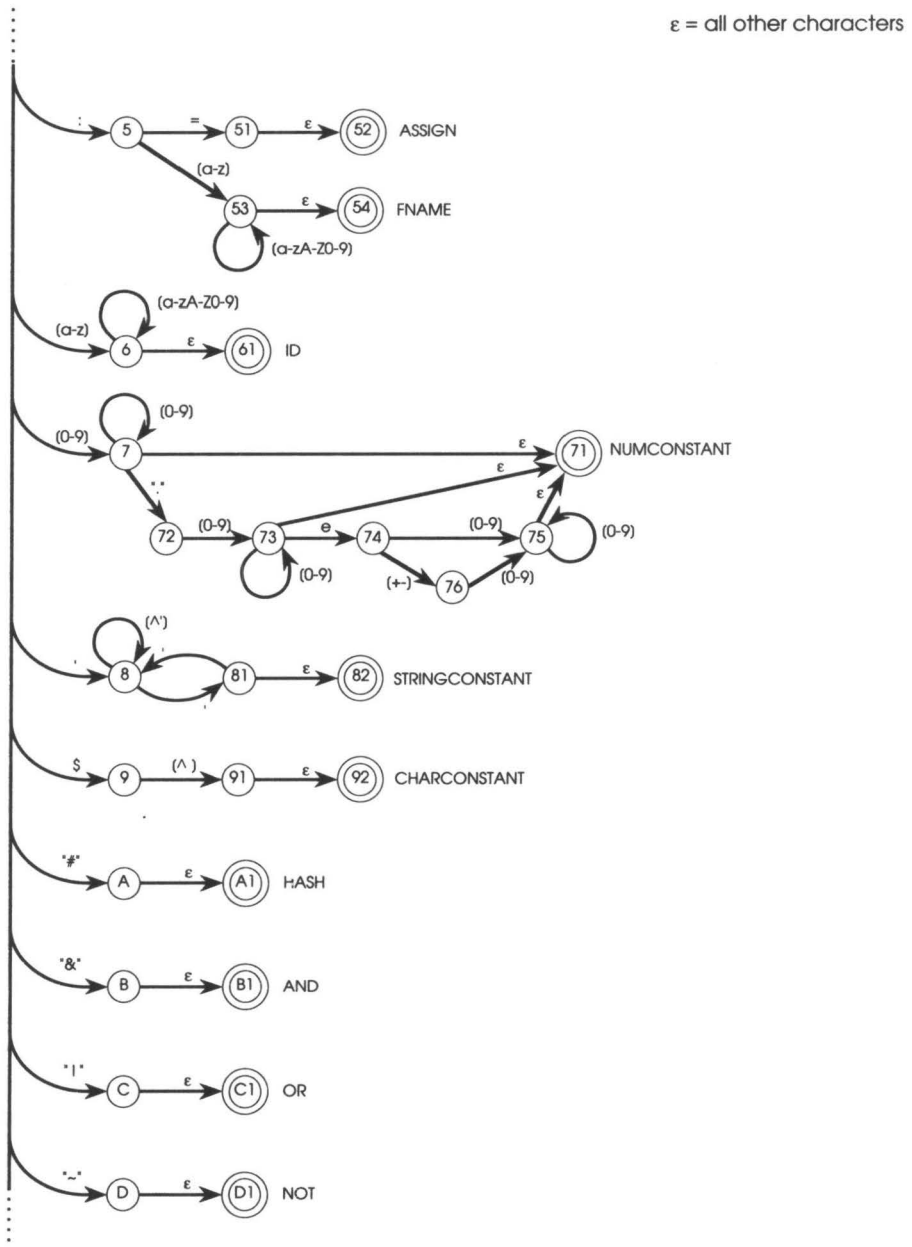


Fig. I.1 Transition diagram **(continued)**.

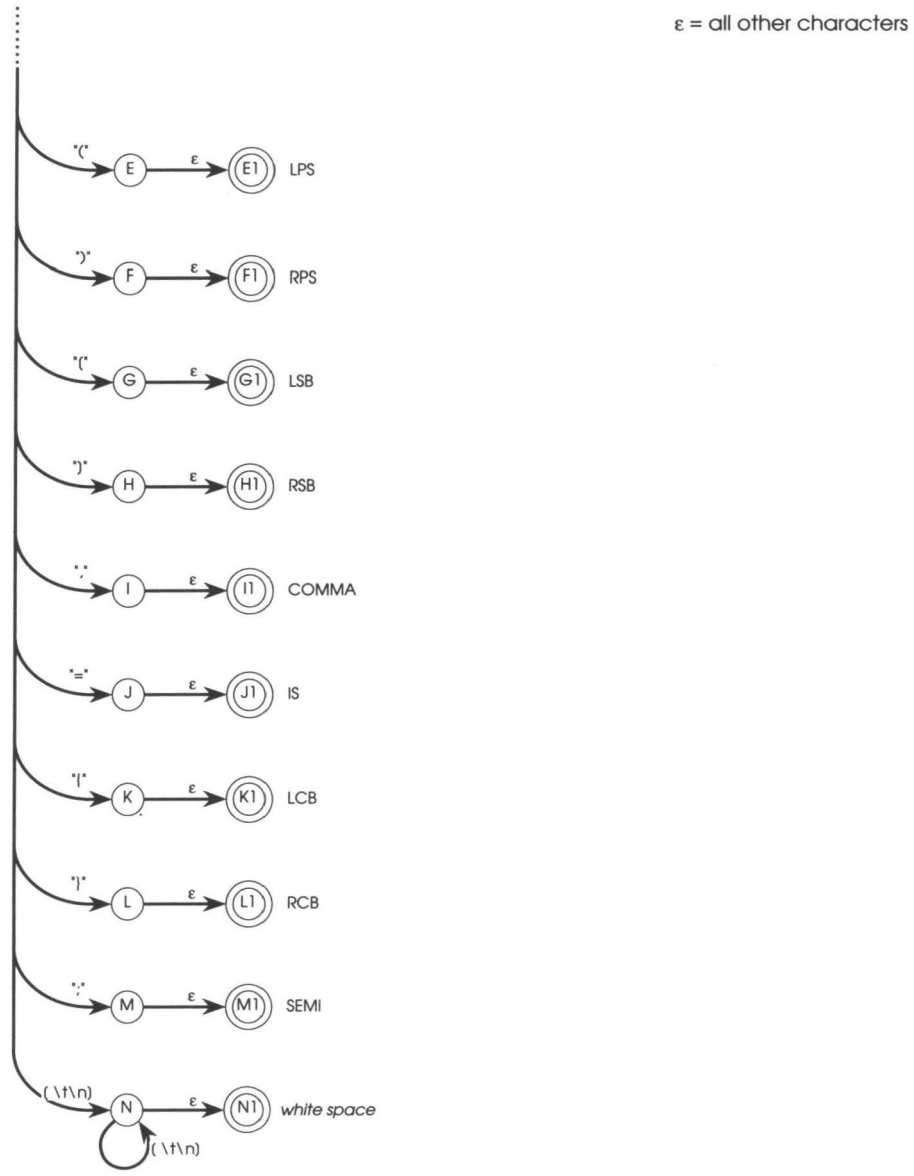


Fig. I.1 Transition diagram **(continued).**

II

ADDL Definitions for Yacc

II.1 Definition of object-level scenarios

The definition of object-level scenarios is given by the following grammer rules:

```
%start scenario
%token IF THEN VARIABLE AND OR NOT FNAME ID
%token NUMCONSTANT STRINGCONSTANT CHARCONSTANT
%token HASH LPS RPS LSB RSB COMMA
%right NOT
%left AND OR OPERATOR
%%
scenario    :  /* 1*/ head rules
            ;
head        :  /* 2*/ ID LPS ID RPS
            |  /* 3*/ ID
            ;
rules       :  /* 4*/ rules rule
            |  /* 5*/ rule
            ;
rule        :  /* 6*/ IF antecedent THEN consequent
            ;
antecedent  :  /* 7*/ atom
            |  /* 8*/ NOT atom
            |  /* 9*/ antecedent AND antecedent
            |  /*10*/ antecedent OR antecedent
            |  /*11*/ LPS antecedent RPS
            ;
consequent  :  /*12*/ atom
            |  /*13*/ NOT atom
            |  /*14*/ consequent AND consequent
            |  /*15*/ LPS consequent RPS
            ;
```

```
atom        :  /*16*/ ID LPS termlist RPS
            |  /*17*/ ID
            ;
termlist    :  /*18*/ termlist COMMA term
            |  /*19*/ term
            ;
term        :  /*20*/ simpleterm
            |  /*21*/ function
            |  /*22*/ simpleterm OPERATOR simpleterm
            ;
function    :  /*23*/ simpleterm FNAME
            |  /*24*/ simpleterm FNAME LSB arglist RSB
            ;
arglist     :  /*25*/ arglist COMMA simpleterm
            |  /*26*/ simpleterm
            ;
simpleterm  :  /*27*/ VARIABLE
            |  /*28*/ ID
            |  /*29*/ NUMCONSTANT
            |  /*30*/ STRINGCONSTANT
            |  /*31*/ CHARCONSTANT
            |  /*32*/ HASH LPS elements RPS
            ;
elements    :  /*33*/ elements simpleterm
            |  /*34*/ simpleterm
            ;
```

II.2 Definition of meta-level scenarios

The definition of meta-level scenarios is given by the following grammer rules:

```
%start scenario
%token IF THEN VARIABLE AND OR NOT FNAME ID
%token NUMCONSTANT STRINGCONSTANT CHARCONSTANT
%token HASH LPS RPS LSB RSB COMMA
%right NOT
%left AND OR OPERATOR
%%
scenario    :  /* 1*/ head rules
            ;
head        :  /* 2*/ ID OPERATOR ID LPS ID RPS
        |  /* 3*/ ID OPERATOR ID

rules       :  /* 4*/ rules rule
            |  /* 5*/ rule
            ;
rule        :  /* 6*/ IF antecedent THEN consequent
            ;
antecedent  :  /* 7*/ atom
            |  /* 8*/ NOT atom
            |  /* 9*/ antecedent AND antecedent
            |  /*10*/ antecedent OR antecedent
            |  /*11*/ LPS antecedent RPS
            ;
```

```
consequent  :  /*12*/ atom
            |  /*13*/ NOT atom
            |  /*14*/ consequent AND consequent
            |  /*15*/ LPS consequent RPS
            ;
atom        :  /*16*/ ID LPS constant RPS
            |  /*17*/ ID
            ;
constant    :  /*18*/ ID LPS termlist RPS
            |  /*19*/ ID
            ;
termlist    :  /*20*/ termlist COMMA term
            |  /*21*/ term
            ;
term        :  /*22*/ simpleterm
            |  /*23*/ function
            |  /*24*/ simpleterm OPERATOR simpleterm
            ;
function    :  /*25*/ simpleterm FNAME
            |  /*26*/ simpleterm FNAME LSB arglist RSB
            ;
arglist     :  /*27*/ arglist COMMA simpleterm
            |  /*28*/ simpleterm
            ;
simpleterm  :  /*29*/ VARIABLE
            |  /*30*/ ID
            |  /*31*/ NUMCONSTANT
            |  /*32*/ STRINGCONSTANT
            |  /*33*/ CHARCONSTANT
            |  /*34*/ HASH LPS elements RPS
            ;
elements    :  /*35*/ elements simpleterm
            |  /*36*/ simpleterm
            ;
```

II.3 Definition of local operations

The definition of local operations is given by the following grammer rules:

```
%start operations
%token FNAME VARIABLE IS ID OPERATOR
%token NUMCONSTANT STRINGCONSTANT CHARCONSTANT
%token HASH LPS RPS LSB RSB LCB RCB ASSIGN COMMA SEMI
%left OPERATOR
%%
operations  :  /* 1*/  operations operation
            |  /* 2*/  operation
            ;
operation   :  /* 3*/  FNAME IS body
            |  /* 4*/  FNAME LSB varlist RSB IS body
            ;
body        :  /* 5*/  LCB statements expression RCB
            ;
statements  :  /* 6*/  statement SEMI statements
```

```
            |  /* 7 statements may be empty*/
            ;
statement   :  /* 8*/  VARIABLE ASSIGN expression
            ;
expression  :  /* 9*/  simpexpr
            |  /*10*/  function
            |  /*11*/  expression OPERATOR expression
            |  /*12*/  LPS expression RPS
            ;
function    :  /*13*/  simpexpr FNAME
            |  /*14*/  simpexpr FNAME LSB arglist RSB
            ;
arglist     :  /*15*/  arglist COMMA simpexpr
            |  /*16*/  simpexpr
            ;
simpexpr    :  /*17*/  VARIABLE
            |  /*18*/  ID
            |  /*19*/  NUMCONSTANT
            |  /*20*/  STRINGCONSTANT
            |  /*21*/  CHARCONSTANT
            |  /*22*/  HASH LPS elements RPS
            ;
elements    :  /*23*/  elements simpexpr
            |  /*24*/  simpexpr
            ;
varlist     :  /*25*/   varlist COMMA VARIABLE
            |  /*26*/   VARIABLE
```

III

Signatures of the Example Design System

III.1 Maximum signature of the process information state

Σ(processinformationstate)

Type	Notation
objectAtom	A
composite	C
slot	SL

Meta-constant	Type
linearMotion, motionMetaModel, guideSpecs, guideGeometry, motionQualities, guideLimits, motionFault, guideRefinement, motionSpecs, slotSpecs, slotGeometry, angleOfFaces, slotWireframe, slotLimits, shaftLimits, slotRefinement	A
slot1	SL

Meta-function	Type
isSlot	C→A

Meta-predicate	Type
goal, success, abstract, concrete, exact	A

III.2 Maximum signature of the fact-base

Σ(fact-base)

Type	Notation
composite	C
number	NU
lever	LE
pivot	PV
motion	MN
guide	GU
slot	SL
objectInMotion	OM
pin	PI
point	PT
face	FA

Constant	Type
20	NU
lever1	LE
pivot1	PV
slot1	SL
pin1	PI
point1, point2	PT
face1, face2, face3, face4, face5, face6, face7, face8, face9	FA

Predicate	Type
isLever, isPivot, isSlot, isPin, isPoint, isFace	C
leftShift, rightShift	NU
value	NU×NU
hasPart	C×C
adjacent	FA×FA
linearMotion	OM×PT×PT
limitArrangement	OM×GU×PT
contact	FA×FA×PT
startPosition, endPosition	PT
slotRefinement, motionSpecs, slotSpecs, slotGeometry, angleOfFaces, motionQualities, slotLimits, motionFault, motionMetaModel	nil

III.3 Signatures of the meta-level scenarios

Σ(solveLinearMotion)

Type	Notation
composite	C
objectAtom	A
Meta-constant	**Type**
motionMetaModel, guideSpecs, guideGeometry, motionQualities, guideLimits, motionFault, guideRefinement, linearMotion	A
Meta-function	**Type**
isGuide	C→A
Meta-predicate	**Type**
unknown, abstract, concrete, exact, goal, success	A

Σ(solveGuideSpecs)

Type	Notation
objectAtom	A
guide, slot, shaft	C
Meta-constant	**Type**
motionSpecs, slotSpecs, shaftSpecs, guideSpecs	A
Meta-function	**Type**
isSlot, isShaft, isGuide	C→A
Meta-predicate	**Type**
abstract, positive, goal, success	A

Σ(solveGuideGeometry)

Type	Notation
composite	C
objectAtom	A
Meta-constant	**Type**
slotGeometry, shaftGeometry, angleOfFaces, slotWireframe, shaftWireframe, guideGeometry	A

Meta-function	**Type**
isSlot, isShaft, isGuide	C→A

Meta-predicate	**Type**
positive, goal, success, concrete	A

Σ(solveGuideLimits)

Type	**Notation**
composite	C
objectAtom	A
number	NU
point	PT

Meta-constant	**Type**
slotLimits, shaftLimits, guideLimits	A

Meta-function	**Type**
isSlot, isShaft, isGuide	C→A
startPosition, endPosition	PT→A
smallerEqual, greaterEqual	NU×NU→A

Meta-predicate	**Type**
positive, goal, success, exact	A

Σ(solveGuideRefinement)

Type	**Notation**
composite	C
objectAtom	A

Meta-constant	**Type**
guideRefinement	A

Meta-function	**Type**
isSlot	SL→A
isShaft	SH→A

Meta-predicate	**Type**
positive, goal, success	A

III.4 Signatures of the object-level scenarios

Σ(solveMotionMetaModel)

Type	Notation
composite	C
type	TY
guide	GU
objectInMotion	OM
point	PT

Constant	Type
slot, pin, shaft, slider, point	TY

Predicate	Type
isLever, isSliderDevice, isMotion, isGuide, isSlot, isShaft, isObjectInMotion, isPin, isSlider, isPoint	C
typeFor	C×TY
hasPart	C×C
linearMotion	OM×PT×PT
limitArrangement	OM×GU×PT
motionMetaModel	nil

Σ(solveMotionSpecs)

Type	Notation
motion	MN
number	NU
string	ST
composite	C

Constant	Type
'start of motion', 'end of motion'	ST

Function	Type
:start, :end, :halfWidth, :innerRange, :startWidth, :motionRange	MN→NU

Predicate	Type
isMotion	C
isNil, notNil	NU
uiNumber	ST×NU×NU×NU
value	NU×NU
motionSpecs	nil

Σ(solveSlotSpecs)

Type	Notation
lever	LE
slot	SL
pin	PI
number	NU
string	ST
composite	C

Constant	Type
'position of slot', 'length of slot', 'width of slot', 'diameter of pin'	ST
10	NU

Function	Type
:position, :length, :width	C→NU
:diameter	PI→NU
:maxPosition, :widthMinusTol	LE→NU
:maxLength	SL×LE→NU

Predicate	Type
isLever, isSlot, isPin	C
isNil, notNil	NU
uiNumber	ST×NU×NU×NU
value	NU×NU
slotSpecs	nil

Σ(solveSlotGeometry)

Type	Notation
slot	SL
face	FA
number	NU
type	TY
composite	C

Constant	Type
face	TY

Function	Type
:position, :halfWidthUp, :halfWidthDown, :posLength	SL→NU
:x, :y	FA→NU

Predicate	Type
isSlot, isPin, isFace	C
typeFor	C→TY
hasPart	C×C
adjacent	FA×FA
value	NU×NU
slotGeometry	nil

Σ(solveAngleOfFaces)

Type	Notation
face	FA
number	NU
composite	C

Function	Type
:angle	FA→NU
:angle	FA×FA→NU

Predicate	Type
isFace	C
isNil	NU
adjacent	FA×FA
value	NU×NU
angleOfFaces	nil

Σ(solveMotionQualities)

Type	Notation
objectInMotion	OM
guide	GU
point	PT
face	FA
number	NU
composite	C

Constant	Type
270, 90	NU

Function	Type
:angle	FA→NU

Predicate	Type
limitArrangement	OM×GU×PT
linearMotion	OM×PT×PT
isFace	C
hasPart	C×C
equal	NU×NU
contact	FA×FA×PT
startPosition, endPosition	PT
motionQualities	nil

Σ(solveSlotLimits)

Type	Notation
point	PT
pin	PI
face	FA
number	NU
composite	C

Function	Type
:x, :y	PT→NU
:diameter	PI→NU
:startPoint, :endPoint	FA×PI→NU

Constant	Type
0, 2	NU

Predicate	Type
isPin	C
contact	FA×FA×PT
startPosition, endPosition	PT
value	NU×NU
isNil, notNil	NU
slotLimits	nil

Σ(solveMotionFault)

Type	Notation
motion	MN
point	PT
number	NU
composite	C

Function	Type
:start, :end	MN→NU
:x	PT→NU
:leftMinus, :rightMinus	PT×MN→NU

Predicate	Type
isMotion	C
startPosition, endPosition	PT
greater, smaller	NU×NU
leftShift, rightShift	NU
motionFault	nil

Σ(solveRefineSlot)

Type	Notation
slot	SL
number	NU
composite	C

Function	Type
:position, :length	SL→NU
:leftShift, :rightShift	SL×NU→NU
:rightShift	SL×NU×NU→NU

Predicate	Type
isSlot	C
leftShift, rightShift	NU
value	NU×NU
slotRefinement	nil

Nederlandse samenvatting

Inleiding

Het in dit proefschrift beschreven onderzoek is gericht op de ontwikkeling van een nieuwe generatie Computer Aided Design (CAD) systemen. Het doel van een CAD-systeem is het ondersteunen van een ontwerper. Deze ondersteuning houdt in dat bepaalde routinehandelingen en arbeidsintensieve klussen aan een computersysteem overgelaten kunnen worden en dat het systeem voorstellen aan de ontwerper doet welke richting het ontwerp op zou kunnen gaan. De ontwerper bepaalt de richting van het ontwerpproces en het systeem moet voldoende flexibel zijn om deze richting te kunnen volgen; op sommige momenten worden bepaalde taken gedelegeerd worden naar het CAD-systeem. Teneinde zulke taken te kunnen vervullen moet het systeem kunnen beschikken over de ontwerpkennis aangaande deze taken. Bovendien moet het systeem kunnen redeneren met deze kennis. De huidige generatie CAD-systemen bezit dit soort kennis en het vermogen tot redeneren niet of nauwelijks. Het onderzoek heeft zich voornamelijk gericht op CAD-systemen die werktuigbouwkundig en ook architectonisch ontwerpen ondersteunen.

Het proefschrift handelt over de programmeertaal *ADDL* (Artifact and Design Description Language), specifiek ontworpen voor de implementatie van CAD-systemen. ADDL is een kennisgebaseerde taal specifiek ontwikkeld voor de representatie van ontwerpkennis. De taal is ontworpen aan de hand van een beschrijvend model van het ontwerpproces. Dit model beschrijft het ontwerpen als een iteratief proces dat een representatie van een *ontwerpobject* stapsgewijs verfijnt totdat een uitgewerkt en fabriceerbaar *produktmodel* is ontstaan. De *object-representatie* is derhalve onderhevig aan toestandsveranderingen beginnend met

een abstracte representatie en eindigend met een volledig gedetailleerde representatie. Tussenliggende objectrepresentaties beschrijven een incomplete toestand van het ontwerpobject die steeds nauwkeuriger wordt naarmate het ontwerpproces vordert. Een ontwerpstap bestaat uit het toevoegen van nieuwe informatie aan de huidige toestand zodat een minder incomplete nieuwe toestand onstaat. Een stap is derhalve gebaseerd op een analyse van de huidige toestand en resulteert in de synthese van nieuwe informatie. De huidige toestand met daaraan toegevoegd de nieuwe informatie vormt een nieuwe toestand die weer geanalyseerd kan worden. Dit proces wordt vervolgd totdat een objectrepresentatie is verkregen die voldoet aan de specificaties.

Uit het bovenstaande blijkt dat ADDL in staat moet zijn zowel ontwerpobjecten als het ontwerpproces adequaat te representeren. Daartoe bestaat de taal uit drie verschillende functionele componenten. De eerste component bestaat uit de representatie van het ontwerpobject. Zowel de afzonderlijke onderdelen van het ontwerpobject als de samenhang tussen die delen kunnen beschreven worden in ADDL. De tweede component representeert de kennis die een CAD-systeem nodig heeft om een ontwerpobject te manipuleren. In andere woorden, het bevat de kennis over de eerste component; ook wel *objectkennis* genoemd. De objectkennis is vastgelegd door middel van als-dan-regels. De derde component redeneert over het redeneren met de objectkennis. Het bevat de kennis (ook wel *metakennis* genoemd) over de wijze waarop de regels beschreven door de tweede component toegepast moeten worden. De metakennis wordt ook gerepresenteerd door als-dan-regels. De metakennis bepaalt *welke* ontwerpstappen er genomen moeten worden in een bepaalde fase van het ontwerpproces. De objectkennis bepaalt *hoe* deze stap vervolgens genomen wordt. Deze componenten worden achtereenvolgens behandeld in de volgende drie paragrafen.

Objectrepresentatie

Een ontwerpobject wordt beschreven als een gestructureerde verzameling losse onderdelen. Deze onderdelen worden gegroepeerd in assemblages die op hun beurt deel uitmaken van (hogere orde) assemblages. Het onderdeel op het hoogste aggregatieniveau representeert het ontwerpobject als geheel. De onderdelen op het laagste decompositieniveau vormen de bouwstenen van het uiteindelijke produkt. Zowel de onderdelen als de assemblages worden in ADDL gerepresenteerd door middel van *ADDL-objecten*. De decompositie resulteert in een objectrepresentatie in de vorm van een hiërarchie van ADDL-objecten.

Een ADDL-object is een abstract datatype bestaande uit attributen en operaties. De attributen representeren een expliciet bepaalde eigenschap van een object zoals bijvoorbeeld de lengte van een tafelpoot of de hoogte van een tafel. Operaties worden gebruikt om bepaalde eigenschappen te berekenen als deze niet expliciet gegeven worden. Deze impliciet bepaalde eigenschappen hangen dan af

van andere eigenschappen. De hoogte van een tafel kan bijvoorbeeld afhangen van de lengte van de poten en de dikte van het blad. In dit geval kan de hoogte van een tafel gerepresenteerd worden door een operatie in plaats van een attribuut. De hoogte wordt bepaald door de eigenschappen van de poten en het blad.

De samenhang tussen de verschillende onderdelen van een ontwerpobject is vastgelegd door middel van predikaatsymbolen. Een predikaatsymbool geeft een relatie weer tussen de objecten waarop het is toegepast. Bijvoorbeeld, de relatie `heeftOnderdeel(tafel1,poot2)` beschrijft dat `poot2` een onderdeel van `tafel1` is en de relatie `steuntOp(blad1,poot2)` geeft aan dat `blad1` op `poot2` steunt. De attributen en operaties beschrijven waarden en berekeningen met betrekking tot de kwantitatieve eigenschappen van objecten terwijl predikaatsymbolen zowel kwantitatieve als kwalitatieve eigenschappen van objecten representeren. De verzameling van objecten en relaties over de objecten heet de *object-informatie-toestand* van het ontwerpobject.

Representatie van objectkennis

Wil een CAD-systeem kunnen redeneren over een ontwerpobject, dan moet het kennis over dit object bezitten. Deze kennis wordt *objectkennis* genoemd. In ADDL wordt deze kennis gerepresenteerd door middel van logische als-dan-regels (ook wel implicaties genoemd). Logische als-dan-regels zijn regels van de vorm "**IF** ··· **THEN** ···". Het gedeelte na de **IF** wordt het *antecedent* genoemd en het gedeelte na de **THEN** het *consequent*. Als het antecedent van een regel waar is met betrekking tot een object-informatie-toestand van het ontwerpobject dan geldt het consequent als een geldige conclusie over het object. Enkel het opslaan van kennis door middel van als-dan-regels is natuurlijk niet voldoende. Om te kunnen redeneren met de kennis moeten de regels ook toegepast kunnen worden. Een *inferentie-mechanisme* zorgt voor het redenerend vermogen van ADDL. Het bepaalt welke regel toepasbaar is op een object-informatie-toestand en trekt valide conclusies uit de regel en de toestand.

Een relatief simpel ontwerpprobleem brengt al een enorm aantal regels met zich mee. Als er geen structuur in deze regels is aangebracht ziet zowel het inferentie-mechanisme als de programmeur door de bomen het bos niet meer. Een groot aantal regels maakt het voor het inferentie-mechanisme erg lastig een geschikte volgende regel te zoeken. Bovendien is het voor de programmeur moeilijk bij te houden waar welke kennis gerepresenteerd is en hoe deze kennis uit te breiden. Daarom zijn de regels in ADDL gegroepeerd in *scenarios*. Een scenario bevat de regels die nodig zijn voor het uitvoeren van één bepaald type ontwerpstap. Door middel van het sequentieel uitvoeren van verschillende scenarios worden de verschillende stadia van het ontwerpproces doorlopen.

Representatie van proceskennis

Zoals hierboven al gesteld is, wordt het ontwerpproces gemodelleerd door het stapsgewijs uitvoeren van scenarios. Dit brengt ons tot het probleem welk scenario gekozen moet worden in een bepaalde object-informatie-toestand. Hiertoe is ADDL uitgerust met meta-niveau scenarios. Meta-niveau scenarios besturen de wijze waarop het ontwerpproces wordt uitgevoerd. Meta-niveau scenarios onderscheiden zich van gewone (of object-niveau) scenarios in die zin dat meta-niveau scenarios een *proces-informatie-toestand* uitbreiden terwijl object-niveau scenarios een object-informatie-toestand uitbreiden. De kennis, opgeslagen in de meta-niveau scenarios, wordt ook wel *proceskennis* genoemd.

De regels op het meta-niveau stellen doelen die gerealiseerd worden op het object-niveau. Gedurende het uitvoeren van een ADDL-programma vindt er derhalve een continue interactie plaats tussen het meta- en object-niveau. Op het meta-niveau wordt er afhankelijk van de proces-informatie-toestand een doel gesteld. Als dit doel is verwezenlijkt is de ontwerper weer een stapje dichter bij de oplossing gekomen. De activering van een object-niveau scenario zorgt hiervoor. De uitvoering van dit scenario leidt tot de toevoeging van gegevens aan de object-informatie-toestand. Na het beëindigen van het scenario verschuift de besturing van object-niveau naar meta-niveau; er wordt een nieuw doel gesteld. Naast het stellen van doelen kunnen meta-niveau regels ook andere conclusies trekken. Deze conclusies hebben betrekking op de toestand van het ontwerpobject gerelateerd aan de fase van het ontwerpproces. Deze zogenaamde *procesparameters* geven aan dat het gedeelte van het ontwerpobject waarop ze betrekking hebben een bepaalde fase van het ontwerpproces achter de rug hebben. De parameter *abstract* geeft bijvoorbeeld aan dat voor het onderdeel waar de parameter betrekking op heeft een abstract beschrijving is gemaakt. Deze beschrijving bevat de belangrijkste functionele componenten van het ontwerpobject zonder dat er sprake is van een duidelijke samenhang tussen de componenten.

Resultaten

De in dit proefschrift beschreven kennis-representatietaal geeft op het eerste gezicht een redelijk complexe indruk. In het dagelijks gebruik blijkt dit alleszins mee te vallen. Zowel een doctoraal student aan de universiteit van Tokio als een hts-student werktuigbouwkunde in Amsterdam konden na een inwerkperiode van twee weken goed met ADDL uit de voeten. Waarbij vermeld dient te worden dat beide studenten niet of nauwelijks ervaring hadden met logische programmeertalen.

De huidige versie van ADDL is een prototype en kan nog verder uitgebreid worden. Zo kan de interactie met de ontwerper nog verbeterd worden. Mijn gedachten gaan daarbij uit naar speciale user-interface scenarios die de dialoog

met de ontwerper onderhouden. Verder zijn ontwerpers zeer geïnteresseerd in grafische representaties van ontwerpobjecten. Voor dit soort representaties zijn er nog geen expliciete taalconstructies ingebouwd. Binnen het onderzoeksproject heeft een promovendus zich juist met deze representaties beziggehouden. Het is de intentie het daaruit voortgevloeide systeem en ADDL in de nabije toekomst te koppelen. Al met al is ADDL een taal die geschikt is voor het representeren van ontwerpkennis. In het huidige stadium is ADDL echter voornamelijk bruikbaar voor het implementeren van ontwerpsystemen die bepaalde taken op vrij mechanische wijze kunnen vervullen. Het systeem dient nog uitgebreid te worden met componenten die op een meer geavanceerde wijze textuele en grafische interactie met de ontwerper kunnen onderhouden. De ontwikkeling van ADDL als een geïntegreerd onderdeel van een complex ontwerpsysteem is de volgende uitdaging die aangegaan dient te worden.

Stellingen behorende bij het Proefschrift

On the Development of an Artifact and Design Description Language

Paul Veerkamp

1. Iemand die samenleeft met een promovendus dient daarvoor beloond te worden met een promotie zonder zelf een proefschrift te hoeven schrijven.
2. De leercurve van Smalltalk is te vergelijken met die van Japans voor een westerling. De syntax is redelijk eenvoudig onder de knie te krijgen maar er is flink wat studie vereist voordat het vocabulaire eigen is gemaakt.
3. Het descriptieve model van het ontwerpproces zoals beschreven is in hoofdstuk twee van dit proefschrift kan ook gebruikt worden om de verschillende stadia van een promotieonderzoek te beschrijven.
4. Misdaad is het enige gezwel waarbij een *celtekort* optreedt.
5. Gezien het feit dat er sinds de jaren dertig onafgebroken confessionele partijen in de regering hebben gezeten kan Nederland amper een seculiere staat genoemd worden.
6. Een volk is een volk niet door een gemeenschappelijke taal, noch door een gemeenschappelijk stuk land maar zuiver door een gemeenschappelijke historie.
7. De kloof tussen het bedrijfsleven en de universiteiten heeft gezorgd voor de achterstand van Nederland op het gebied van informatica.
8. Het gegeven dat het nederlandse woord 'kop' en het engelse woord 'mug' zowel de betekenis 'hoofd' als 'drinkgerei' heeft komt voort uit de gewoonte van de Vikingen Calva te drinken uit de schedels van de overwonnen vijanden.
9. Het rijgedrag van automobilisten zou aanzienlijk verbeteren als zij verplicht zouden worden één dag per maand in Amsterdam te fietsen (en vice versa).