

Conceptual design of  
controlled electro–mechanical systems  
– a modeling perspective –

**PROEFSCHRIFT**

ter verkrijging van  
de graad van doctor aan de Universiteit Twente  
op gezag van de rector magnificus  
prof. dr. Th.J.A. Popma  
volgens het besluit van het College voor Promoties  
in het openbaar te verdedigen  
op donderdag 17 februari 1994 te 15.00 uur.

door

**Theodorus Jacobus Adrianus de Vries**

geboren op 3 maart 1966  
te West–Stellingwerf.

Dit proefschrift is goedgekeurd door  
prof.dr.ir. J. van Amerongen, promotor  
dr.ir. P.C. Breedveld, assistent promotor

To Inge,  
and to Jorik.

CIP data Koninklijke Bibliotheek, Den Haag

Vries, Theodorus Jacobus Adrianus de

Conceptual design of controlled electro-mechanical systems, a modeling perspective /

Theo J.A. de Vries. – [S.l.:s.n.]. – III

Thesis Enschede – With ref. – With summary.

ISBN: 90-9006876-7

Subject headings: conceptual design, modelling, mechatronics, CAE systems.

Cover design: Inge de Vries-Stuurwold.

© Theo J.A. de Vries, Enschede, The Netherlands.

---

## Summary

The subject of this thesis is the development of computer-based support for conceptual design of technical-physical systems, specifically controlled electro-mechanical systems.

Two new design approaches have been proposed recently in response to increasing demands on products to be designed, namely “concurrent engineering” and “mechatronics”. Both these approaches advocate a systems point of view in design, and stimulate integrated problem solving within designing. Two types of problems oppose the usage of an integrated problem solving approach during (conceptual) design:

- cooperation and coordination among members of the design team is difficult due to a lack of shared meaning of communicated messages.
- reduction of the initial lack of understanding of the design problem is hard, and the required learning takes too much time.

A formal description of the design process reveals that the use of models and abstractions plays a crucial role in communication and learning. Solution of the above problems demands improved modeling capabilities of design support systems.

One required improvement is to enable simultaneous formulations of one model in multiple languages, in such a way that the model can be manipulated in either of the formulations. This concept is called “multiple model formulations”. A system setup that enables multiple model formulations and yet keeps different formulations of a model consistent and tractable is devised. The setup incorporates automatic conversions between different formulations of one model. Bond graphs and iconic diagrams are taken as an example set of formulations for demonstrating the feasibility of the setup.

Furthermore, designers require more flexible model descriptions than contemporary modeling systems provide. The lack of flexibility is due to the way in which classification of subsystems is realized in these systems. An improvement of this is obtained by modularizing a subsystem description into a type and a specification, and by subtyping, i.e. by expressing a type as a specialization of a more general type. The combination of modularization and subtyping, named “polymorphic modeling”, leads

to hierarchical subsystem libraries and gives modeling systems the possibility to conform to the evolutionary nature of model building.

A model building environment for mechatronic systems is presented: the MAX system. MAX supports the user in creating models and evaluating them by means of network-based analyses. Multiple model formulations (i.e. bond graphs and iconic diagrams) and polymorphic modeling are incorporated in the system. Conversions between model formulations and polymorphic model refinements enable to learn about the design problem. Due to the availability of an extendible, well organized model library, the use of explicitly described models is made easy, which enhances communication about the design problem. These features make MAX into a powerful model building environment that is well adapted to usage by designers.

---

## Samenvatting

Dit proefschrift behandelt de ontwikkeling van computergebaseerde ondersteuning voor het conceptuele ontwerp van technisch–fysische systemen, met de nadruk op geregelde electro–mechanische systemen.

Twee nieuwe ontwerpbenaderingen zijn onlangs voorgesteld om tegemoet te komen aan de toenemende eisen die worden gesteld aan te ontwerpen producten, namelijk “concurrent engineering” en “mechatronica”. Beide benaderingen bepleiten een systeemgeoriënteerd gezichtspunt, en stimuleren een geïntegreerde aanpak van ontwerpvragestukken. Twee typen problemen bemoeilijken het gebruik van een geïntegreerde aanpak tijdens conceptueel ontwerpen:

- samenwerking en coördinatie van leden van het ontwerpteam verloopt moeizaam, omdat men geen gemeenschappelijke betekenis toekent aan uitgewisselde boodschappen.
- vermindering van het gebrek aan begrip van het ontwerpprobleem is moeilijk en het benodigde leerproces neemt veel tijd in beslag.

Een formele, integrale beschrijving van het ontwerpproces maakt duidelijk dat het gebruik van modellen en abstracties van de werkelijkheid een doorslag–gevende rol speelt, zowel tijdens communicatie als bij het leerproces. Oplossing van bovenstaande problemen vereist een verbetering van de modelleringsmogelijkheden van ontwerpssystemen.

Ten eerste moet het mogelijk zijn om één model gelijktijdig in meerdere talen te formuleren, zodanig dat het model kan worden gemanipuleerd in ieder van deze formuleringen. Dit concept wordt “multiple model formulations” (meerdere modelformuleringen) genoemd. Een systeemopzet is ontwikkeld die meerdere modelformuleringen mogelijk maakt en toch de verschillende formuleringen consistent en beheersbaar houdt. Deze opzet omvat automatische conversies tussen verschillende formuleringen van hetzelfde model. Met bondgrafen en iconische diagrammen als voorbeeld is aangetoond dat deze systeemopzet realiseerbaar is

Voorts verlangen ontwerpers meer flexibiliteit van modelbeschrijvingen dan huidige modelvormingssystemen bieden. De oorzaak van het gebrek aan flexibiliteit is de manier waarop classificatie van subsystemen is gerealiseerd in deze systemen. Een verbetering hiervan wordt bereikt door modularisering van subsysteembeschrijvingen

in een type en een specificatie, en door subtypering, dat wil zeggen het uitdrukken van een type als een specialisatie van een meer algemeen type. De combinatie van modularisatie en subtypering, genaamd “polymorphic modeling” (polymorf modelvormen), leidt tot hiërarchisch georganiseerde subsysteembibliotheken en geeft modelvormingssystemen de mogelijkheid aan te sluiten bij het evolutionaire karakter van modelvormen.

Een modelvormingsomgeving voor mechatronische systemen is beschreven: het MAX systeem. Dit systeem ondersteunt de gebruiker in het aanmaken van modellen en het evalueren ervan door middel van netwerk-gebaseerde analyses. De concepten “multiple model formulations” (en wel voor bondgrafen en iconische diagrammen) en “polymorphic modeling” zijn samen opgenomen in het systeem. Conversies tussen modelformuleringen en polymorfe modelverfijningen maken het mogelijk snel inzicht te krijgen in het ontwerpprobleem. Door de aanwezigheid van een uitbreidbare, goed gestructureerde modellenbibliotheek kost het gebruik van expliciet vastgelegde modellen weinig tijd en moeite, zodat communicatie over het ontwerpprobleem verbetert. Deze eigenschappen maken MAX tot een krachtige modelvormingsomgeving die goed is aangepast aan het gebruik door ontwerpers.



---

## Table of Contents

1	Introduction.....	1
1.1	Problem area .....	1
1.2	Subject of research.....	2
1.3	Approach and aims .....	7
1.4	Thesis overview .....	8
2	Understanding the design process .....	11
2.1	Introduction.....	11
2.2	Characterizing design.....	12
2.3	Existing models of designing.....	14
2.4	Proposition for a new model of designing.....	19
2.4.1	World view .....	19
2.4.2	Basic model .....	20
2.4.3	Adding structure .....	26
2.4.4	Incorporating the TEA model.....	30
2.5	Evaluation of the new model .....	30
2.5.1	Comparison to existing models .....	32
2.5.2	Characteristics revisited.....	33
2.5.3	Enhancing design.....	35
2.6	Developing advanced support.....	37
2.7	Conclusions.....	40
3	Multiple model formulations .....	43
3.1	Introduction.....	43
3.2	Languages for conceptual design.....	45
3.2.1	Requirements .....	46
3.2.2	General characteristics.....	46
3.3	Selection.....	48
3.3.1	Functional formulation .....	49
3.3.2	Schematic formulation.....	51
3.3.3	Comparison.....	55
3.3.4	Behavioral description.....	56
3.3.5	Evaluation.....	57

3.4	Design of the system set-up .....	58
3.4.1	Main design issue .....	58
3.4.2	Conversions .....	60
3.5	Realizability .....	63
3.6	Formalism for describing the core model .....	64
3.7	Conclusions .....	65
4	Iconic diagrams and bond graphs .....	67
4.1	Introduction .....	67
4.2	Describing iconic diagrams .....	68
4.3	General considerations on conversions .....	71
4.4	Iconic diagram to bond graph algorithm .....	73
4.4.1	Available methods .....	73
4.4.2	Outline .....	74
4.4.3	Orientation analysis .....	75
4.5	Bond graph to iconic diagram algorithm .....	77
4.5.1	Available methods .....	77
4.5.2	Inquiry .....	78
4.5.3	General characterization of expansions .....	79
4.5.4	Remaining issues .....	82
4.5.5	Outline .....	86
4.6	Conclusions .....	87
5	Polymorphic modeling of engineering systems .....	89
5.1	Introduction .....	89
5.2	Existing techniques .....	91
5.2.1	Parametrization .....	91
5.2.2	Typing .....	92
5.2.3	Port based interfacing .....	94
5.3	Evaluation .....	96
5.4	Polymorphic modeling .....	98
5.5	Consequences .....	100
5.5.1	Support for evolutionary model building approach .....	101
5.5.2	Hierarchical subsystem library .....	101
5.5.3	Creation of alternatives .....	103
5.6	Design issues .....	103
5.6.1	Modularization .....	103
5.6.2	Subtyping .....	106
5.7	Application advice .....	107
5.8	Conclusions .....	109

6	MAX, a mechatronic modeling environment.....	111
6.1	Introduction.....	111
6.2	System development.....	112
6.2.1	Main issues.....	112
6.2.2	Identification of required support.....	113
6.2.3	Organization into system framework.....	116
6.2.4	Presentation to the user.....	118
6.2.5	Multiple model formulations and polymorphic modeling.....	119
6.3	State of the art.....	121
6.3.1	Overview.....	121
6.3.2	Hierarchy of subsystem types.....	123
6.4	Case study.....	125
6.5	Evaluation.....	129
6.5.1	Favorable points.....	130
6.5.2	Weaknesses.....	132
6.5.3	Desirable extensions.....	133
6.6	Conclusions.....	134
7	Discussion.....	137
7.1	Synopsis.....	137
7.2	Conclusions.....	138
7.3	Suggestions for future work.....	141
A	Iconic diagram symbols.....	143
B	Simplification of bond graphs and iconic diagrams.....	145
B.1	Rule 1: Dangling junctions.....	145
B.2	Rule 2: Elimination of junctions.....	145
B.3	Rule 3: Joining of junctions.....	146
B.4	Rule 4: Elimination of a Double Difference.....	146
C	The THESIS formalism.....	149
C.1	The context diagram.....	150
C.2	The activity diagrams.....	151
C.3	The interface declaration part.....	152
C.4	The primitive process specification.....	154
C.5	The control specification.....	154
C.6	The store specification.....	154
D	Models of the example system.....	157
	References.....	161



## Introduction

### 1.1 Problem area

A typical and fascinating characteristic of human beings is the ability to create artifacts that suit their own purposes. Whilst this activity may have started with something like the ‘simple’ modification of a flint so that it could be used as a knife, evolution of both the abilities and the needs of mankind resulted in the development of completely new kinds of objects and processes. An everyday example which clearly demonstrates this evolution is the computer. Prehistoric man was not able to create such things, but also did not need them. Due to the creativity of mankind, we live in a world which is more and more man-made, or artificial, than natural. In here, we will be concerned with the “science of creating the artificial” (Simon, 1981).

In modern society, the creation of new artifacts is largely carried out in industrial companies operating in an economic environment. As a result, the characteristics required of newly created artifacts have changed. They not only need to suit purposes as before (*functionality*), they also need to work well (*performance*) over a significant period of time (*reliability*) for acceptable prices (*cost-effectiveness*). In addition, the time needed to generate a new artifact (*innovation time*) has become an important issue due to the pressure of competition. To deal with all this, the creation of artifacts has been split up into two phases, which are quite separate (Cross, 1989). In the first phase, the *design process*, the conception of the artifact to be created is formed and a description, or *model*, of it is made. This model may not only contain features of the artifact itself, but also of the way in which it can be made, the production. Only after the final, complete description (the design) has been finished, is the second phase entered: the artifact will be physically realized in the *manufacturing process*, leading to an actual, physical product.

The functionality, performance and reliability of a product are largely determined during the design process, as during this process decisions are made on what the artifact will be like and how it will be produced. The decisions made during the design process also have the largest influence on the cost-effectiveness of the product. This is suggested by evaluation of data from the Ford Motor Company and Xerox (Ullman, 1992). Finally, it can be noted that the time needed for the design of a new product is usually much longer than the time needed for producing a new product. Therefore, the innovation time is also greatly dependent on the design process. It can thus be

concluded that in order to stay competitive it is critical for an industrial company to master the design process and to continually improve it. This thesis deals with the understanding of design and the development of proper support to enhance it.

We will specifically consider the design of *controlled electro–mechanical systems*. Examples of such systems are assembly machines, automatic guided vehicles, consumer products like CD–players and video cameras, etc. The number of systems belonging to this class is rapidly growing. Recent advances both in electronics and computer technology, in software engineering and in control engineering enable the addition of ‘intelligence’ to traditionally purely mechanical products. This increases their functionality, performance and reliability simultaneously, without necessarily increasing their cost.

The research described here is focused on controlled electro–mechanical systems, and approaches design activities in this field as follows:

- We regard artifacts from a systems point of view, i.e. as a structure of interrelated elements that are embedded in an environment.
- We study systems that realize their functionality using mainly electronic and mechanical parts, and a controller (either digital or analog). The use of a controller implies that the systems of interest have an active part. This active part performs some kind of signal processing to force the system to behave in a certain way.
- We concentrate on aspects that relate to the energetic (and therefore the dynamic) behavior of the systems. The reason for this is that it is through energy exchange that functional interactions between subsystems take place.

## 1.2 Subject of research

Controlled electro–mechanical systems are generally made up of a large number of parts, which interact in many ways. Such systems are *complex*, because the whole is more than the sum of the parts in the sense that given the properties of the parts and the laws of their interaction, it is not a trivial matter to infer the properties of the whole (Simon, 1981). Their complexity is essential; with enough effort we may master it and even engineer an illusion of simplicity in their interfaces, but we can never remove the internal complexity (Booch, 1991).

The inherent complexity of the design of a controlled electro–mechanical system will have to be tackled in order that the outcome will be a reliable, high performance product with the proper functionality. The technique to do this has been known since ancient times: *divide et impera* (divide and rule). Two major ways of decomposing the design problem have traditionally been applied in industry: a division concerned with the *product life cycle*, and a division concerned with problem domains or *disciplines*. Both divisions have had considerable consequences on design practice. Neither of the decompositions alone is sufficient to explain the way design problems are approached. Although presented here as distinct, in fact a strict separation of the two is not possible. Different phases of the product life cycle imply to some extent a focus on

different problem domains, and vice versa. However, within the scope of this treatment it suffices to consider them one after the other.

A product life cycle can be assumed to consist of five stages:

- 1 Design. The conception of the product and of the means of production is formed, and a more or less precise description of both is made.
- 2 Production. The product is realized as a result of a manufacturing and/or assembly process.
- 3 Sale. The product is distributed, and sold to the consumer.
- 4 Service. The product is in normal operation.
- 5 Product retirement. Liquidation of the product, recycling of product parts.

Within the design process, three design phases have traditionally been distinguished, which address different stages of the product life cycle.

- 1.1 *Product definition.* A statement is developed what the product should be in terms of specifications and requirements from the perspective to optimize its sale. The feasibility of the product (both technical and economic) is estimated and a plan of resources for the remaining phases is made. Most of the times, the marketing department is responsible for this phase.
- 1.2 *Product design.* A plan is made on how to implement the deliverables of the product. Recently, concerns of product retirement influence this plan considerably. This phase is the responsibility of the design department.
- 1.3 *Manufacturing engineering.* The method of production is devised and the production system is prepared. The engineering department is generally responsible for this phase.

During product assessment, product design and manufacturing engineering, the design problem is usually decomposed into subproblems in specific disciplines. As a result, design is generally done in a team composed of different specialists. Decomposition over disciplines also effects the outcome of the design process, because typically the resulting design consists of a collection of subsystems which provide functionality in a single domain and have relatively little mutual interaction. In other words, each function of the product is largely realized by one specific subsystem or module. Inter-module linkages have been avoided as much as possible. Finally decomposition over disciplines also influences the phasing of the design process, as generally one subsystem will be designed after the other. In the case of product design of a controlled electro-mechanical system, the standard pattern is to design first the mechanical modules (which might be viewed as the skeletal and muscular subsystems), then the electronic parts (the sensory- and nervous subsystems), and finally the control modules (the brains).

From the above we may conclude that the decomposition of design problems over phases of the product life cycle and over problem domains overcomes their complexity. However, it has had three major, unfavorable effects on design:

- it has stimulated designers to only regard the solutions they propose from the perspective of the subproblems they are concerned with, and hence to pay too little attention to whether the solutions are satisfactorily at the overall problem level.
- it has turned the design process into a sequence of quite separate phases, whereby the design is passed from one phase to the following in a manner that can best be characterized as ‘throwing over the wall’.
- it has led to designs consisting of subsystems providing functionality in single domains and having little mutual interaction, which often implies that the resulting design is over-complex and not optimal.

Due to the nature of evolution, designers are continually faced with increasing demands both on the products they are designing and on their own productivity. In terms of the previous section, they are required to come up with new designs which have more functionality, higher performance and better reliability (i.e. better *quality*) for less cost and in less time. These demands have become so high that the traditional approach to design is no longer sufficient. The sequence of design phases causes the innovation time to be too long and the quality to be hard to guarantee. The specialized subsystems are a barrier to the improvement of quality if costs are not to be increased. Two new approaches have been proposed to change this.

### ***Concurrent engineering***

Concurrent engineering (also referred to as simultaneous engineering) is “a systematic approach to the integrated, concurrent design of products and their related processes, including manufacture and support. This approach is intended to cause the developers, from the outset, to consider all elements of the product life cycle from concept through disposal, including quality, cost, schedule, and user requirements” (IDA, 1988). One might say that this approach emphasizes that the decomposition of design problems over the product life cycle is useful, but that the design phases which result should be carried out more or less concurrently instead of sequentially. Many success stories about the merits of this design approach have appeared in literature, especially from the side of the automotive industry (Clark and Fujimoto, 1991). These merits generally are impressive reduction of innovation time and at the same time improvement of the quality of new products.

### ***Mechatronics***

A commonly used definition of mechatronics is “a synergetic combination of precision mechanical engineering, electronic control and systems thinking in the design of products and manufacturing processes” (IRDAC, 1986). Buur (1990) proposed a more concise definition: mechatronics is “a technology which combines mechanics with electronics and information technology to form both functional interaction and spatial integration in components, modules, products and systems”. Mechatronics can thus be viewed as an approach which recognizes the usefulness of decomposition of design problems over disciplines, but which emphasizes that this should not lead to domain-specific subsystems with minimal functional interaction and spatial integration. Benefits claimed to result from use of a mechatronic design approach are mainly the



increase of functionality (more intelligence and flexibility), of performance and of reliability without increasing costs.

Both concurrent engineering and mechatronics advocate a *systems point of view* in design. They are similar in nature and stress the need for integration within the design process. Yet, they are more or less ‘orthogonal’, as one concentrates on the product life cycle, while the other is mainly concerned with the problem domain. In other words, concurrent engineering and mechatronics are complementary design approaches to stimulate ‘integrated problem solving’ (Clark and Fujimoto, 1991) in the design process. Based on their study of American, Japanese and European automobile companies, Clark and Fujimoto (1991) concluded that integration of problem solving has a direct bearing on a number of measures, e.g., innovation time and total product quality. Hence, concurrent engineering and mechatronics both will improve the design process and can be naturally applied in combination.

Considering their similarities, it is not surprising to see that the two approaches are opposed by the same kinds of problems when applied. Two different types of problems can be observed.

#### ***Communication problems.***

Integrated problem solving requires better cooperation and coordination among the members of the design team (Takeuchi and Nonaka, 1986). Their collaboration is to be realized through communication. However, members are generally individuals separated by discipline and/or functional responsibility. Their communication is made difficult due to a *lack of ‘shared meaning’* (Konda et al., 1992). Simply stated, people just do not speak the same language or do not attribute the same value to a common word (figure 1.1). These problems were also present in the traditional way of designing, but they are much more dominant in the new situation.

#### ***Learning problems.***

During design a circular dependency is present. Finding a good solution requires an understanding of the problem; conversely, problem understanding can almost only be gained throughout the solution process (figure 1.2). This “design process paradox” can only be solved by trying to reduce the lack of knowledge as much as possible early in the process. In other words, one needs to learn about the design problem and the proposed solution as soon as possible. Compared to the traditional approach, the need for learning is both more serious and more difficult to satisfy when applying an integrated problem solving approach.

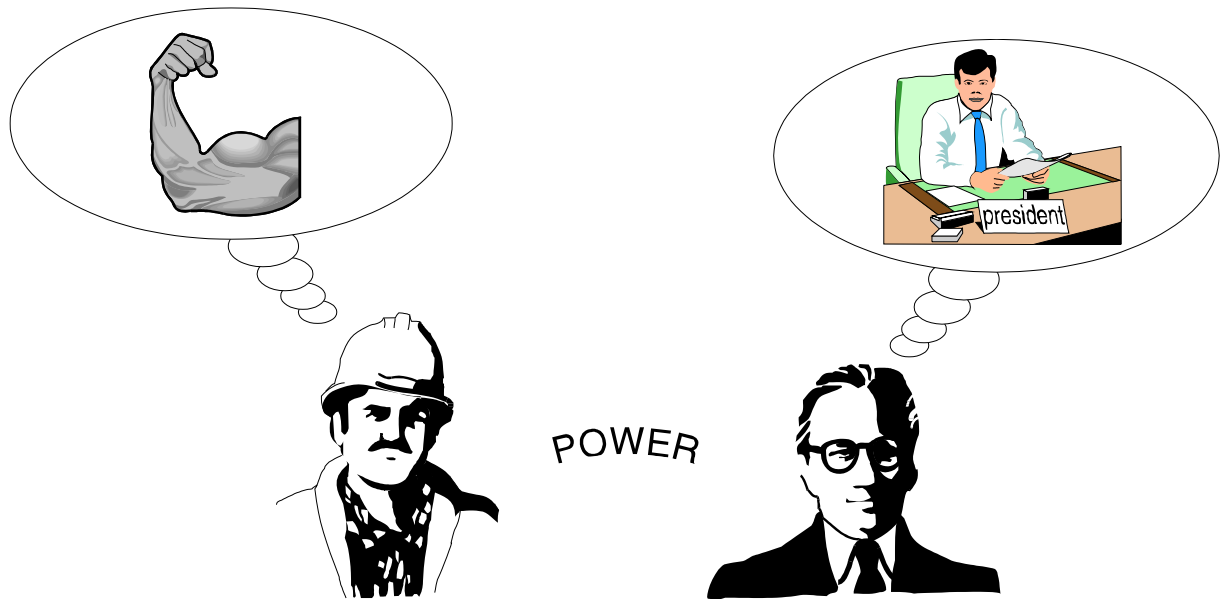


FIGURE 1.1 *Lack of shared meaning*

To successfully apply an integrated problem solving approach, the communication– and learning problems need to be tackled. This can be done in three different ways: through organizational measures, by means of education and by means of technological solutions.

Japanese companies have proved that *organizational means* can enhance integrated problem solving. This means generally to modify organizational structures with the aim to increase cooperative behavior, to optimize organizational transfer of learning, and to stimulate self–organising project teams (Takeuchi and Nonaka, 1986; Buur, 1990; Clark and Fujimoto, 1991). *Education* of people involved in designing might be another way to overcome the problems. This education should be focused towards improving communication by means of ‘generalistic’ training and towards resolving the design paradox by experimenting with problem solving under uncertainty. *Technological solutions*, finally, can also be implemented. These technological solutions involve the development of support, i.e. tools, techniques and methodologies, that can help to prevent or to deal with the communication problems and the design paradox.

Like others (Konda et al., 1992; Buur, 1990) we believe that none of the indicated solutions alone will be effective, but need to be accompanied by the others. However, in this investigation we will only consider *technological issues* of the problems mentioned above. Both organizational and educational matters are beyond the scope of this thesis.

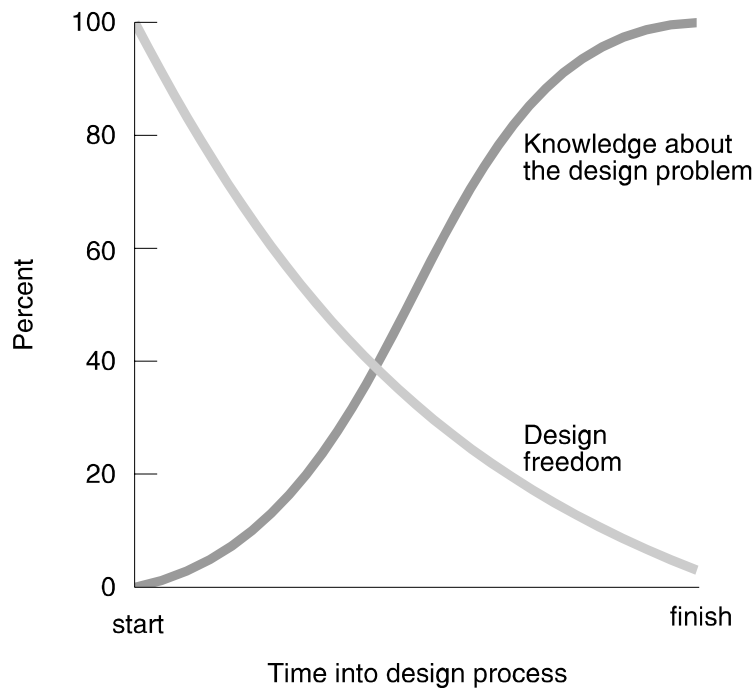


FIGURE 1.2 *The design process paradox (adapted from Ullman (1992), figure 1.8)*

### 1.3 Approach and aims

For synthesizing good solutions a deep understanding of the problem is required. Therefore, a theoretical investigation of the design of engineering systems is needed. On basis of this investigation, an analysis of the problems mentioned above can be given. It will be shown that a major cause of the problems is the lack of proper support for the *conceptual design task*, i.e. the part of the process where it is roughly decided how the product will function and what it will look like (Ullman, 1992).

Development of support requires a normalization of the design process. However, conceptual design is often regarded as hard, or maybe even impossible, to formalize, because it is an art form that requires creativity and expertise. On one hand, we agree with this view in that we think it is currently not possible to *completely* formalize this process, i.e., to automate it. On the other hand, we believe (and intend to show in this thesis) that parts of it can well be formalized. A major problem is to find out *which parts* can and should be formalized and automated. This selection has to be done with great care due to the nature of the conceptual design task. When some kind of support system is not properly set up, i.e., when it ‘over-’ or ‘underformalizes’, it will be of no help to a designer. Moreover, it may well influence the process in a negative way, as it frustrates creativity.

We think that a *language-based approach* is required. The importance of languages in design is well known (Cross, 1989; Dasgupta, 1991; Ullman, 1992). Furthermore, both

communication and learning, which are central to the problems discussed here, critically depend on the use of (the proper) language (Popper, 1972).

Based on the foregoing, we can conclude that the aims of this thesis will be the following.

- 1 Analyze the design process and formulate a model of design that can help to understand the design process and develop proper support for it.
- 2 Formalize technological concepts which will support the conceptual design of controlled electro–mechanical systems.
- 3 Evaluate the aforementioned model of design and technological concepts by (further) developing a prototype Computer Aided Engineering environment.

## 1.4 Thesis overview

*Chapter 2* starts with a description of the design process by means of a *model of designing*. Next, an analysis is given of the problems that are present when using an integrated problem solving approach in designing. It follows that the lack of proper support is largely due to the fact that the modeling techniques and means for the conceptual design task have not been powerful enough. Two concepts are proposed to improve this: multiple model formulations and polymorphic modeling.

In *chapter 3*, the concept of *multiple model formulations* is described. The idea of this concept is to enable one model to be formulated in multiple ways, i.e., using multiple ‘languages’. First, characteristics of the languages that are applied during conceptual design are identified. On the basis of this, a set of formal modeling languages is selected for the design problems of interest here. Next, a system set–up to support multiple model formulations in a CAE system is proposed.

The system set–up derived in *chapter 3* incorporates conversions between the different formulations. The design of *conversion algorithms* for two model formulations, namely iconic diagrams and bond graphs, is discussed *chapter 4*. Applications of the conversion algorithms are shown.

*Polymorphic modeling*, which is the topic of *chapter 5*, relates to the flexibility that designers require of model descriptions. It is shown that current modeling systems do not offer this flexibility, due to the way in which classification of subsystems is realized in these systems. A proposition is put forward on how to improve this. It is concluded that polymorphic modeling is powerful because it supports the evolutionary nature of design by facilitating the variation of detail of a model and the creation of alternative design solutions. Examples of its usage are included to illustrate this.

*Chapter 6* presents the *system* (called MAX) in which the concepts of multiple model formulations and polymorphic modeling have been implemented. The chapter includes a *case study* to demonstrate the functionality of the system. The theory and concepts

which have been discussed in the previous chapters are evaluated on basis of this. It is shown that the system is indeed able to support the modeling of the design during the conceptual design task.

Overall *conclusions* are listed in *chapter 7*, together with opportunities for further research and possibilities to extend and improve MAX.



## Understanding the design process

### 2.1 Introduction

An important way to deal with the difficulties of engineering design is to provide good support for the designers. Two basic criteria can be formulated for qualifying design support as ‘good’:

- it must aid the designer with a task of genuine concern
- it must not introduce more (or more difficult) problems than those it solves

Information technology enables sophisticated support by means of computer-based support systems. However, a literature review by John (1988) reveals that current systems do not meet the above criteria. The systems suffer from serious deficiencies that may be summarized as:

- poor support of relevant design functions and phases
- support is not tailored to the use by a designer
- foundations of the systems are not explicit or even wrong.

A main cause for these deficiencies is that such systems are being developed without sufficient understanding and regard of the actual design process (Pugh, 1984). For example, conventional ‘Computer Aided Design’ (CAD) systems provide support to the mechanical designer in making the final drawing of the design. But few designers will consider this task to be problematic while designing. Even more, if designed badly, they limit the designer’s freedom and negatively influence the thinking process. On the other hand, CAD systems do provide good support for the maintenance and reuse of final drawings of designed artifacts. Therefore, one might say that conventional CAD systems are based on the misconception that designing mainly involves making and maintaining a final drawing of the designed artifact. To stress this, these systems are sometimes referred to as Computer Aided Drafting systems. The term ‘Computer Aided Engineering’ (CAE) systems has been introduced for systems specifically aiming at the support of other design tasks as well. We use the latter term to avoid confusion.

So insight into the design process is important for developing good CAE systems. Insight can be gained by formally describing the design process, i.e. by *modeling*

*design*. Our research hypothesis is that better support results when systems are developed on the basis of such models. The use of this approach is not unique; it seems to be the motivation for much of the research on the design process (Finger and Dixon, 1989a).

To provide insight, a model of designing should “explain how the design process unfolds” and “be able to predict future successes and failures” (Ullman et al., 1988). Therefore, a superficial description of observations of design processes will not suffice. Instead, a ‘*deep*’ model of designing is needed. A deep model identifies structure and separates important issues from less important ones. The model of designing we seek will be applied in the development of (improved) design support. Based on the model it should thus be possible to:

- 1 *explain* the applicability of design support on the basis of characteristic features of the process.
- 2 *predict* what kind of support would increase chances for successful completion of the process.

In this chapter, a model of designing which meets these demands is presented, and lacking support is identified. Note that if we speak about a model in this chapter, we mean a model of designing, unless explicitly stated otherwise.

We first characterize engineering design in the next section. Based on this, existing models of designing are reviewed in section 2.3. It is shown that these models are inadequate; modifications and extensions to be incorporated in a new model of designing are proposed. The new model is introduced in section 2.4 in three steps. First, we define the context in terms of a world view. Next, we identify the main structure of the model. Finally, we add detail to the parts contained in the structure, thereby increasing the amount of information captured by the model. The resultant model is evaluated in section 2.5, and is used to identify lacking support in section 2.6. Section 2.7 summarizes the conclusions.

## 2.2 Characterizing design

A single-sentence definition of designing is not sufficient to capture all of its relevant aspects. A more fruitful way of stating what constitutes design is to identify the fundamental characteristics of design, and use these as a means for understanding it at a deeper and less intuitive level (Dasgupta, 1991). Three characteristics are crucial.

### *Design is context dependent*

“There is growing evidence that context-free universal methods are most often inapplicable and inappropriate in design practice” (Konda et al., 1992). In other words, the *context* within which designing takes place is of critical importance. For example, the social context influences the design process, because social factors like negotiation mostly determine the outcome of major decisions during the process, thereby changing



the very character of the solution space. Also the technical context has influence, as suitability of support in the form of tools and (formal) methods depends on the domains of application. The problem context as well is crucial, as the requirements for the design are linked with the non-static environment in which design takes place.

### ***Design problems are ill-structured and incomplete***

The majority of design problems are *ill-structured problems* (Simon, 1973; Jones, 1980; Cross, 1989). Ill-structured problems have a problem setting that is not definitive, both the initial state and the goal are not stable nor verifiable. Also, such a problem often appears to be represented improperly in the course of the solution process, because the representation obscures the real issue, does not match with relevant knowledge or excludes optional solutions. This fact is easily understood if one realizes that specifications are generally composed on the basis of known ways to approach a problem. Design problems are usually *incomplete* at the start of the process as well, and possibly inconsistent. This is due to the complexity of the problem context and the lack of understanding of the problem area. Ill-structured and incomplete problems are dealt with by proposing solutions before the problem is well defined and understood. In other words, proposing solutions is a means to understand and (re)formulate the problem. However, multiple solutions are often proposed. These solutions might be considered equally valid initially. Only later are solutions evaluated to be preferable or invalid, if at all this will become clear. As a result of all this, there is no clear distinction between problem formulation and problem solution. Therefore, the process of designing does in general not only involve problem solving, but problem stating as well; making and revising the problem specification forms an essential part of designing. Consequently, both problem statement and solutions need to be captured integrately in the design object, for example in the form of a set of constraints (Simon, 1981).

### ***Design involves a time-constrained initiation of change***

Design is concerned with describing an artifact that can adapt the situation at the outset to a more suitable one, i.e. it *initiates a change* (Jones, 1980; Simon, 1981). This implies that the question of ‘what ought to be’ is addressed with more emphasis than the question of ‘what is’. Furthermore, the required change generally has to be realized *within time constraints*. As a result, the method of experimentation as common in natural sciences is mostly inapplicable, because time is lacking or it concerns a “one-off” design problem, such as a satellite. This is why things like ‘experience’ and ‘intuition’ generally play a major role in the design of new artifacts.

Simon (1973) shows that these three characteristics imply that designing involves a continual evaluation of whether the problem as formulated and the solution that is pursued still have relevance in the “real world outside”, and the modification of the design problem accordingly. He summarizes this as “the elusiveness of structure”. Hence, due to its characteristics, design has to be viewed as a *contextually situated, evolutionary process*. The evolution takes place on two levels. On the one hand the set of descriptions specifying requirements and features of the artifact being designed (i.e. the *design object*) evolves during designing. On the other hand the knowledge about

the design problem and how it can be solved (i.e. the *design knowledge*) evolves, because the understanding and experience of the designers involved in the design process grows. Both of these evolutions generally do not start from scratch at the beginning of the design process, and also probably will not stop after it finishes. The context is important, because it will have a considerable influence on the course of the evolutions.

To function well, design support should be adapted as much as possible to the above mentioned characteristics. Therefore the model of designing which we seek should explain:

- how the *context* within which designing takes place influences the process
- how the *evolution* of the design object and design knowledge takes place.

## 2.3 Existing models of designing

Many models of designing have been proposed in the past decades. Overviews of well-known models can be found in, for example, Finger and Dixon (1989a) and Dasgupta (1991). Because of the large number of models, we will discuss existing models on the basis of a taxonomy (or classification scheme), instead of treating them individually.

A rather complete taxonomy of models of designing has been proposed by Konda et al. (1992). Their taxonomy is an improved and extended update of the one described by Roozenburg and Cross (1991). The taxonomy has the form of a tree, in which the prime division is between models of designing that focus on the ‘design process’ versus models of designing that focus on the ‘design artifact’. At the second level, ‘descriptive | declarative’ models are separated from ‘prescriptive | procedural’ models. For descriptive process-focused models finally a distinction is made between ‘individual’ and ‘social’ models. Though interesting, there are two main problems with this taxonomy.

Firstly, a tree-like shape is not proper for a taxonomy. One reason for this is that a tree forces one to rank the aspects according to which the taxonomy is built up from most important to least important, i.e. it forms a hierarchy. This is not adequate; the prime division in the taxonomy of Konda et al. (1992) is not more important than the second one. A second reason is that a tree-shape does not enforce a systematic classification. Sub-divisions of different branches do not have to be based on the same aspect. This is unwanted for a taxonomy, as it prevents the identification of unfilled areas and as it makes comparison of models in different branches difficult. It is more appropriate to use a more-dimensional *matrix representation* than a tree-like shape for a taxonomy.

The second problem is that we do not agree with the aspects on which the taxonomy is based. The selected aspects are neither clearly described, nor easily identified (process-focused versus artifact-focused) nor do they cover all models (descriptive

and prescriptive). For these reasons, a new taxonomy will be constructed hereafter, inspired by the taxonomy of Konda et al. (1992).

The next three aspects can be used for categorizing models of designing:

### **1 Purpose**

The purpose for which a model is created is an essential characteristic, as it has a major influence on both its contents and its representation. According to Ullman (1989), three different categories of models can be distinguished:

- *descriptive* models of designing, intended to represent what actually *is* done during design.
- *prescriptive* models of designing, intended to represent how design *should* be done.
- *computational* models of designing, intended to represent how design *could* (partially) be done using machine computation.

### **2 Modeling approach**

In principle, two different approaches to modeling can be distinguished (Booch, 1991; De Vries and Breedveld, 1992). One can use:

- *a process oriented* approach, i.e. denote at the highest level the *operations* (steps, procedures) which are distinguishable in the overall design process.
- *an object oriented* approach, i.e. denote which *components* (parts, units) need to be active for realizing the design process.

### **3 Time character**

In principle, time can be captured in a model in two ways:

- the elements of the model (objects or processes) are invoked *sequentially*. This implies that the relations in the structure of the model, if explicitly present, incorporate a time *sequence*. The structure thus depicts a kind of planning. Such a model is *serial* of nature.
- the elements of the model can be active *simultaneously*. In such models, the structure incorporates *co-existing* relations, which do not suggest a global plan. Rather, they describe a framework or environment within which designing takes place. Models of this kind can be characterized as *concurrent*.

This leads to a taxonomy as depicted in figure 2.1. We have made no effort to categorize many or all known models. Merely, we have chosen one well known example for each category that is distinguished.

The aim of this chapter is to present a model of designing that explains the applicability of design support and predicts what kind of support would further enhance the design process. The question in which category of the taxonomy the most suitable model for this purpose is found can now be answered.

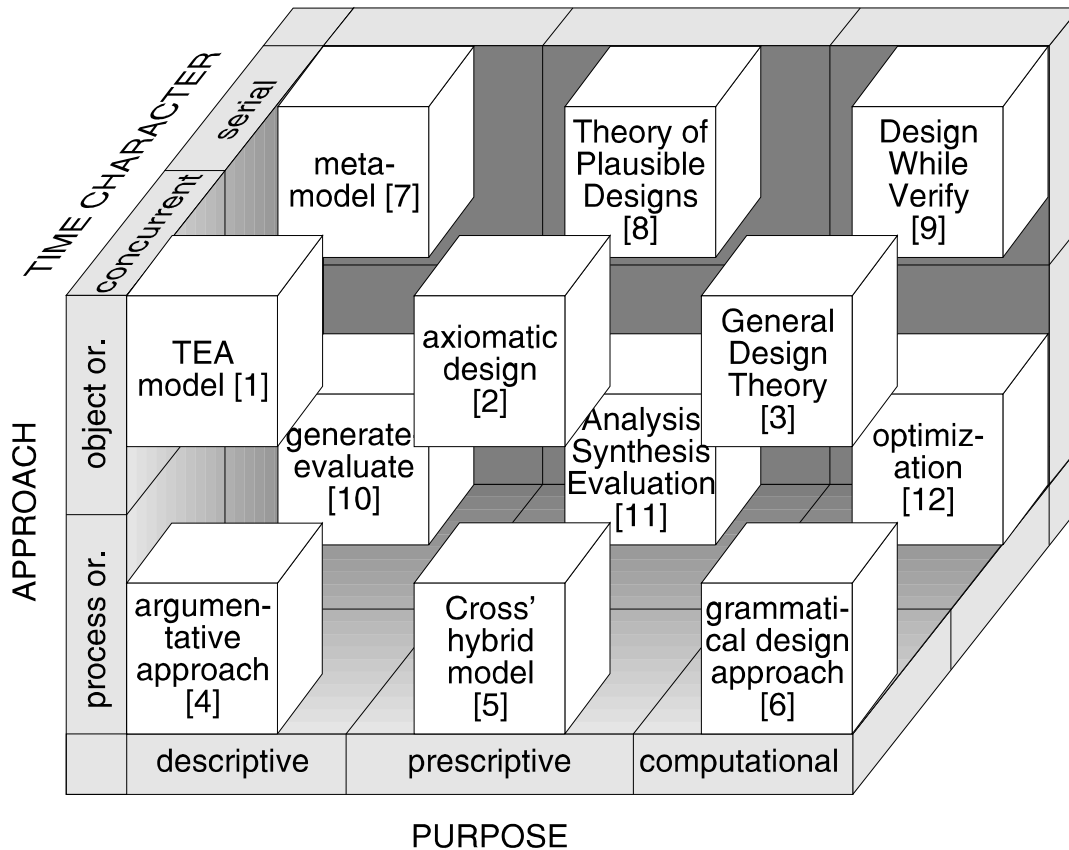


FIGURE 2.1 *Taxonomy of existing models of designing*

- |                                   |                              |
|-----------------------------------|------------------------------|
| [1] Ullman et al. (1988)          | [7] Tomiyama et al. (1989)   |
| [2] Suh (1990)                    | [8] Dasgupta (1991)          |
| [3] Tomiyama and Yoshikawa (1987) | [9] Dijkstra (1976)          |
| [4] McCall (1986)                 | [10] Darke (1979)            |
| [5] Cross (1989)                  | [11] Asimow (1962)           |
| [6] Mullins and Rinderle (1991)   | [12] Jain and Agogino (1990) |

A *descriptive* model of designing is required, because we want to explain features of the design process. Furthermore, an *object oriented* approach is preferable. The reason for this is that components and relations between components vary less for contextual differences than operations / procedures do. To see this, consider the metaphor of a play. When something unexpected takes place in a play, the script (i.e. the 'process' model) immediately fails, because the lines (the steps of the procedure) are no longer appropriate. But if the unexpected event is not too serious, the scene (the 'object' model) continues, for the actors (i.e. the objects) are able to adapt their role. In the same way, an object oriented approach leads to models of designing that are more robust to context-dependency. Finally, the time character should be *concurrent*, as a framework is needed that predicts how support should function rather than a planning when support is needed. Summarizing: a descriptive, object-oriented and concurrent model of the designing is best suited for explaining the applicability of design support and predicting what kind of support can further enhance the design process. The models

that fall in the other categories may have provided significant contributions to the field, but will not be considered hereafter, for the above reasons.

### ***The TEA model***

A well-known descriptive, object oriented and concurrent model of designing is the *TEA model* (Task/Episode Accumulation model) by Ullman et al. (1988). The TEA model explains many aspects of the design process, and it provides significant insight into the way (mechanical) designs are developed. Therefore, it will be outlined shortly hereafter. For a more detailed discussion, see the reference.

The TEA model is a model of non-routine designing, developed on the basis of an empirical study on the process of mechanical design. The fundamental components are the design state and the design operators. The *design state* contains all information about the artifact under design. The representation of the design state has one critical property, which is its *level of abstraction*. This level of abstraction is characterized by what it explicitly describes and by what it omits. In the TEA model, the continuum of this property has been divided into three levels: abstract, intermediate and concrete. Furthermore, different types of media for the representation of the design state are distinguished: “verbal | textual”, “visual”, and “physical”.

*Design operators* are primitive information processes that modify the design state, i.e. they are ‘applied to’ the design state. The TEA model contains ten operators: “select”, “create”, “simulate”, “calculate”, “compare”, “accept”, “reject”, “suspend”, “patch and refine”. Thus, in the TEA model *the design is accumulated gradually* by the incremental contributions of each operator to the design state.

A meaningful sequence of operator applications that addresses some primitive goal is called an *episode* in the model. In an episode, the decision about what operator to apply next is guided by heuristic rules, of which a set of thirteen is identified. In other words, the design process is controlled locally, within an episode. Alternative designs are only considered within episodes; by the time an episode is completed, one of the alternatives will have been accepted into the design state. After completion, the designer tackles another, closely related primitive goal in a new episode. So to accomplish a design, the design engineer performs a (large) series of episodes. Six different types of episodes have been distinguished: “assimilate”, “document”, “plan”, “repair”, “specify”, and “verify”. A non-complete set of eight rules that govern the sequence of episodes is given.

A collection of related primitive goals (and thus of related episodes) is called a *task*. Generally, a task can be described as a goal of larger scope. Four types of tasks are defined: “conceptual design”, “layout design”, “detail design”, and “catalog selection”. From the empirical study it followed that (contrary to what many models of designing propose) design tasks do not match design phases, i.e. that task-related episodes are not performed in a linear sequence.

The goals of the different tasks altogether make up the main goal comprising the satisfaction of the given design requirements. Thus a three-level goal structure is obtained: primitive goals at the episode level, sub-goals at the task level and the main goal at the design process level. Herewith we conclude our outline of the TEA model.

Application of the TEA model to the development of design support is not straightforward. This has a number of reasons:

- 1 The design state and the design operators have been characterized independently. However, there is a constraining relation between the two: the design operators modify the design state. This implies that each operator must modify at least one feature of the design state, and vice versa, for each feature of the design state that varies during the design process there must be at least one operator that realizes this. Therefore, the independent characterization that has been done is not allowed; *inconsistencies may arise* in the model.
- 2 The primary focus of the model is on the episodes, operators and their control. The *evolution of design state* that results from this has not been worked out in detail. The use of (automated) support during the design process will (by definition) have considerable consequences for this evolution. Evaluation of these consequences is hard in the TEA model, while this is a major reason for using a design process model.
- 3 The TEA model has been set up specifically for the mechanical design process, not considering group design. In the model, *no separate context-determined part and context-free part* are distinguished. This is unfortunate, as it makes reuse of the model in a different context harder. It is also unnecessary, as the model has the potential to easily incorporate this separation and maintain a large context-free part.

Therefore, a new model of designing is needed. In this model, the basic features of the TEA model, being the tasks, episodes and operators, should be maintained. To overcome the above mentioned problems, these should be combined with ideas put forward in other models of designing:

- embed the model in a world view in the spirit of Popper (1972) and Tomiyama and Yoshikawa (1987) in order to clarify context-dependency.
- characterize the main components analogously to the way it is done in the Comprehensive Design Process Model as proposed by Rivero (1977), see also David (1987). In this way, inconsistencies will be prevented and the evolutionary aspect will become naturally incorporated.

The new model is presented in the succeeding sections. It is a further elaborated and reformulated version of previous publications (De Vries et al., 1991; De Vries et al., 1993).

## 2.4 Proposition for a new model of designing

### 2.4.1 World view

Designing takes place in an environment which influences the process, it is contextually situated. Therefore, it is necessary to clarify its context. The context of our model of designing will be defined by means of a “world view” similar to the ones proposed by Popper (1972) and by Tomiyama and Yoshikawa (1987). Three worlds are considered (figure 2.2): the real world R, the symbolic world S and the conceptual world C.

The most obvious world is the *real world R*. In this world ‘entities’ exist and show a certain ‘activity’. The crucial feature of everything which exists in this world is that it can be detected by anyone who is able to perceive. Thus people, their behavior and their physical environment are part of the real world. Elements of this world are depicted by shaded rounded rectangles. (Note that ‘activity’ is meant in a restricted sense here, namely as ‘observable action’.)

The *symbolic world S* contains ‘descriptions’ of parts of the other two worlds in the form of models, laws and theories. By means of language, elements of the symbolic world symbolize (i.e. refer to) entities or concepts. A language adds semantics (i.e. a reference) to the carrier of the description, the ‘medium’. The content of a description is the same for every observer, although the interpretation might be different. Consequently, the term objective can be used for a description. For example, the content of this paper is a part of the symbolic world. A reader who knows the English language is able to recognize (observe) the content. To someone who cannot read, this is only a piece of paper with meaningless curves. Stated otherwise, the semantics are not apparent to this person and only the medium, which is part of the real world, remains. Everybody who can read this thesis sees the same words, so the content of it can be observed objectively. In the figures, descriptions are depicted as white rectangles.

The *conceptual world C* resides inside the mind of a human being. Every human being has a private conceptual world. Elements of this world cannot be perceived by anyone else; it is a subjective world. In this world ‘concepts’ of both the other worlds reside. They are formed by interpretation of observations (experience) or by assimilation of concepts already present (thinking). Concepts may drive activities and as such cause a behavior (in R), and concepts can be communicated by means of description (via S). Sticking to the same example, the reader interprets the content of this paper and thus forms a subjective concept of this description, the message. He relates this to the concepts he already had about the subject. Maybe he will conclude that the proposed ideas are totally wrong, which then is a new concept formed by inference. Nobody else is able to observe this, unless the concept is described in one way or another or causes

a particular, recognizable behavior. Concepts will be depicted as shaded ellipses from now on.

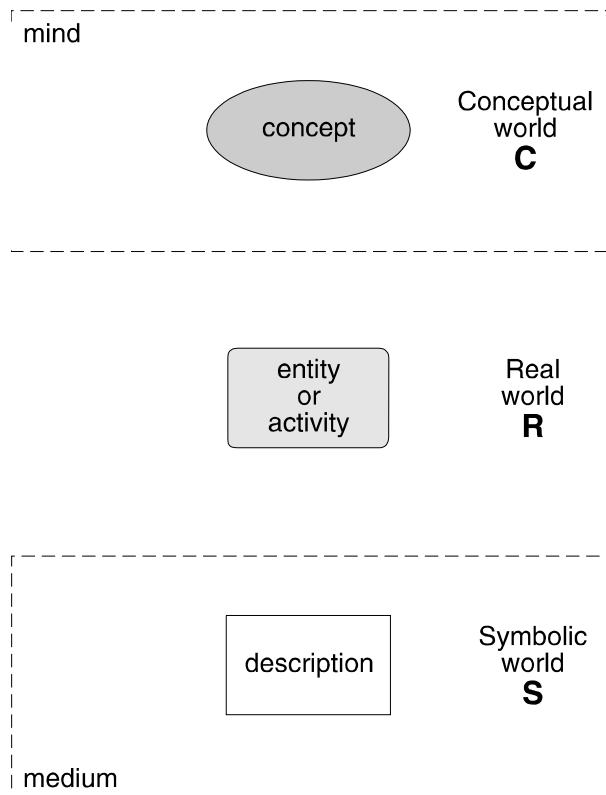


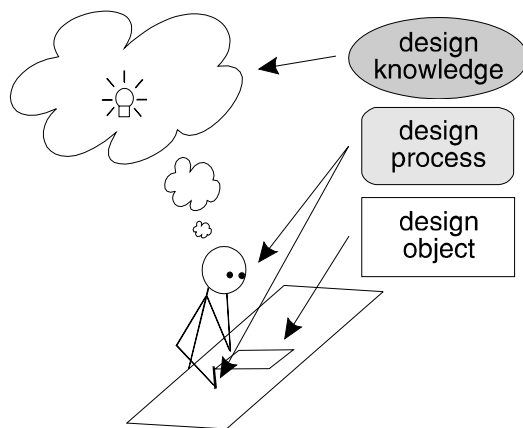
FIGURE 2.2 *World view*

It can be noted that in general it will always require an activity to form a concept of a description (i.e. observing) or to turn a concept into a description (i.e. describe). Hence, the conceptual world and the symbolic world do not have a mutual interface; the real world is always in between. The ordering of the worlds as depicted in figure 2.2 is conform this. The fact that the conceptual world is at the top and the symbolic world is at the bottom does not have a special reason; it could equally well have been reversibly or from left to right.

### 2.4.2 Basic model

In terms of the above world view, the major goal of the design process is to transform some concept of a need (in C) into a description of an artifact (in S) which can actually be manufactured into a product (in R). Consider first designing at a simple and high level, in particular an individual designer working on a design problem without any sophisticated support (figure 2.3). Then three components (i.e., 'active parts') can be distinguished that naturally correspond with the world view described above: the design object, the design process and design knowledge.



FIGURE 2.3 *Designing*

The *design object* is the symbolic world element which is generated while designing. In other words, it is the collection of descriptions that specify the artifact under design. As was noted before, the design object mostly evolves from a previous design, through a modified design specification to a complete design at the end. This evolution results from the real world part of designing, the activities performed during the *design process*. The activities are initiated and controlled, i.e. activated, by *design knowledge*. In figure 2.3, the activities of the designer are initiated in the mind, and thus all the design knowledge is part of the conceptual world. However, design knowledge may also be available in the form of descriptions, i.e. as part of the symbolic world. Two distinct parts of design knowledge can thus be separated: conceptual design knowledge and described design knowledge.

We may view a human, and thus a designer, as a goal seeking, information processing system (Newell and Simon, 1972). The condition of any goal seeking system is that it is connected to its environment through two channels: the afferent, sensory channel through which it receives information about the environment, and the efferent, motor channel through which it acts on the environment (Simon, 1981). Therefore, we can say that within the design process, the designer acts in two different ways:

- as an *observer*, with the aim to receive information
- as a *descriptor*, with the aim to produce information

Another refinement to be made is concerning the conceptual design knowledge. Conceptual knowledge in general, so also conceptual design knowledge, is located in two significantly different parts of the mind: the *subconscious* part of the mind contains the knowledge that is not explicitly and/or actively available, but ‘resides in the background’. This part of the mind has seemingly infinite storage capacity, but has relatively slow processing speed. The *conscious* part of the mind on the other hand has a limited capacity of typically seven “chunks” (Miller, 1956), but is very fast. The knowledge located here is active and/or readily available.

Figure 2.4 gives a schematic overview of the main components of designing that have been identified.

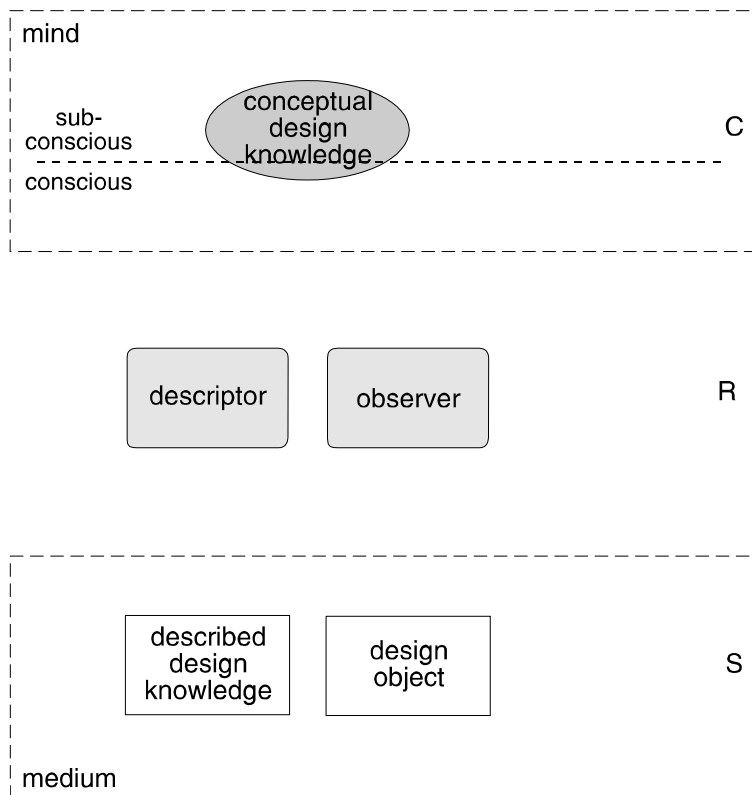


FIGURE 2.4 *Main components of the model of designing*

*Interactions* between the main components of the model are event-based, and will be incorporated in the model in the form of arrows. The following main interactions are present.

The observer observes the design object and described design knowledge through their *representations*. The *interpretation* of these observations leads to new concepts in the conscious part of the mind. Also through *assimilation* of subconscious design knowledge, new concepts become available in the conscious part of the mind. By means of *activation*, conscious knowledge initiates and controls the activities of the descriptor and the observer. The descriptor finally changes the collection of descriptions that specify the artifact under design, i.e. it causes a *modification* of the design object. Figure 2.5 depicts the interactions between the main components of designing.

In figure 2.5, only interactions between the main components of designing have been incorporated. However, also interactions will take place between the components of designing and the environment in which designing takes place. The following events are present:

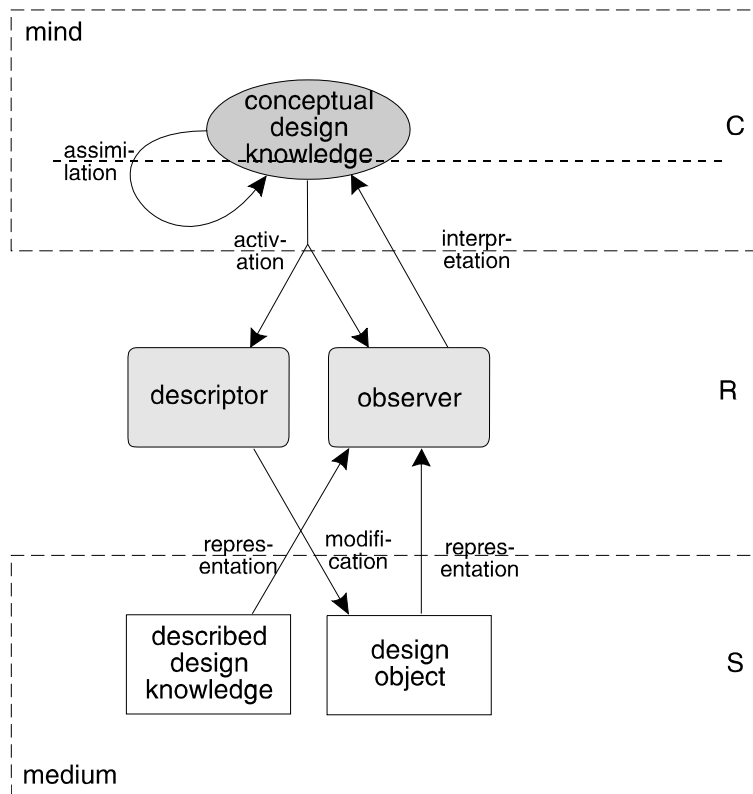


FIGURE 2.5 Interactions between the main components

- *assimilation* of subconscious general knowledge, leading to new subconscious design concepts.
- *occurrences* of entities or activities not part of the design process will be observed by the observer.
- *representations* of descriptions not contained in the design object or described design knowledge will be observed by the observer.

When these interactions are added to the model, figure 2.6 is obtained. For the simple and high level at which designing is discussed so far, this figure gives a complete description, and thus will be called the initial model of designing.

Two adaptations are needed to obtain a more relevant model of designing. Firstly, designing is almost always a *group activity*. All participants of the design group are working on (different parts of) the same design object, and in principle have access to the same sources of described design knowledge. But cooperation of multiple designers implies that there are multiple, independent observers and descriptors. Also it follows that there are multiple instances of the conceptual world, one for each designer. There is no way to be sure that conceptual knowledge about a certain matter is equivalent for two persons, although usually this will be likely to some degree. In other words, the outcome of the interpretation of an observation done by somebody is not controllable by someone else. Using the terminology of Konda et al. (1992): one can never guarantee shared meaning, even in the presence of shared memory (shared

described knowledge); one can at most expect it. Although this fact seems undesirable, it also has positive effects, as we will see later on. In fact it explains why a group is generally more creative than are individuals.

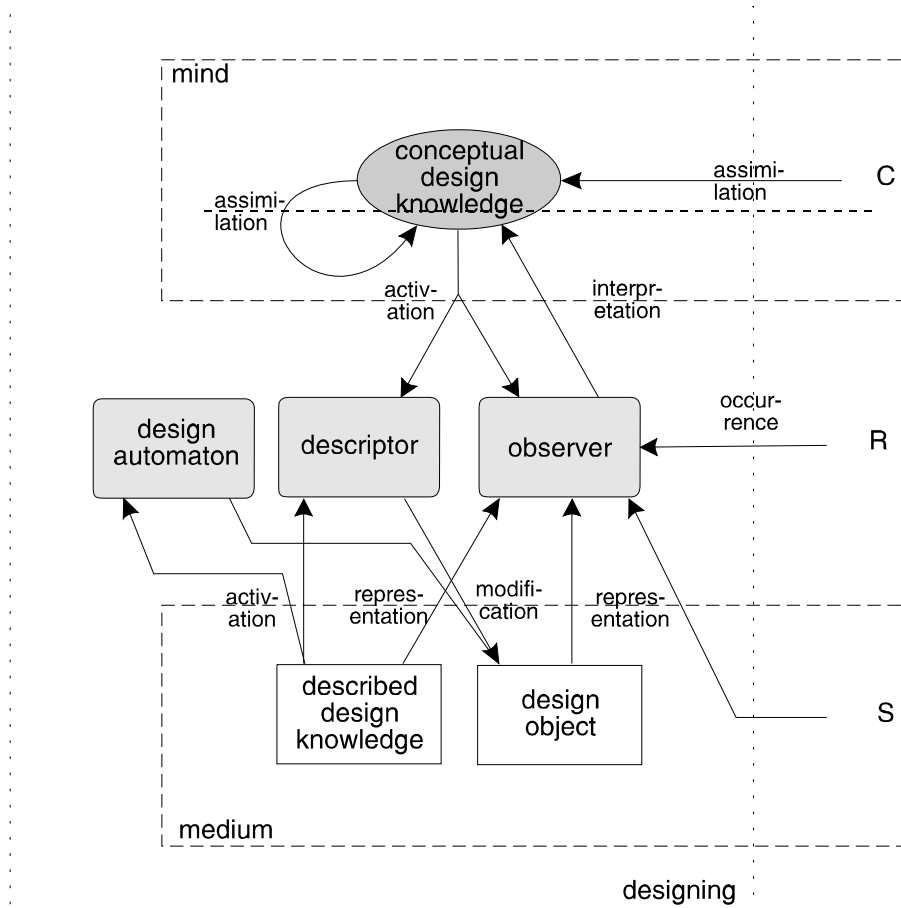


FIGURE 2.6 *Initial model of designing*

The second adaptation to be made is concerned with automated design. *Automation* requires that the knowledge needed for driving an activity is explicitly described, because only an explicitly formulated task can be executed by a computer. Full automation of design tasks thus requires that design knowledge is completely available in the symbolic world, which is generally not true. Therefore, we conclude that automated design is currently impossible without strongly limiting the problem domain. A more realistic view is to consider partial automation of designing. Activities which are not well understood and for which the driving knowledge is not (completely) known, still require human input.

With the current state of the art it is not possible to completely describe the knowledge that initiates and controls the *observer*. The underlying cognitive process of interpretation and assimilation (“creativity”) is not well understood. Therefore it is not useful to consider automation of observation actions. As a consequence, the described

knowledge in a partially automated design system will not expand automatically: the system is not self-learning in a general sense.

The *design actions* are partly well understood and formalized. They can thus be separated into two groups: automated design actions, performed by *design automatons* and completely driven by described design knowledge, and manual design actions, (partly) driven by conceptual design knowledge and performed by a descriptor as before. The question of to which degree automation can and should be done is dealt with later in this chapter.

Thus to make the initial model of designing more realistic, we should include in the model the design automatons, and multiple instances of the descriptor, of the observer and of the mind. Figure 2.7 depicts the adapted model. For reasons of clarity, we have chosen to depict the situation for the smallest group possible, namely two designers. Expansion to larger groups does not introduce new relations, although it does lead to new features of the overall process (like ‘live lock’). We will refer to this figure as the basic model of designing.

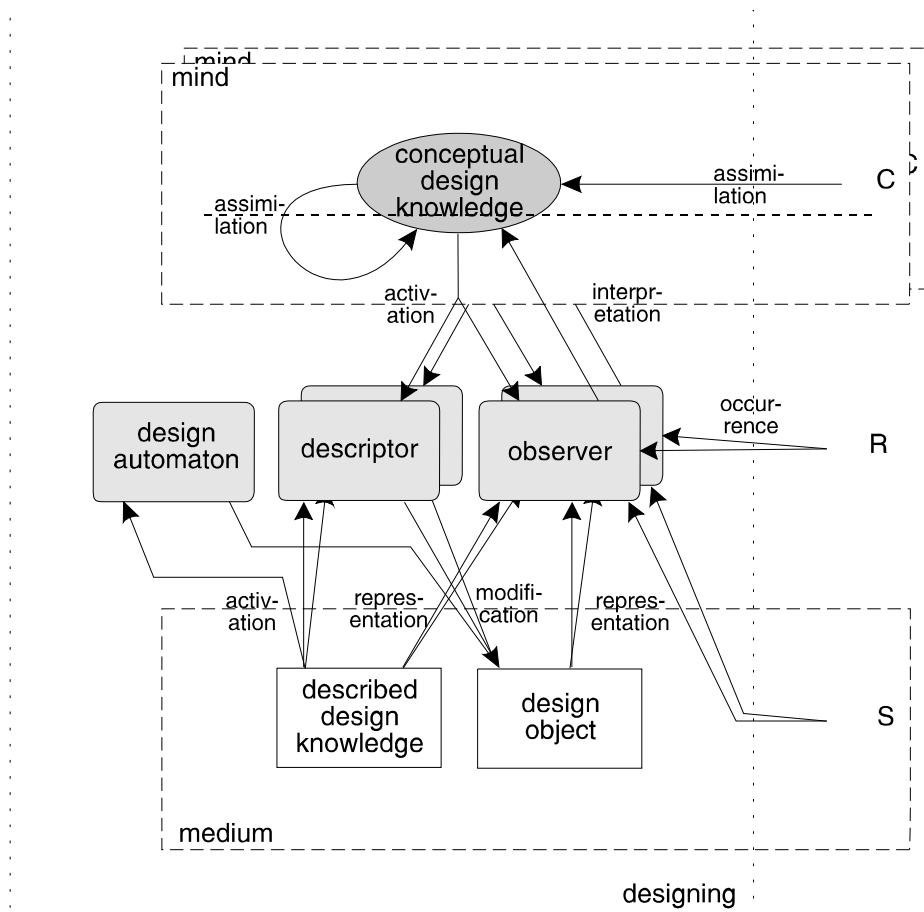


FIGURE 2.7 Basic model of designing

### 2.4.3 Adding structure

We can get a more detailed model of designing by incorporating additional features of the different objects contained in the model. As has been noted before, we cannot just consider each object separately, for there are constraining relations between them and this approach thus could lead to inconsistencies. Instead, we should select one object which is the starting point for a further characterization, and propagate the obtained features through the rest of the model. We propose to use the *design object* for this purpose, for the following reasons:

- the design object is part of the symbolic world. This world can be observed more objectively than the other two worlds, simply because it is based on a formalism. Due to this, the justification and verification of a conjectured characterization can be done more objectively in this world than in the other ones.
- models and abstractions provide the framework within which design refinements are done and thereby constrain and direct the outcome of the design process (Hoover et al., 1991). Therefore, it is logical to extend the model through the part containing the models and abstractions, i.e. the design object.
- the design object is the part of the above model that can best be characterized while not making strong assumptions on the problem context. Both the design process and design knowledge are more dependent of the class of design problems that is considered, the domain of application, etc.

So the basic model of designing is expanded by characterizing the design object. To do so, we first have to elaborate a bit on what the design object actually is. Like Ward (1989), we believe that the design object is not a collection of descriptions of single artifacts, but instead defines *sets of equivalent artifacts*. Reasoning in terms of sets is the major way designers deal with the large solution space faced during designing. Using equivalence sets, designers can reason about many individual artifacts in a very economical way. They only have to consider the features that define an equivalence set, i.e. that all artifacts that are a member of the set have in common. Because they use set-based descriptions, they do not have to think of other features, which would distract them of the problem. Set-based reasoning is their method to not over-constrain the solution space, while still having reasonably small amounts of information to deal with. The evolution of the design object then is the continuous modification of the sets that are contained in the design object. This evolution takes place by changing the features that define these sets.

#### ***Additional structure for the design object***

Among others, Ullman et al. (1988) have proposed to use the “level of abstraction” as the critical property of the design object that varies during designing. However, there is no clear characterization of level of abstraction; even more, the label covers more than one property, and is thus ambiguous. In his Theory of Domains, Andreasen (1980) made a better proposition: he separated level of abstraction into “domain” (roughly speaking the aspects that are described), degree of detail and attribution of parameters (see also Buur, 1990). This proposition will be taken over here, although a different

terminology will be used. In addition to Andreasen's proposal, the amount of different solution sets that are incorporated in the design object is also a critical property. In conclusion: there are four distinguishable ways in which the features that define the sets incorporated in the design object vary:

- *level of concreteness*. (This is what Andreasen (1980) called “domains”.) Designers consider a limited number of aspects of the design problem at a time. They do not attempt to specify what parts it should be made of right away, but first treat the problem in less concrete terms like what should it do or how should it work. Later on, other aspects such as form features are elaborated, where the freedom to define these features is limited so that the design still has the desired abstract features, i.e. still belong to the set defined earlier.
- *resolution*. The amount of detail taken into account varies independently from the level of concreteness. Initially, only main features are regarded. Details which are expected to be insignificant or less relevant are only considered later. For example, when designing a vehicle and concentrating on the functional level of concreteness, the main desired effect is the “transport” function. In addition, a function “inform the passengers about the covered distance” may be required. The concreteness level is the same, but the scope is different and more issues are regarded. As a result the structure of the design object (distinguished features of the set and their mutual dependencies) changes during the design process.
- *precision*. Descriptions of features of sets will eventually require specification of attributes (i.e. quantifiable measures). The term precision is used here for the exactness with which these attributes are specified. Designers may deal with the design problem in a qualitative sense, for example specify attributes in terms of intervals, and only state exact quantities when required.
- *number of alternatives*. Designers tend to consider several alternative equivalence sets as a solution to a problem. After comparison, they select the most promising one. In other words, the design object is not one set of artifacts, but a set of sets of artifacts. Based on the foregoing, sets are distinguished in three ways: in content (distinguished features), in configuration (dependencies between features), and in quantification (parametrization of features).

The above listed four items, level of concreteness, resolution, precision, and number of alternatives, will be called *design object characteristics* from now on.

As a means to illustrate the setup of our model of designing and its implications, we introduce the ‘design space metaphor’. The basic assumption of this metaphor is that the design object needs a four dimensional space to be characterized during all stages of designing. This space is referred to as the *design space* (David, 1987). Because it is hard to depict a four-dimensional space in an insightful way, we use a projected version of the design space in figures; the level of concreteness, the resolution and the precision are depicted along the same axis. Furthermore, we assume in the metaphor that the dimensions are orthogonal.

### ***Additional structure for the design process***

As apparent in the basic model of designing (figure 2.7), every design action causes a modification of the design object. In other words, each design action results in a change of at least one of the design object characteristics. Therefore, a design action can be viewed as a move of the design object in the design space. This implies that a certain design process manifests itself as a trajectory of the design object in the design space, see figure 2.8. Several authors have proposed similar views (Andreasen, 1980; Tomiyama et al., 1989; Ullman, 1992; David, 1987).

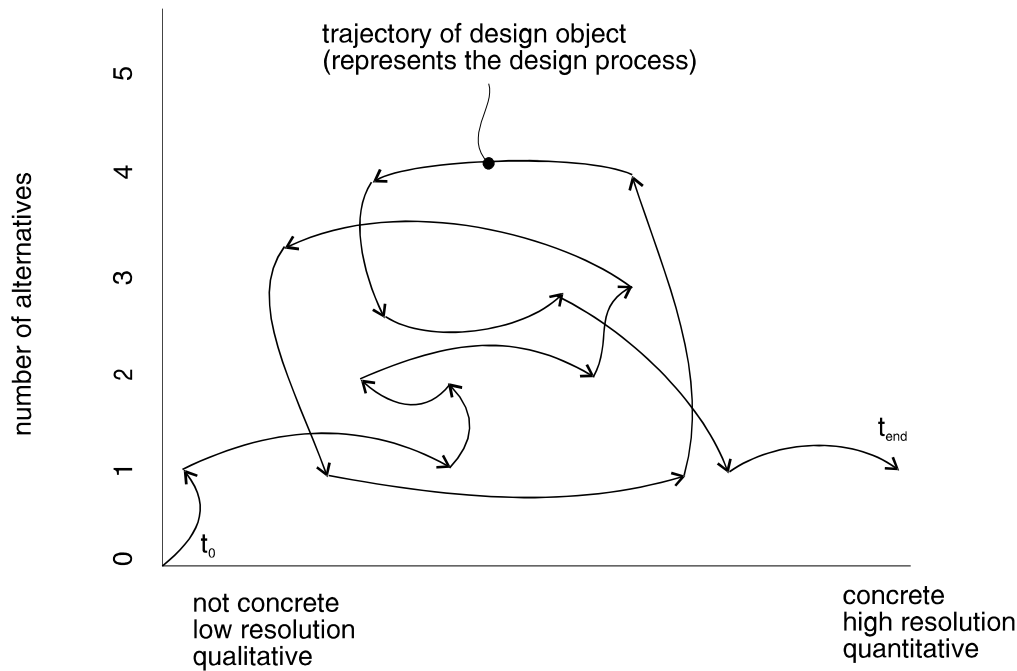


FIGURE 2.8 *The design space metaphor*

A natural way to identify elementary design operations is then to consider the set of movements along only one dimension. This set is sufficient to cover the complete design process, because every part of a trajectory in the design space can be expressed as a concurrent movement of a certain magnitude along the separate axes. With the proposed four design object characteristics, we get the next set of eight *elementary design operations*:

- concretize / deconcretize (along concreteness)
- refine / aggregate (along resolution)
- fuzzify / defuzzify (along precision)
- generate / refute (along number-of-alternatives).

The second kind of action that is performed on the design object is observation. In the metaphor, we can interpret this as looking at the moving design object from a certain position in the design space. We can imagine that we cannot see the complete space and all aspects; we are looking through a pair of binoculars, that limits our view.



However, at the same time the binoculars provide ways to control how we are looking at the design object; against which background, with which focus and what colors we can see. Based on this, we come to the following *characterization of observation actions*:

- choice of *perspective*. At least the following perspectives seem to be useful during design of controlled electro–mechanical systems:
  - an *information processing* perspective
  - a *signal processing* perspective
  - a *multi–domain power processing* perspective
- *zoom*. Within a chosen perspective, this can range from the full design object to a particular elementary part.
- *filter*. This can vary from observing all information available within the selected perspective and zoom to observing a single bit of information within the selected perspective and zoom.

#### ***Additional structure for design knowledge***

The design process is activated by design knowledge. Before activation can take place, decisions have to be taken what, when and how to activate. This is also done by design knowledge. A classification of knowledge reflecting this can be related to the structure obtained so far. Assuming that conceptual knowledge and described knowledge can be classified in the same way, we can reconsider a general categorization (De Jong, 1986) within the framework of the metaphor obtained so far:

- *declarative knowledge* concerning entities, facts, theories, conventions, principles, etc. This knowledge is the source of data about the part of the design space in which the design object moves.
- *procedural knowledge* concerning algorithms and heuristics how to perform actions. This knowledge is the operating instructions generator.
- *strategic knowledge* concerning planning, strategies and the like, to master a complex, “large scale” problem setting. This knowledge functions as the (route) planner.
- *situational knowledge* concerning recognition of the (ir)relevance of other knowledge. This forms is the supervisor of the movement of the design object in the design space.

Knowledge is the basis on which planning and control of the design process takes place. Based on the above classification of knowledge, we can therefore distinguish different modes of designing:

- 1 the *explorational* mode (or opportunistic mode, (French, 1993)); in this mode, design activities are driven mainly by design principles, i.e. *declarative* knowledge. Hence, activities are not really planned, but initiated instantaneously, on the basis of local information about the state of the design process. The random search strategy and the incremental strategy as described by Jones (1980) belong to this category.

- 2 the *systematic* mode; in this case, design activities are planned beforehand on the basis of methods, i.e. using *procedural* knowledge. The linear strategy, cyclic strategy and branching strategy that Jones (1980) distinguishes all represent this mode.
- 3 the *problem solving* mode; in this situation, actions are driven by a plan that is constructed on the basis of a guided search, and that is evaluated and revised at specific points in time. In other words, activation is done on the basis of strategic knowledge. Jones' (1980) adaptive strategy fits into this.

In all modes, situational knowledge is used to clarify how the actual problem fits into the design mode.

#### 2.4.4 Incorporating the TEA model

The TEA model basically describes information processing taking place in the mind of the designer. Therefore, its main elements can be incorporated in our model in the following way:

- the ten *operators* of the TEA model (select, create, simulate, calculate, compare, accept, reject, suspend, patch and refine) are part of the procedural knowledge.
- the *rules* that control the *application of operators* are part of the situational knowledge.
- of the six *episodes* of the model, five (assimilate, document, repair, specify and verify) result in a certain sequence of elementary design actions. Therefore, these manifest themselves in the design space as a piece of the trajectory of the design object; the piece which was traversed in the duration of the episode (see figure 2.9 ). Only one episode (plan) will not activate any elementary design action, and thus will not be visible in the design space. This is in accordance with the remark of Ullman et al. (1988) that little planning was observed in the experimental data.
- the *rules governing the sequence of episodes* are part of the strategic knowledge.
- the four *tasks* of the model (conceptual design, layout design, detailed design, catalog selection) represent a partitioning of the design space, see figure 2.9.

The TEA model suggests that mechanical design is done with a mix of the explorational and the problem solving mode.

## 2.5 Evaluation of the new model

The model proposed in here explicitly presents an overall view upon designing, which helps to prevent inconsistencies. It is a general model for engineering design; we only filled in some domain-specific features when incorporating the TEA model. We tried to capture “certain forms of invariance as a basis for relatively context independent norms” (Konda et al., 1992). In other words, we have described a pattern how context might influence the design process and how the evolutions can be given a framework. We have done this in a way so that opportunities for support can be evaluated.

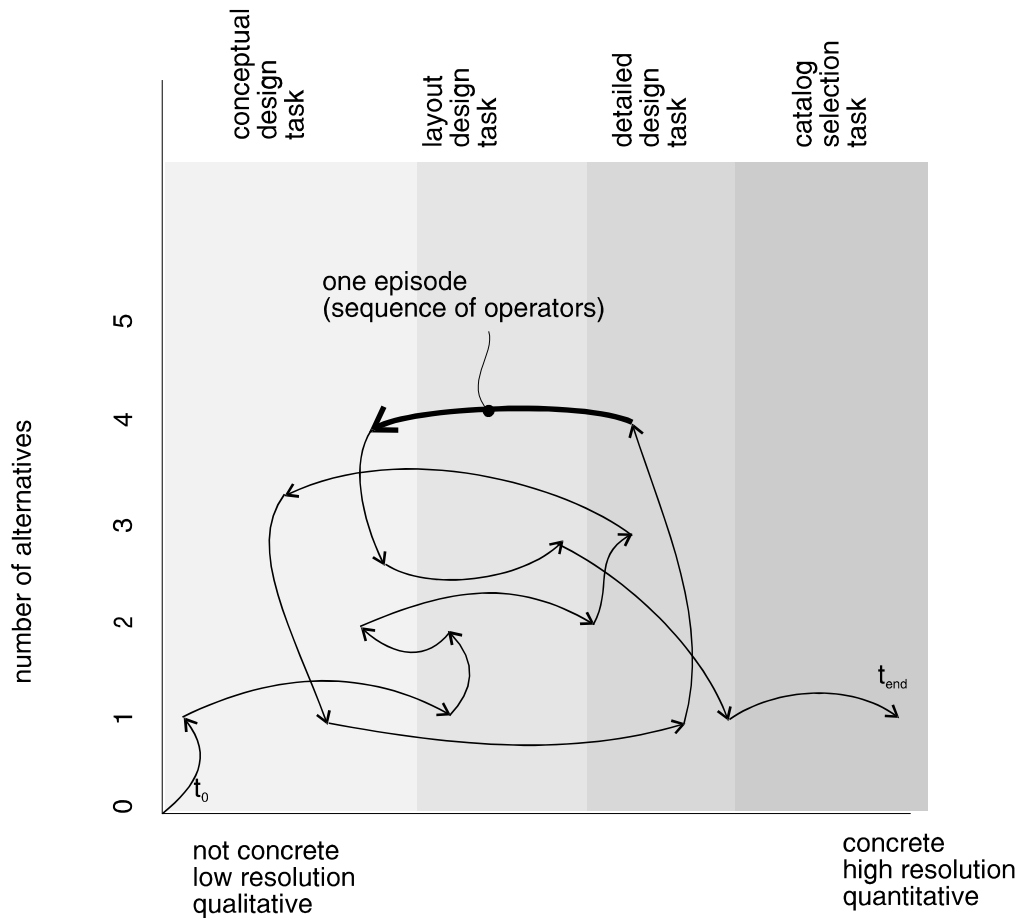


FIGURE 2.9 *The TEA model interpreted in the design space*

When actually using the model in the development of CAE systems for a certain domain, specific features need to be added. This entails four (inter-related) aspects:

- which descriptions and representations are used to cover the relevant part of the design space?
- how are the elementary design actions actually realized?
- how can observation actions be controlled?
- what specific knowledge is present?

In the next section and in the following chapters, when applying the model for the development of design support, these questions are answered for a specific application area. Hereafter, we first briefly evaluate the generic model obtained so far. This is done in three ways: by comparing our model with existing models, by indicating what explanation the model provides for the way in which designers cope with the difficulties of designing, and by identifying in what way designing can be enhanced.

### 2.5.1 Comparison to existing models

Like before, we will not treat existing models of designing individually, but instead base the comparison on the taxonomy presented in figure 2.1.

The model presented here has the *purpose* to be descriptive. Other descriptive models generally are within the framework of our model. They mainly differ from our model in that they focus on either the design process, or the design object, or design knowledge, and work out this part in more detail. Prescriptive models relate to our model by proposing a sequence of desirable states or state transitions along which the process should unfold. In terms of the design space: they advise on the ‘ideal’ design trajectory. Computational models finally describe a sequence of desirable states or state transitions that can be covered automatically by means of computation. In other words, they provide a trajectory (or a set of trajectories) in the design space that could be traversed automatically.

The *approach* we followed for our model has been object oriented. Object oriented models primarily focus on the states that are or can be present in a system. Process oriented models on the other hand mostly emphasize the state transitions that (can) take place. In the design space this can be interpreted as a (goal) location or a (next) movement respectively. Therefore, also most process oriented models can be related to our model.

Finally, we have classified models of designing in our taxonomy according to their *time character*. In general, it can be observed that the explorational mode of designing is worked out in concurrent models, and the systematic mode is elaborated in serial models. The problem solving mode is spread over both categories. Hence, we can conclude that concurrent models typically specify declarative knowledge, whilst serial models will mainly describe procedural knowledge. Either kind of models might incorporate strategic and situational knowledge. Both declarative and strategic knowledge have been included in our model, so it can be tailored to either time character.

The above leads to the conclusion that the model of designing presented in this chapter *unifies* most of the existing models of designing, even though these mutually seem to be quite different. This unifying ability can be understood because of the systematic approach followed and because of the fact that it is quite strongly based on the “unifying theme” of shared memory (Konda et al., 1992). This does not imply that the model presented here can replace existing models, which are mostly more specific and serve other purposes. Rather, the strength of this model is that it indicates what parts or aspects of designing other models leave out. Therefore, it has something to say about the applicability of these models in particular contexts.

This unification gives confidence that the model is a competent description of the design process. The model captures most of its main aspects, and it provides the ability to understand it at a deeper and less intuitive level. However, it is incomplete in

several ways. Most notable in this respect is that it explains little about how observations lead to new concepts, and how conceptual knowledge drives activities. To improve this, the conceptual world and the observer part of the model should be described in more detail. Interesting work in these directions that fits well within our model can be found in Eekels (1973).

## 2.5.2 Characteristics revisited

In section 2.2, we identified three fundamental characteristics of designing: it is context dependent, ill-structured and incomplete and it involves a time-constrained initiation of change. It is interesting to see what explanation can be given for the way in which designers cope with these difficulties on the basis of the model.

Figure 2.7 suggests that the basic strategy used to solve design problems is the following (note: the steps can take place concurrently):

- formalize (a part of) the concept of the problem
- modify, using conceptual and/or described knowledge, a (partial) solution
- observe whether the formalized problem and the formalized solution match the actual situation
- interpret and assimilate design knowledge on the basis of these observations.

This strategy takes advantage of the fact that a design problem is “ill structured in the large, but well structured in the small” (Simon, 1973). In other words, a design problem is tackled by separating partial problems from the overall, ill-structured problem, idealizing these such that they can be dealt with using formalism and available knowledge, and subsequently revising the problem on basis of the obtained insight. Two abilities appear to be essential in utilizing this strategy, see figure 2.10:

- 1 the path ‘formalize – modification – description – representation – observe’, i.e. communication (Cross, 1989; Dasgupta, 1991)
- 2 the path ‘observe – interpretation and assimilation – conceptual design knowledge’, i.e. learning (Simon, 1973; Ullman, 1992; Van Luenen, 1993).

The basic model of designing explicitly incorporates this specific form and combination of learning and communication. Indeed we believe that these two abilities enable the designer to cope with its characteristics. *Learning* is the means to deal with the lack of knowledge due to the intractability of changes, the ill-structuredness and incompleteness of design problems and the unpredictable context-dependencies. *Communication* is the way to get feedback on what has been learned, and to distribute the work over multiple persons in order to meet the time constraints and to enlarge the collective knowledge. Summarizing: learning and communication are essential design abilities in order to transfer the ill-structured overall design problem into a set of idealized well-structured, solvable subproblems, in a way that is acceptable within the context and that will lead to satisfying solutions within time constraints.

Obvious difficulties can arise when solving problems with the strategy outlined above (Simon, 1973). Interrelations among the various well-structured subproblems are likely to be neglected. Solutions to particular subproblems are apt to be disturbed at a later stage. Considerations leading to the original solutions are forgotten or remain unnoticed. This observation makes it clear why communication and learning are so important when applying an integrated design approach: the number of interdependent well-structured subproblems and solutions is much larger than with a conventional approach. This is due to the fact that with an integrated approach, domain-specific subsystems of the design object are no longer functionally and spatially decoupled, and design phases are not carried out sequentially anymore.

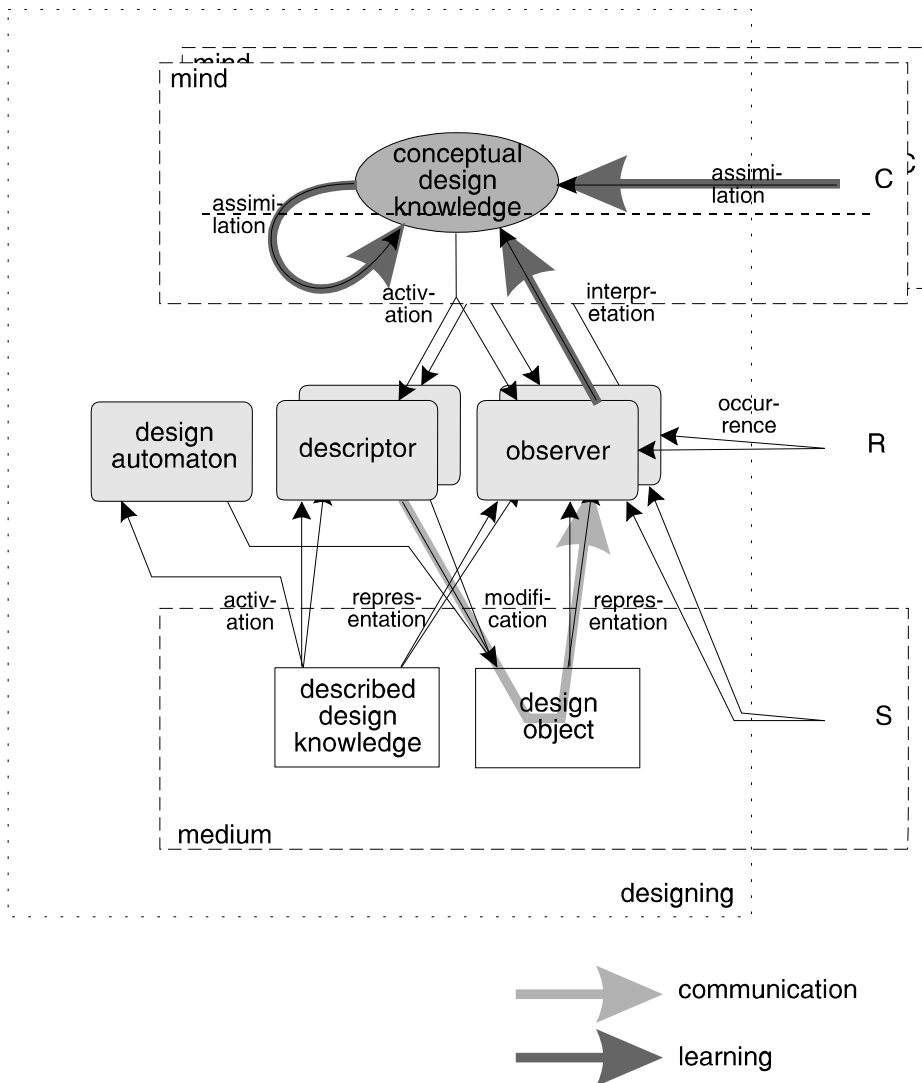


FIGURE 2.10 *The essential design abilities: learning and communication*

The model also helps to explain why practicing designers seem to design in an exploratory or problem solving mode, and less in a systematic mode. The systematic mode is not very amenable to the strategy outlined above, for it does not comply well

with learning and evolution. The fact that iterations are incorporated in systematic methods does not really solve this. This can be seen as follows. Iteration implies that because of some observed problem one returns to a point in the process that has been passed before, and continues in a different way from there. In terms of the design space metaphor (figure 2.8), this can be seen as going backwards along the trajectory that the design object has passed, and branching off along a different path from some previously passed point. This means that one throws away the investment that has been made in this part of the trajectory terms of insight. Hence, part of the learning process and of the evolutions that have taken place has to be redone. In our view, this is not what designers typically do; rather, they try to patch the design object obtained so far for the problem that has been noted. In other words, they will try to continue along the same trajectory, but probably in a different (and possibly non-goal-directed) direction. In that way, they do not need to throw away the insight that has been obtained.

### 2.5.3 Enhancing design

From the basic model of designing (figure 2.7), it follows that there are three ways to enhance designing:

- 1 *formalize design knowledge*, that is, transfer conceptual design knowledge into described design knowledge. In order to be goal-directed and effective, it is useful to separate the kind of knowledge that is formalized (i.e. declarative knowledge, operational knowledge, situational knowledge or strategic knowledge). This task entails what might be called the fundamental question underlying design research: in what way does conception take place while designing?
- 2 *automate activities* on the basis of formalized knowledge. This is a separate and relevant issue, as automation requires that formalized knowledge can be utilized computationally. It means that issues like formal completeness and decidability in finite times have to be considered.
- 3 *incorporate formalized knowledge* and automated *activities* in computer-based systems that are helpful in design practice. Making formalized design knowledge and automated activities available in a convenient way to many designers is again an issue that can best be regarded as separate from the previous two. As follows from above, the applicability of design support critically depends on whether or not it facilitates the specific learning and communication processes involved in designing. This especially holds when an integrated problem solving approach is used. Systems that support the design process should be adapted as much as possible to the designer's learning and communication habits. Only in that case it is possible to enhance designing. The example of conventional CAD systems mentioned in the introduction of this chapter illustrate this. These systems hardly help the designer gain insight into characteristics of the artifact to be designed (like behavior, appearance, performance etc.) and how these relate to the design problem. Above that, the systems require designers to communicate geometrically

in terms of lines and points, whereas communication in terms of design features or constraints is much more natural to them.

Simon (1973) proposed a general system structure that is suitable for tackling design problems, and that complies well both with our model of designing and with the strategy outlined above (see figure 2.11). The system consists of two active parts:

- 1 a (set of) *problem solver(s)* that process well-structured subproblems of the design problem. These problem solvers typically take a few arguments as input, and produce a relatively small amount of output.
- 2 an *evaluator* that compares the information contained in the immediate problem space with (large amounts of) information held in a long term memory, and modifies the immediate problem space accordingly.

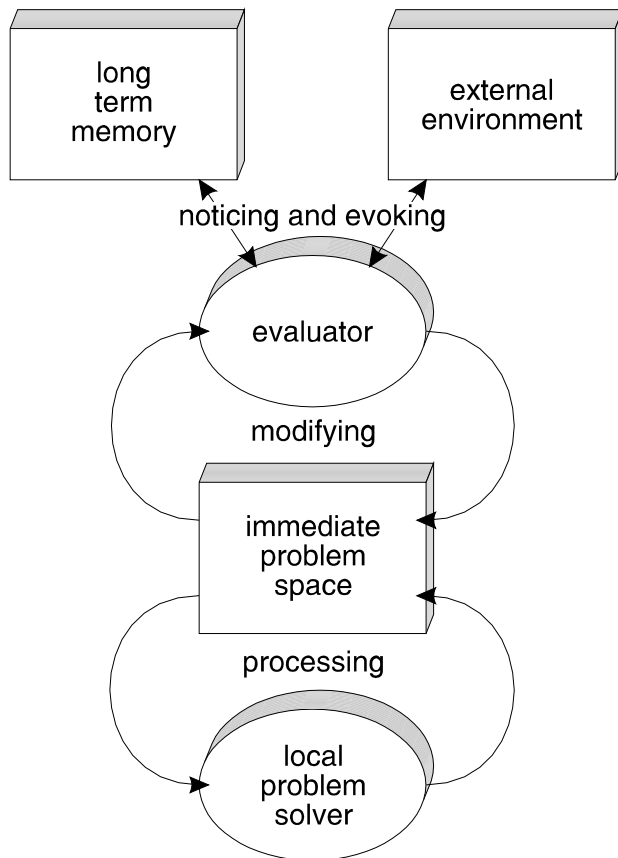


FIGURE 2.11 *Schematic structure of a system for solving ill-structured problems (Adapted from Simon (1973))*

An interesting discussion that has relevance in the light of enhancing design by means of computers is going on about which of the parts of this structure can and should be computer-based (see for instance Alberts, 1993). It can be concluded that the problem solver and the immediate problem space can be well implemented on computers, but the way in which this should be done still poses problems. Also, it seems that the long



term memory can fruitfully be located inside computer systems, although this is a major subject of research. Whether the evaluator can and should be hosted on a computer is much more of an issue.

In our view, humans are (and will be for quite some time) superior to computers when it comes to evaluating and modifying the immediate problem space. Two fundamental reasons can be given for that:

- the perceptual mechanisms that humans have enable complex perceptions that computers currently by far cannot match.
- the way in which knowledge is stored in computers (in terms of descriptions, using formal languages) inherently implies that this knowledge is limited to a context and becomes useless when this context is not present. This means that interpretation of contextually rich environments such as present in designing is not feasible.

These two reasons indicate that computers do not possess the communication – and learning capabilities that humans have, and therefore fail to tackle real life design problems autonomously. Practically speaking, this implies that designing can best be enhanced by creating systems that support designers in their communication and learning processes.

The foregoing shows that our model of designing at a general level meets the demands formulated in section 2.1: it provides an *explanation* of the applicability of design support on the basis of characteristic features of the process, and it *predicts* to some extent what kind of support would further enhance the process.

## 2.6 Developing advanced support

Above, it has been explained that integrated design is difficult, because domain-specific subsystems of the design object are not functionally and spatially decoupled, and design phases are not carried out sequentially. In other words, designing with an integrated approach introduces many additional couplings in the design object. It is the structure (or scheme, French, 1985) of the design object that specifies which couplings extend over problem domain and product life cycle phase. The structure for the design object is largely determined during the conceptual design task. Therefore, *the conceptual design task is of crucial importance* when using an integrated problem solving approach.

By taking decisions from a system point of view, as advocated by integrated problem solving, the conceptual design problem becomes considerably more complex. Hence, successfully designing with an integrated approach requires proper (technological) support for conceptual design.

For the area of controlled electro-mechanical systems, quite some design knowledge has been formalized, for instance procedural knowledge (Koster and Van Luenen,

1993), declarative knowledge (Bradley et al., 1991; Koster, 1993) and strategic knowledge (Buur, 1990). The goal of our research is to create a practical computer-based system in which this formalized knowledge might be incorporated, and eventually automate design activities based on this knowledge. As said previously, this requires the knowledge to be incorporated in the system in such a way that the users of the system, designers, will communicate and learn more effectively and efficiently.

Both in learning and communication, and therefore in design, the use of *abstractions* plays a crucial role. Abstractions are descriptions of a system in terms of a formal language that are competent for answering specific questions. In our model of designing, the importance of abstractions is clearly visible. Abstractions essentially constrain the focus and perspective of the designer through their representation. Thus abstractions provide the framework within which the evolution of the design object takes place (Hoover et al., 1991). As they say, “the quality of the completed design depends on the ability of the designer to select useful abstractions, to use them to model the performance of the design, and to use the results of the evaluation to guide further design refinements”. Hence, we may expect that communication and learning problems are caused by an improper or ineffective use of abstractions.

In recent years, substantial progress has been made in the area of supporting the ‘downstream’ tasks of the design process, especially detailed design. This progress indeed has been accompanied by the possibility to make more advanced abstractions in CAE systems, such as geometric modeling (Finger and Dixon, 1989a), parametric modeling and feature-based modeling (Salomon et al., 1993). Apparently, for these tasks support is being provided that properly uses abstractions. For the ‘upstream’ tasks such as conceptual design however, support is largely not available (Finger and Dixon, 1989a). A more concise statement of what this support should be follows by characterizing conceptual design in terms of our model of designing.

With respect to the design object:

- level of concreteness: function, behavior, initial form
- resolution: low, but with a varying structure
- precision: pre-parametric, constraints, “order-of-magnitude”
- number of alternatives: high

With respect to the design process:

- concretize/deconcretise: invoked around models describing principle solutions, to deal with the form–function dependencies
- refine/aggregate: invoked in combination with concretize/deconcretise
- fuzzify/defuzzify: present in the form of finding missing parametric constraints
- generate/refute: relatively frequently invoked

With respect to the design knowledge:

- declarative: ‘commonsense knowledge’, principles and previous experiences (i.e. ‘expertise’) are important

- procedural: little algorithms, many heuristics and implicit rules, opportunistic decision taking (i.e. ‘creativity’)
- strategic: little planning possible
- situational: initially mostly not explicitly available (i.e. ‘intuition’) but rapidly increasing (hopefully)

The characterization implies that there are many interdependent perspectives on the design object. Also, it shows that formal evaluations are hard to perform and that many ‘gut feeling’ decisions are taken that will have a major influence on the final result. Furthermore, it emphasizes that learning plays a crucial role and will mostly involve situational knowledge, and that there is little explicit knowledge. Finally, the explorational mode seems to be dominant for the conceptual design task.

As identified above, the maintenance and manipulation of models and abstractions is something that computer-based design support systems can take over from the designer. These systems can significantly enhance designing, provided that they are better suited for these tasks than designers. The basic reasons why computer based design support are *not* be better suited for these tasks are as follows (Buur and Andreasen, 1989).

- Maintenance of abstractions falls short in computer-based design support systems because they are *not formulated properly*. For example, abstractions are not specified in terms of relevant objects (i.e. objects from the problem domain of the designer), or cannot be represented in a relevant perspective. Also, the descriptions are often ‘stand alone’, and not embedded in a structure of declarative knowledge that specifies the context of different abstractions and in what ways they are interrelated. We might say that the maintenance of abstractions in CAE systems generally gives problems if the descriptions of the design object are *underformalized*.
- Manipulation of abstractions is problematic because designers *cannot modify* the design object in the way they would like. Examples of this are the restrictive, rigid sequence of invocations that systems enable, and the lack of support for converting design descriptions to a higher level of abstraction. If CAE systems lack flexibility, they limit possibilities for making design refinements, thereby missing the opportunity to enhance designing. This is caused by an *overformalization* of the design actions.

It should be noted that the overformalization of design actions is mostly a consequence of the underformalization of the maintenance of abstractions; because abstractions are not described in the proper way, it becomes possible to operate on them in ways that just do not make sense. One way of preventing that is to severely restrict the manipulations that can be done on the abstractions. Of course, a better way would be to improve the maintenance of abstractions. This clarifies that in fact the dominant problem is one in the area of *modeling*: what is the exact meaning of abstractions, and how are they treated. Therefore, we discuss the development of design support from this perspective hereafter. We propose two new concepts to improve the maintenance

of abstractions in the next chapters: one stemming from an analysis of linguistic aspects (multiple model languages, chapter 3 and 4), and one from an analysis of implementational aspects (polymorphic modeling, chapter 5).

## 2.7 Conclusions

Three characteristics of designing are crucial for understanding and supporting it:

- design is *context dependent*.
- design problems are *ill-structured and incomplete*.
- design involves a *time-constrained initiation of change*.

Because of these characteristics, design has to be viewed as a contextually situated, evolutionary process. A descriptive, object oriented and concurrent model of the design process is best suited for explaining how designers cope with these characteristics. Analysis of an exemplar from this category of design models, the Task/Episode Accumulation model (Ullman et al., 1988), shows that application of this model to the development of design support is not straightforward, because:

- inconsistencies may arise in the model
- the evolution of the design state has not been worked out
- no separate context-dependent and context-free part are distinguished.

To overcome this, a new model of designing is needed. The main structure of the proposed new model relates the design object, the design process and design knowledge (figure 2.7). It clarifies that designing involves a specific form and combination of learning and communication.

There are four distinguishable ways in which the features that define the sets of artifacts incorporated in the design object vary:

- 1 level of concreteness
- 2 resolution
- 3 precision
- 4 number of alternatives

A useful metaphor of a four-dimensional design space in which the design object evolves during designing follows from this. Using this metaphor and the main structure of the model, the design process was characterized and design knowledge was classified. Finally, modes of design were identified. When actually applied in the development of CAE systems for specific domains, additional features need to be added to the model.

The model explicitly presents an overall view upon designing, which helps to prevent inconsistent or incompatible assumptions. It describes a pattern of how context influences the design process and how evolution can be given a framework.

Opportunities for support can be evaluated. The model is of a unifying nature: it is able to embrace most existing models of designing, and provides the ability to understand designing at a deeper and less intuitive level.

There are three different ways to enhance designing: by formalizing design knowledge, by automating design activities on basis of formalized knowledge and by creating practical computer-based systems that incorporate formalized knowledge and automated activities. Enhancing designing by means of practical systems requires systems that support designers in their communication and learning processes.

The conceptual design task is of crucial importance when using an integrated problem solving approach. Concepts that will aid the designer in maintenance and manipulation of abstractions are needed. Therefore, development of design support is discussed from the perspective of modeling hereafter.



## Multiple model formulations

### 3.1 Introduction

To communicate and to think about a physical system, either existing (in case of modeling) or to be created (in case of design), people use models of this system. What can be expressed in these models is determined by the language in which they are stated. For it is the language that specifies what terms can be incorporated in a model (the *vocabulary*), and how these can be meaningfully combined (the *grammar*). The *expressiveness* of a language is large if it has a large vocabulary and an unrestrictive grammar. Note that in terms of the model of designing formulated in chapter 2, the expressiveness relates to the size of the design space that can be covered with the language.

Along with expressiveness, the language also determines how a model is to be encoded (the *notation*). Finally, there is generally some sense of good and bad layout acquainted with it (the *style*), although this may not be formalized and may to some extent differ from person to person. The coding and layout of a model, i.e. its *representation*, critically determine the ease with which the contents of a model is observed and interpreted. A language is more *powerful* if it is readily interpreted, i.e., if its representation is clear, insightful and unambiguous. From the point of view of the model of design, we might say that a powerful language provides a well-chosen, limited perspective and an illuminating filter on the design object.

When synthesizing or analyzing a model, one has to select a language in which to formulate the model. This would not be an issue if there were a language that is both expressive and powerful. However, it appears that a trade-off has to be made (Mullins and Rinderle, 1991); because the larger the amount of aspects and features that can be expressed in a language, the less clear and insightful and the more ambiguous the representation will be in general. For example, the expressiveness of mathematical equations is large, but it is often difficult to estimate specific characteristic features of the system on basis of such a description (Van Dijk, 1994). Conversely, graphical representations like the Bode plot or the  $s$ -plane can only express specific aspects of linear dynamic systems, but have proven to be powerful in the design of control systems (see e.g. Dorf, 1989). Therefore, the choice of the language in which to formulate the model is an important issue. It has a critical

influence on the communication and learning processes in which the model is taking part (Popper, 1972). Unfortunately, only few model builders realize this.

As both communication and learning are central to the use of an integrated design approach, it would be wise to select the most powerful language which is available to formulate the model in. However, as has been discussed in chapter 2, a designer has to concurrently consider the design at different levels of abstraction, ranging from not concrete, with a low resolution and a low precision (generally at the system level), to concrete, with a high resolution and a high precision (at the component level). This requires the ability to inspect the design object from *multiple perspectives*, using *different filters* and a *varying focus*. This holds more when the design project is done by a team comprised of designers from various disciplines and backgrounds, which is typically the case when dealing with mechatronics and concurrent engineering (Buur and Andreasen, 1989). Therefore, it is impossible to find one single language that is both powerful and appropriate to formulate the model in. If we find an appropriate language, it will be one that is expressive, but not very powerful (natural language for example). Using such a language is not a feasible solution; it will work counter-productive while designing, as it will enlarge communication and learning problems.

An approach that can circumvent this is to use *multiple formulations of a model* (Tomiya et al., 1989; De Vries et al., 1992; Roozenburg, 1993). Then a *set* of languages can be selected which individually are powerful in a specific area, and together provide all the expressiveness and representations that are needed. Multiple languages provide powerful formulations of models without sacrificing expressiveness. In fact, this is nothing new for designers; the iconic diagram typically used during conceptual design is a formulation different from the solids and views formulation of detailed design, which is again different from the feature-based descriptions applied during process planning. What is new in the above proposition however, is to apply these multiple languages *simultaneously* instead of in a phased manner. In case of an integrated design approach, the simultaneous formulation of the design object in multiple languages may even be indispensable (Bradley and Buur, 1993).

Simultaneous application of multiple model formulations introduces the need for establishing and maintaining relations between the different formulations of a model. Modifications of the model should principally be possible in any of these formulations. This implies that bi-directional conversions between the different formulations should be available. These should preferably be instantaneously realizable. Unfortunately, conversions are generally not straightforward, time-consuming and error-prone, if available at all. Therefore, design would be supported by a system that provides for *automatic conversions* between appropriate model formulations. Such a system would release the designer from the distracting and difficult task of converting between formulations, and shorten the time needed for conversion while reducing the risk of errors.



For restricted areas, systems featuring automatic conversions between model formulations have been realized, such as in software engineering (Interactive Development Environments, 1991) and in linear control system design (e.g. Grace et al., 1990; Van den Bosch, 1989). However, the multiple model formulations required when designing controlled electro–mechanical systems cover a larger range of aspects, which might be referred to as technical aspects of physical systems. Due to that, the languages involved will be more different from each other in terms of expressiveness. Hence, the conversions will probably be more difficult.

In the next section, 3.2, requirements for and general characteristics of appropriate languages for conceptual design of controlled electro–mechanical systems are identified. The selected set of formulations is presented in section 3.3. In section 3.4, we address the issue of how to set up an extendible system to support the simultaneous use of these languages. Realizability of the resultant design proposal is evaluated shortly in section 3.5. In the proposed set–up, the conversion from one model formulation to another goes through a central model, the core model. The way in which this core model can be described is worked out in section 3.6. Finally, section 3.7 lists conclusions.

## 3.2 Languages for conceptual design

Taken strictly, the foregoing implies that a lot of languages are necessary during (conceptual) design. Of course, that is not the intention of the statement made about using multiple languages: we should use as *few* languages as possible. Specifically, we should not use both of two languages if the perspective and/or expressiveness they offer differ only a little. This implies that the selection of languages should be done carefully, such that the most appropriate set is chosen. Appropriate in this sense implies three things:

- 1 the languages that are included are *suitable*, i.e. they are tailored to usage during conceptual design of controlled electro–mechanical systems;
- 2 each of the languages is considered to be *necessary*, i.e., each language has a specific area in which it is significantly better applicable for synthesis or analysis than any other;
- 3 the set of languages is *sufficient* for this task, i.e. the complete set together can cover all aspects of interest during the conceptual design of controlled electro–mechanical systems.

In order to determine whether or not a language is appropriate for conceptual design of controlled electro–mechanical systems, it is necessary to identify the characteristics that are required of a language for this task. This is done in section 3.2.1. Based on this, a general characterization of model formulations that comply to this is given in section 3.2.2.

### 3.2.1 Requirements

Based on the characterization of conceptual design as done in section 2.6, and on conclusions of relevant publications (Hogan, 1987; Libardi et al., 1988; Rinderle et al., 1989; Akman and Ten Hagen, 1989; Wijbrans, 1993), we have come to the following requirements for the languages:

- the *representations* should be graphical, e.g. networks, iconic diagrams, mnemonic diagrams or labeled graphs. This is because graphical representations are more informative and easier to interpret. Several reasons for this can be given. Graphical representations depict relations directly, which seem to be the terms in which people think (Simon, 1981). Textual representations on the contrary need to be processed (‘read’) before a relation is explicit. Also, all relations in a graph are observed immediately due to the pattern recognition abilities of humans. Non-graphical representations only allow sequential observation of multiple relations. Finally, the topography of a diagram contains extra information that is not present in a textual representation.
- the *expressiveness* has to include all the terms, principles and laws (i.e. all abstractions) that are theoretically sound and well-established in the relevant problem domains. Preferably, the grammar should inherently capture the ‘laws’ as much as possible. Then it is impossible to specify models that violate these laws.
- the *perspectives* that need to be provided upon the integrated system description are those of information processing (mostly for the software part of the system), signal processing (electronics, control), and multi-domain power processing (electric, mechanics).
- in order to support *zooming in and out* while observing the design object, the organization of the model should take the form of a part-of hierarchy. This means that a component (process) can be made up of lower level components (processes), and conversely, an aggregation of components (processes) can be dealt with as one higher level component (process).
- the *aspects* that can be covered should be function, behavior, and configuration.
- the *partitioning* of a model should be as modular as possible. In this context, modularity refers to the fact that the description of an aggregation in which a model fragment is embedded does not change when the internal structure of the fragment changes, or conversely, that the description of an internal structure of a model fragment does not change when the aggregation in which the fragment is embedded is changed.
- it should be possible to *characterize* models in a qualitative or parametric way.

### 3.2.2 General characteristics

Model formulations that follow the above listed requirements will be hierarchically organized network models. Such models contain four main concepts that can be generally characterized, see figure 3.1:

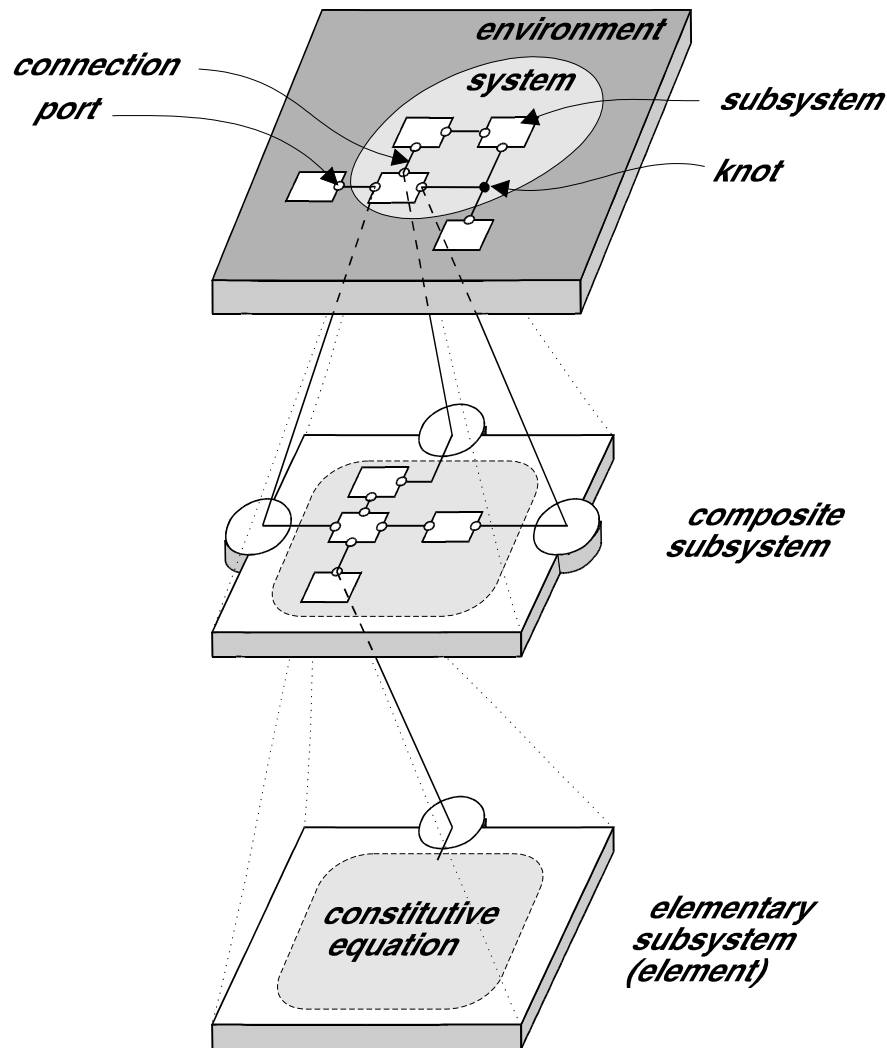


FIGURE 3.1 *General characterization of model formulations considered in this thesis*

- *subsystems*. A subsystem symbolizes a process or an object. Hence, states (in the sense of attributed variables) and operations can be acquainted with a subsystem. In hierarchically organized networks, two kinds of subsystems can be distinguished:
  - *composite subsystems*. These subsystems are specified in terms of a (sub)network.
  - *elementary subsystems*. These subsystems are specified in terms of equations.
- *connections*. In the formulations considered here, a connection symbolizes a set of constraints, namely that the variables on either side of the connection are pairwise equal. Stated differently: a connection does not specify any explicit variable or operation, but merely defines that a variable of the subsystem on one side is equal to a variable of the subsystem on the other side. It should be noted that most of the formulations considered hereafter have directed connections. One usually associates a flow with such directed connections.
- *knots*. A knot is a joining of one end of more than two connections at a point where there is no subsystem. Hence, a knot does not represent a process or an object, but rather the splitting or fusion of connections.

- *ports*. A point where a connection can enter or leave a subsystem will be signified with the term port here. Hence, a port links a connection outside the subsystem to a connection inside the subsystem. This makes clear that ports are the elements that connect two layers in the model hierarchy. Also, it clarifies that ports do not specify any operation, but that they do specify variables, namely those with which a subsystem interacts with other subsystems in its environment, i.e. interaction variables. What makes a port semantically meaningful, is that in a port restrictions are defined for the attributes of its variables. Therewith, ports give complete control over interfaces of subsystems (see also section 5.6.1). It should be noted that this usage of the port concept does not match the port concept as used in electric circuit diagrams; what is called a port here is called a terminal there. We come back to this later.

In the next section, where specific languages are discussed, we characterize languages by further determining the character of subsystems, connections and ports.

### 3.3 Selection

Among others, Buur (1990), Welch (1992) and Bradley and Buur (1993) have identified and compared languages that seem suitable for conceptual design of controlled electro–mechanical systems. We shall not repeat these discussions; rather, we present a selection of these (or similar) languages that we consider appropriate. During the characterization of the selected languages, it becomes clear what the strong points of these languages are.

The goal of conceptual design is to find the *configuration* for the design object such that it will *function* properly. Or stated in transformational terms, the aim is to convert a *functional* description into a *schematic* description. So at least a functional and a schematic formulation will be needed during conceptual design. These are selected in section 3.3.1 and 3.3.2 respectively. In section 3.3.3, these two formulations are compared and it is concluded that an intermediate formulation is needed. The selected intermediate formulation is discussed in section 3.3.4, and the overall selected set is evaluated in section 3.3.5.

For illustration purposes, one example system is depicted in each of the selected formulations. This system consists of a dc–motor, that is fed by a dc power supply through a safety switch. The motor drives a gearbox that is connected to a flywheel (the load).

#### 3.3.1 Functional formulation

A *functional description* of a system is a description of *what* it can do (or rather should do), without specification of how this is to be done. A functional description needs to incorporate the following (Buur, 1990):

- which processes can take place in a system

- in which states a system can be, i.e. which combinations of processes can be active simultaneously
- which state transitions can take place
- what are the conditions under which these transitions take place
- what is the role of time in this, e.g. with respect to synchronization and timing.

Languages for functional descriptions stemming from the mechanical design community typically only address the first issue. Consequently, these languages have found very limited usage, and are not considered here. More suitable are the various languages that are used in computer science (Bradley and Buur, 1993), such as the Modern Structured Analysis notation (Yourdon, 1989) and Timed Petri nets (Peterson, 1981). These languages have a broader expressiveness, but still emphasize one or more of the above issues. Recently, Wijbrans (1993) has defined a combination of these languages, called the THESIS formalism. It can be regarded as a variant of the Modern Structured Analysis notation, where (among other extensions) possibilities for describing control, concurrency and event handling have been improved. An illustrating example of a model in this formalism is given in figure 3.2. This notation is the most powerful functional formalism that is available.

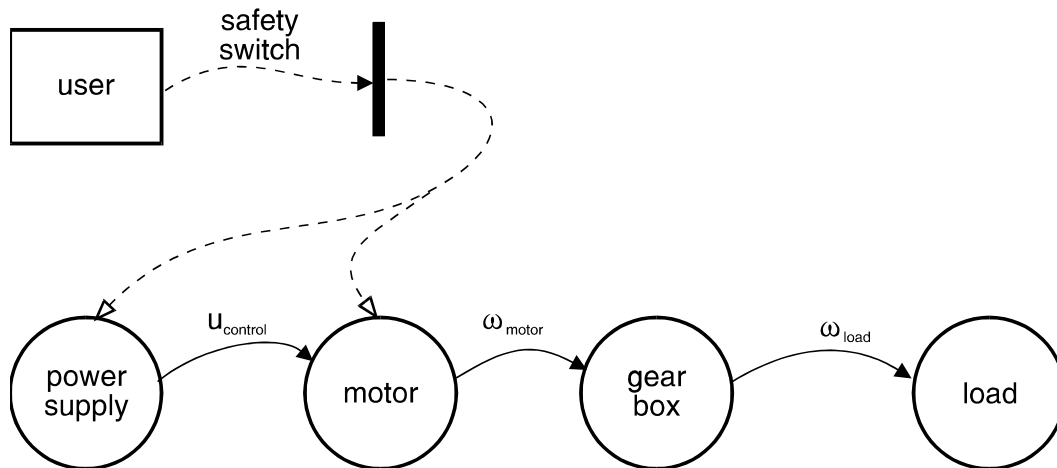


FIGURE 3.2 Example model in the THESIS formalism

A port in a THESIS model specifies either an input or an output variable of its subsystem. Ports have no explicit representation in THESIS. Three different types of input/output ports are distinguished:

- *data ports*: for variables that represent information to be processed
- *control ports*: for variables that represent events
- *activation ports*: for variables that represent enable/disable flags.

A connection denotes that a variable of an input port equals a variable of another output port. Consequently, we can say it describes a variable flow that can be of three different types: *data flow*, *control flow* and *activation flow*. Also, it follows that in

THESIS there are two types of knots: *merge points* and *split points*. The subsystems describe the transformation processes working upon the port variables, and can be of four kinds:

- *terminators*: processes not belonging to the system but interacting with subsystems of the system.
- *data processes*: normal operations that are performed on data inputs to obtain data outputs. An activation input may specify whether the process is active or not. Data processes can be specified by e.g. block diagrams, mathematical formulas and algorithms.
- *control processes*: finite state machines that operate on control inputs and produce control outputs and activation outputs.
- *stores*: processes that do not perform any operation, but merely copy input to output.

Figure 3.3 shows the symbols used for the various types of connections and subsystems. Appendix C gives a more elaborate overview of the formalism.

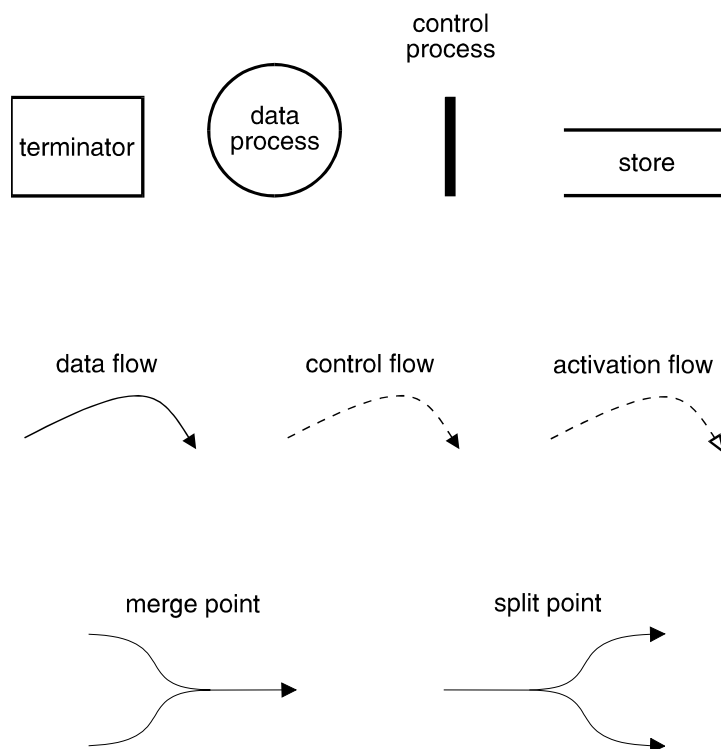


FIGURE 3.3 THESIS symbols

The THESIS notation has been defined for the purpose of structured development of embedded control systems (i.e. computer-based closed loop controllers that are integrated in the systems they control). Consequently, this notation emphasizes the information processing perspective. To improve that, the semantics of data ports, data flows and data processes should be extended such that it covers not only data variables, but any kind of functional variable. If this modification is made, THESIS

complies well with the above formulated requirements. It is only a slight modification; a change of mindset rather than an actual redefinition.

### 3.3.2 Schematic formulation

A schematic description of a system is made to determine how the system works, without being explicit about the form in which this happens or the processes that result. A schematic description specifies a configuration of (conceptual) objects or processes that realize some effect(s). Typically, such models are stated in terms of a diagram of interconnected icons. Hence, we will use the language of *iconic diagrams* to obtain schematic formulations. Figure 3.4 shows the example system as an iconic diagram.

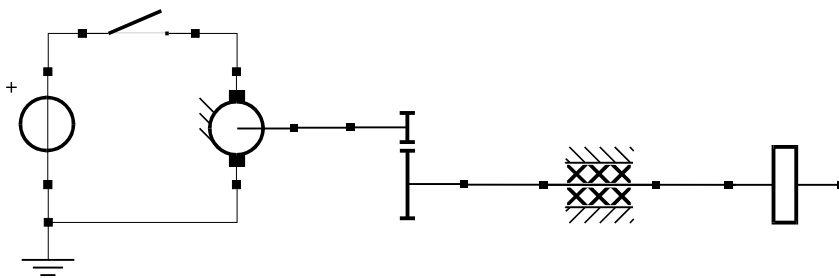


FIGURE 3.4 Example system in the iconic diagram formulation

Icons are mostly discipline-related. For some physical domains, like the electrical, hydraulic and pneumatic domain, (inter)nationally standardized icons are available for the basic subsystems of these domains (e.g., IEEE, 1987; NNi, 1991). For multi-domain subsystems or subsystems in domains that have no generally accepted standards, ‘local’ standards are developed usually, or else new icons are postulated instantaneously. In appendix A, an overview is given of the icons used in this thesis and their meaning.

The fact that icons are not fully standardized has some important implications. First, there is no static set of icons in the vocabulary of an iconic diagram language, unlike for example in THESIS. If new components are synthesized, new icons generally need to be devised. Secondly, iconic diagrams are to some extent context-sensitive, that is, their exact interpretation may be dependent of the problem context or the social context. And thirdly, there are few formal rules available for iconic diagrams. From this we conclude that iconic diagrams are an *intuitive* rather than a formal language. In other words, conceptual appeal of the representation is of more importance than exactness or unambiguity. This might well be the reason why iconic diagrams are frequently used during the conceptual design task: vagueness and some ambiguity can lead to “creative” interpretations, and this language stimulates that. Iconic diagrams comply well with the requirements formulated for conceptual design languages, except for the fact that they are usually not hierarchically organized. Extension of the language such that this is improved is straightforward; it requires the introduction of

icons for ports, such that these can be incorporated in an iconic diagram as to specify aggregate subsystems. The extension has been done for the domains considered here in appendix A.

As can be expected from its intuitive character, the semantics of the subsystems in iconic diagrams is not fully agreed upon. One view is that they describe the ‘initial form’. However, an iconic diagram often contains elements that are only conceptually present, as an effect, and not physically, as a component. Usage of the term initial form suggests that all subsystems of the model are physical, which is not the case. Therefore, this view seems incorrect. A second option is that the elements describe ‘design features’ (Salomons et al., 1993). Unfortunately, this does not help much, as the term ‘feature’ itself has no clear meaning; it has been used for many things related to a design (Finger and Dixon, 1989a). In a modeling context, iconic diagrams have been used to depict ‘Ideal Physical Models’, i.e., models that describe domain–attributed, lumped, idealized physical phenomena (e.g. Shearer et al., 1967). In this interpretation, an icon represents a *conceptual* phenomenon by means of a sketch of a *tangible* counterpart with a behavior in which the modeled phenomenon is dominant. This means that in an Ideal Physical Model, a physical effect is described, and a possible form to obtain this effect is indicated, but not determined. In other words, in such a model it is not explicitly made clear whether a phenomenon describes a component or merely an effect that is incorporated inside a component. This is a way of looking at the matter that matches the purposes of designers. However, Ideal Physical Models are built from a restricted set of subsystems, namely the basic building blocks of the involved physical domains. The iconic diagrams considered here do not comply to these restrictions, for icons that represent composites of basic building blocks can be present, and also artificial phenomena such as information processing. The interpretation should be widened somewhat; iconic diagrams describe interconnected, domain–attributed, lumped *phenomena*, either idealized or composite, and either physical or artificial. Hence, an iconic diagram is a *phenomenological* model formulation.

For most physical domains, a special kind of iconic diagram knot is defined that is called *global reference* here. Figure 3.5 shows the icons for global references of the domains considered in this thesis. Like other knots, a global reference does not specify a physical phenomenon; rather, it defines that a certain physical variable of the involved domain is set to zero at its location (e.g., the electrical voltage, the rotational and/or translational speed, the hydraulic pressure). The domain–independent term that is used here to signify this type of physical variable is *across–variable* (Shearer et al., 1967). Across–variables are used in conjunction with through–variables. A *through–variable* is the generalized term to signify the physical variable of which the product with the across variable defines a power (e.g., the electrical current, the mechanical torque and force, the hydraulic fluid flow).



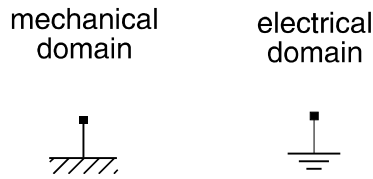


FIGURE 3.5 *Icons for global references of the mechanical and electrical domains*

Some remarks have to be made with respect to this terminology. Across variables and through variables originally have been given an operational definition (Firestone, 1933). An across variable has been defined as the power variable that can only be determined comparatively in an actual physical system, by measuring at two points across a system part. The through variable has been defined likewise; when a measurement device is inserted at one point in a connection of an actual physical system, it is imagined that this variable can be measured because it flows through the device. Obviously, the operational definition of across variables already fails for the global references; these do not specify a difference. The second and more fundamental problem with these terms is that they become confusing when used in a generalized thermodynamic framework (Breedveld, 1984, page 130). However, in here we will not employ this framework, and we prefer not to introduce new terminology if existing terminology can capture what is meant. Therefore, we maintain the terms across and through variables, but have introduced them differently to solve for the operational definition matter.

In an iconic diagram subsystem, ports are represented by means of fixed points in the icon, mostly located at the border. The points generally are chosen such that they have an actual connection counterpart in the tangible component that the icon sketches. Hence, we can conclude that iconic diagram ports describe component connectivity. Because of the fixed location, the ports of a subsystem in an iconic diagram are called *terminals*. Inspection of some common icons (see figure 3.6) shows that there are three major kinds of terminals:

- *paired terminals*, e.g. with the electric inductance (figure 3.6a) and the diode (figure 3.6b). As the name already suggests, paired terminals are always coupled pairwise. These terminals specify an across variable with respect to each other, and a common through variable. (Note that in electric circuit diagrams, a terminal pair is said to form one port. Hence, this port concept differs from the one used in this thesis). Paired terminals can be further divided into the following subtypes:
  - *symmetric* paired terminals, e.g. with the electrical inductance. In this case, the two terminals cannot be differentiated. That is, interchanging the connections of both terminals does not alter the model.
  - *asymmetric* paired terminals, e.g. with the diode. For such a terminal pair, it does make a difference when connections of both terminals are interchanged. Hence, the two terminals of the pair can be differentiated. Properly designed icons for subsystems having such terminals always contain an indication on

- basis of which the differentiation is possible. Conversely, it can be noted that if such an indication is present, terminals of this type and subtype will be present.
- *autoreference terminals*, e.g. with the translational mass (figure 3.6c). These specify an across variable with respect to a global reference, and a through variable. Often, the icon of a subsystem with these kind of terminals incorporates some image of a reference, see e.g. the rotational friction (figure 3.6d). However, the representation of the translational mass shows that this is not always the case. Multiple autoreference terminals are coupled if they represent a common across variable. This is for example the case for the translational mass and the rotational friction. If multiple coupled autoreference terminals are present, then one of these terminals will specify a through variable with respect to the other terminals.
  - *signal terminals*, e.g. the middle terminal of the electric current meter (figure 3.6e). These terminals specify either an across variable, or a through variable, or some other variable.

From the inventory of terminal kinds it follows that there are two different kinds of connections in an iconic diagram:

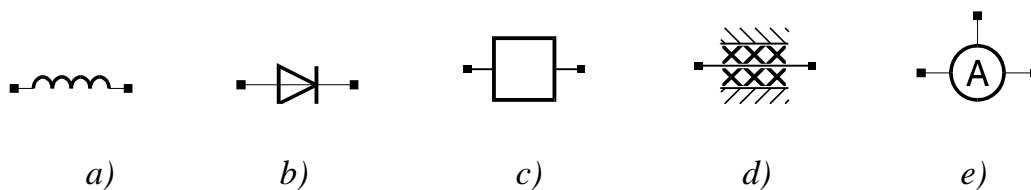


FIGURE 3.6 *Icons with different kinds of terminals*

a) *electric inductance*

b) *electric diode*

c) *translational mass*

d) *rotational friction*

e) *electric current meter*

- *power connections*. These are connected to non–signal terminals. They specify two constraints: one for the across variables, and one for the through variables of the terminals on either side. Hence, if the graph is fully specified and correctly defined (that is, it contains all global references needed to determine values for all across variables), a power flow can be acquainted with each connection of this kind. Power connections are non–causal in iconic diagrams. Moreover, the power flow direction is mostly not shown.
- *signal connections*. These connections are connected to signal terminals, and specify only one constraint. Consequently, a signal flow can be acquainted with this type of connection.

Knots finally are generally represented in iconic diagrams by means of a dot. In electrical circuit networks, knots are called nodes.

An important characteristic of an iconic diagram is that its structure incorporates *invariants*, such as Kirchhoff's laws. These invariants are based on conservation

principles that hold for physical systems. Two types of invariants can be distinguished (e.g., Shearer et al., 1967):

- the *continuity requirement*. This requirement states that through variables sum to zero at a knot. Kirchhoff's current law and Newton's law for equilibrium of forces at a point are instantiations of this requirement.
- the *compatibility requirement*. This requirement specifies that along a closed path of connections and paired terminals, across variables sum to zero. A path that has a global reference at both ends is also considered to be closed. This implies that this invariant is always *non-local* (i.e., not concentrated in one network element), but rather spread out over a substructure of the network. Kirchhoff's voltage law is the electrical representative of this requirement.

An iconic diagram is suitable for providing the perspectives of signal processing, multi-domain power processing and configuration.

### 3.3.3 Comparison

So far, two languages for conceptual design have been proposed: the THESIS formalism, and iconic diagrams. An indication for the decisions and difficulties that are involved when converting a model from one formulation to the other can be given by the essential differences that exist between the model formulations. These are shown in table 3.1. The differences follow directly from the above discussion.

Each of the entries in the table differs considerably for a THESIS model or an iconic diagram. This indicates that *many decisions are involved* when converting one formulation into the other. Even more, when performing the conversion in one step, there are *no ways to evaluate intermediate results*, because the languages do not give the possibility to formulate these intermediate results. In other words, an incremental approach is impossible in this way; all decisions have to be taken jointly. This is undesirable in evolutionary processes such as modeling and designing.

A solution to this would be to use some kind of *intermediate* language, i.e., one that lies between a functional model formulation and a phenomenological model formulation. Publications on automated conceptual design of physical systems have indicated that the *bond graph* language (Paynter, 1961; Breedveld et al., 1991) is appropriate for this (Ulrich, 1988; Finger and Rinderle, 1989; Rinderlee and Subramaniam, 1991; Redfield and Krishnan, 1992; Sharpe and Bracewell, 1993). We follow these indications and select bond graphs as an intermediate model formulation. The bond graph language meets the requirements that have been formulated in section 3.2.

	THESIS model	iconic diagram
1. variables	functional variables	physical variables
2. connections	causal	non-causal
3. ports	inputs / outputs	component connectivity
4. subsystems	transformation processes	physical phenomena
5. structure	no invariants, reference free	(non-)local invariants, with references

TABLE 3.1 *Essential differences between a THESIS model and an iconic diagram*

### 3.3.4 Behavioral description

In figure 3.7, the example system is depicted in bond graph terms. The bond graph language is a well-defined, formal language suitable for describing non-domain-related *energetic effects* of physical systems. These effects fully define the dynamic behavior of these systems. Hence, bond graphs can be regarded as a *behavioral* model formulation. It is presumed that the reader is familiar with the language, so the characterization of bond graphs is kept short. The subsystems of a bond graph describe either non-idealized (i.e. composite) physical phenomena or idealized elementary energetic processes. Contrary to iconic diagrams, this difference is explicitly represented in the bond graph formulation: idealized elementary processes are depicted by means of mnemonic codes, whereas composite physical phenomena are represented by means of ellipses, see figure 3.7. (This is a particularly clear reason why the combined usage of bond graphs and iconic diagrams is useful.) Subsystems can have two types of ports: power ports and signal ports. Power ports specify both an effort variable and a flow variable, whereas signal ports only specify one variable, which may be an effort or a flow, but can also be a mathematical variable. The connections, called bonds, indicate the (power or signal) flow between the subsystem ports. The half arrows of the power bonds indicate the positive power flow orientation. Two types of knots are defined in the bond graph language: the zero junction and the one junction. These junctions implement domain-independent generalizations of Kirchhoff's laws, and hence represent the continuity and compatibility requirements. The perspectives that are offered by bond graph models are those of signal processing and multi-domain power processing.

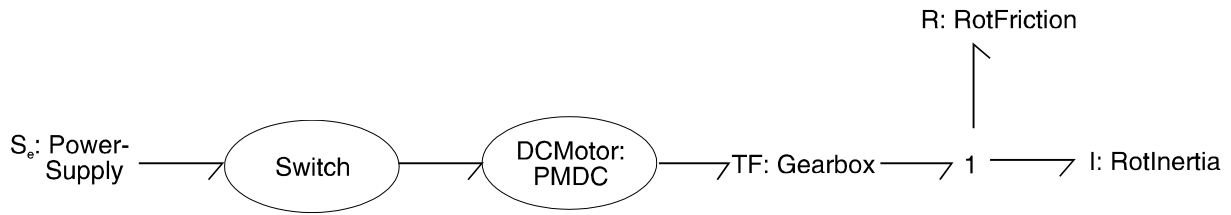


FIGURE 3.7 *Bond graph formulation of the example system*

The connections in a bond graph can be both causal or non-causal. A non-causal bond graph model can be augmented with causality on the basis of well established rules (Karnopp and Rosenberg, 1968; Breedveld, 1986; Van Dijk, 1994). A port of a bond graph subsystem incorporates the variables with which the subsystem exchanges energy with its environment. Consequently, a bond graph port allows calculation of a generalized *impedance* or *admittance*, i.e. it specifies a dynamic interface. Invariants are captured locally in the bond graph, in the junctions (although not always, see Hogan and Fasse, 1989). Mostly, the choice is made to not include global references in the structure of bond graphs. Background of this is that global references do not contribute to energetic behavior, and are not needed in a bond graph for purposes of definition (as in iconic diagrams). Hence, leaving global references out will reduce complexity of the graph. Also, leaving them out will generally reduce the number of causal problems (Van Dijk, 1994).

### 3.3.5 Evaluation

Based on the foregoing observations, table 3.1 can be augmented with the bond graph formalism. The result of this is shown in table 3.2. This table suggests that bond graphs indeed are an intermediate formalism; for each of the entries, the bond graph characteristics are more concrete than those of a THESIS model, and less concrete than those of an iconic diagram. Furthermore, when converting a functional model into a phenomenological model or vice versa, the bond graph language can cover ranges of intermediate models in two important ways: from fully causal to fully non-causal (for the connections), and from fully in terms of idealized elementary processes to fully in terms of components (for the subsystems). Therefore, it can be concluded that the bond graph language is a good intermediate between a functional and a phenomenological formalism.

The three model formulations selected so far are all *necessary* during conceptual design of controlled electro-mechanical systems; each has a specific area in which it is more suitable for proposing and/or evaluating decisions to be made about the integrated system than the others. Without doubt, the set is not *sufficient* in general, as a designer of controlled electro-mechanical systems will soon encounter situations in which other model formulations are needed. This will especially concern model formulations that are discipline-specific. A feasible approach to solve for this is to strive for a set that is sufficient as far as *inter-disciplinary* formalisms are concerned, and provide import- and export links from and to discipline-specific formalisms. In

this sense, the set identified above does seem to be a reasonable starting point. The set of three formulations will also suffice to demonstrate and evaluate the principle of multiple model formulations. Therefore, additional model formulations will not be considered.

	THESIS model	Bond graph model	iconic diagram
1. variables	functional data	energetic effects	physical variables
2. connections	causal	(non-)causal	non-causal
3. ports	inputs / outputs	dynamic interface	component connectivity
4. subsystems	transformation processes	physical phenom. / ideal processes	physical phenomena
5. structure	no invariants, reference free	local invariants, reference free	(non-)local inv., with references

TABLE 3.2 *Essential differences with a bond graph model as intermediate*

## 3.4 Design of the system set-up

### 3.4.1 Main design issue

When simultaneously applying multiple model formulations, one model is formulated and can be altered in multiple ways. For example, the models of figure 3.2, figure 3.4 and figure 3.7 will be at the disposal of the model builder simultaneously. Either of these formulations can be chosen to modify the model at any time. There are two central problems to be solved when setting up a support system for this:

- how to keep different formulations of one model *consistent*, i.e., such that they indeed describe the same model. For example, the power flow specified in a model should be a common part of both the bond graph and iconic diagram representation and therefore has to be included consistently in both formulations.
- how to keep different formulations of a model *tractable*, that is, to provide the user with the ability to manipulate the model in a predictable way, without losing oversight what the current model constitutes, how features relate to each other in the different formulations, and how a desired modification can be made.

The main system design issue underlying this is where the information defining the model is stored and how it is updated. There is a range of options for this. One extreme

is to store the model completely distributed, i.e., separately for all of the different

formulations. The other extreme is to centralize all this information, i.e., to have one integrated object, called the *core model* here, in which the information for all model formulations is stored. Figure 3.8 shows an impression of both of these solutions.

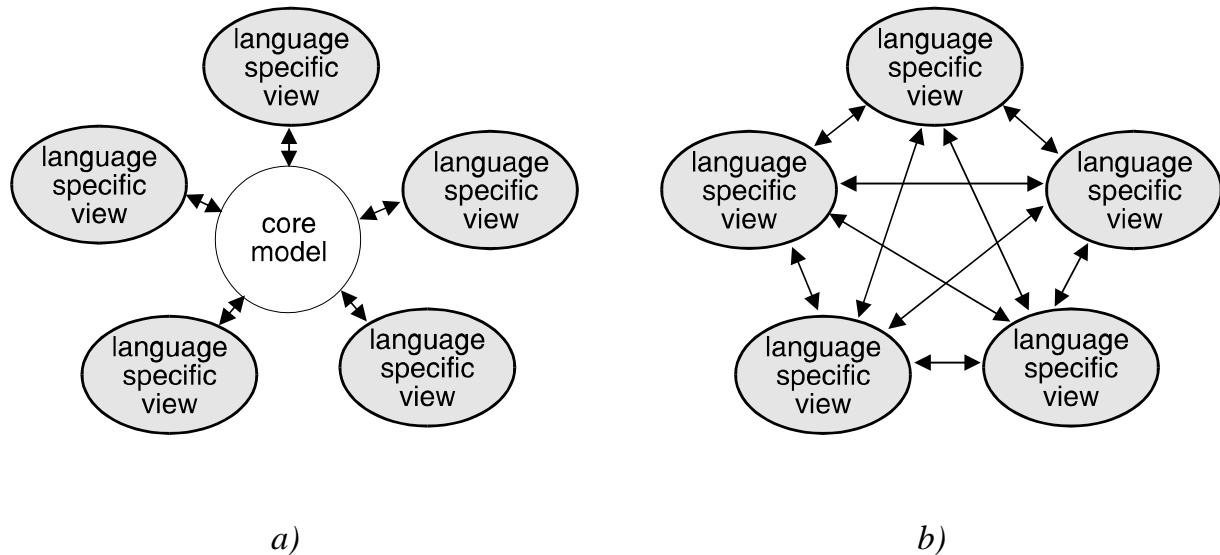


FIGURE 3.8 *Extreme solutions for model storage:*  
 a) *completely centralized*  
 b) *completely distributed*

It is not a trivial matter to say which of these two options is closest to the optimum; many aspects of system design are related to it, while on the other hand predictions of consequences are hard to make. For three reasons, we have decided to start from the centralized set-up:

- *conceptual clarity*. The conversions are more rigidly structured in the centralized solution. This means that the end user of the system will have less problems with understanding how the system works. Also, it will force system developers to discuss conversions and storage formats for models at a more fundamental level.
- *extendibility*. Above, three model formulations have been selected and it has been discussed that addition of others is likely if additional design tasks are considered or the domain of interest is changed. Therefore, extendibility is an important issue. Suppose we have  $N$  model formulations, and want to add a new one. In case of the distributed solution, the total number of (uni-directional) conversions that has to be added then is  $N(N - 1)$ . For the centralized solution, this number is equal to  $2N$ . Hence, these numbers are the same when  $N$  equals 3 (i.e., our situation). For  $N$  larger than 3, the centralized solution is preferable.
- *maintainability*. The network specifying the dependencies between model formulations is smaller and simpler in the centralized case. This will most likely improve maintainability of the system.

We assume thus that there is one central description, the core model, that stores all information defining the model. The remainder of this section addresses the question *what* this information should be, and *how* this information is changed during conversions. The resultant system set-up should be optimized for consistency and tractability of the model.

### 3.4.2 Conversions

#### *Protection*

The concept of multiple model formulations inherently implies that there are multiple editors available in the system, each providing a partial view upon the core model and enabling manipulation thereof. Without special measures, every operation that can be done on the core model is available in each editor. Consider the next example. Causality analysis is an important tool for evaluation of a model before simulation (Van Dijk, 1994). Without special measures, it would be possible to assign causality to the core model by issuing this command in the iconic diagram editor. However, the result of this operation cannot be evaluated in the same editor, as iconic diagrams do not explicitly show causality. In other words, the core model can be changed without direct notification to the user. This implies that the operation is not tractable, unless an editor for one of the other formulations is also inspected. Such a situation is dangerous, and should be avoided. The language specific model upon which an editor provides a view should be protected such that only operations that have a visible effect are available. This is not a restriction of model manipulations that the user of the system can do; it merely means that the places from which this can be done are restricted. So conversion from the core model to a language specific view (or vice versa) should at least consist of the addition of a *protection* to a language specific model.

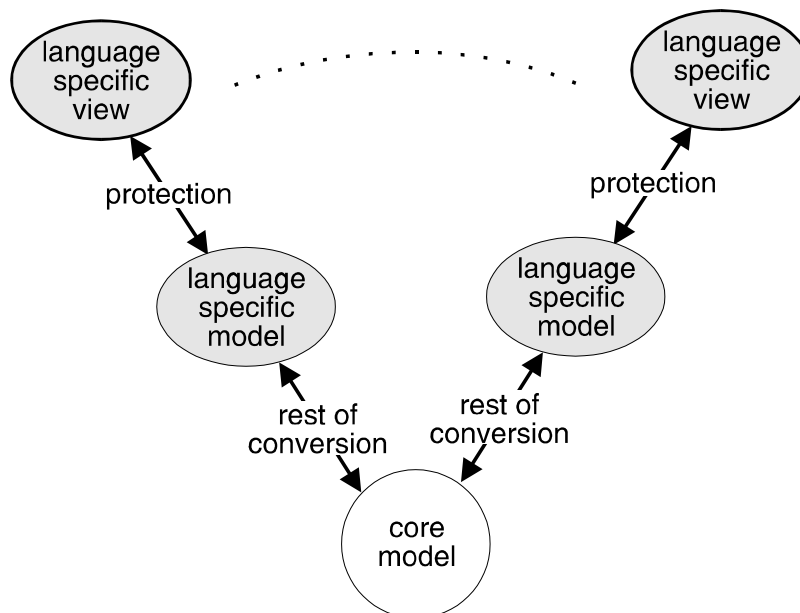


FIGURE 3.9 *Step 1: Protection*



### Visualization

When one model is formulated in two languages, the notation and layout (representations) generally are different, although they may define the same properties for the model. For example, the way in which a motor is depicted (figure 3.10) or the topographical position in the graph are generally different for a bond graph formulation and iconic diagram formulation of a model in which the motor is incorporated. In formal terms: one formulation of a model is (at most) isomorphic to the other, rather than the same.



FIGURE 3.10 *Isomorphic formulations of a motor*

It is possible to separate *representation-specific data* of a model from information that should be the same for all model formulations, i.e. *intrinsic* model properties. There is no need to keep all representation-specific data of one formulation inside the core model, as it is not of concern for other model formulations. Even more, assuring consistency between different model formulations is done much more securely and faster if the information that needs to be checked is minimized. Therefore, representation-specific data should be stored or generated separately from intrinsic model properties. In other words, a textual language specific description should be maintained that holds all intrinsic model properties. Graphical notations and topographical positions should be added to this description at conversion time, to obtain the language specific model. I.e., conversion from the core model to a language specific view requires a *visualization* phase before the protection (figure 3.11).

### Translations and transformations

Above, it has been shown that the semantics of the port-concept depend on the language; in THESIS a port specifies an input or output, in a bond graph it specifies a generalized impedance or admittance and in an iconic diagram it specifies component connectivity. An important consequence of this is that multiple formulations of one model are not isomorphic: either the subsystems are different (the *decomposition* differs), or the structure of the model is different (the *relations* between the subsystems differ), or *both*. An electric resistor in an iconic diagram for example has two connections, whereas in the bond graph formulation this element has only one bond connected to it. Of course, the parts that are distinguished and their structure bear some dependency, but they are generally not isomorphic. In other words, intrinsic model properties such as the structure depend on the formulation of the model, and thus we have language specific description that should be coordinated.

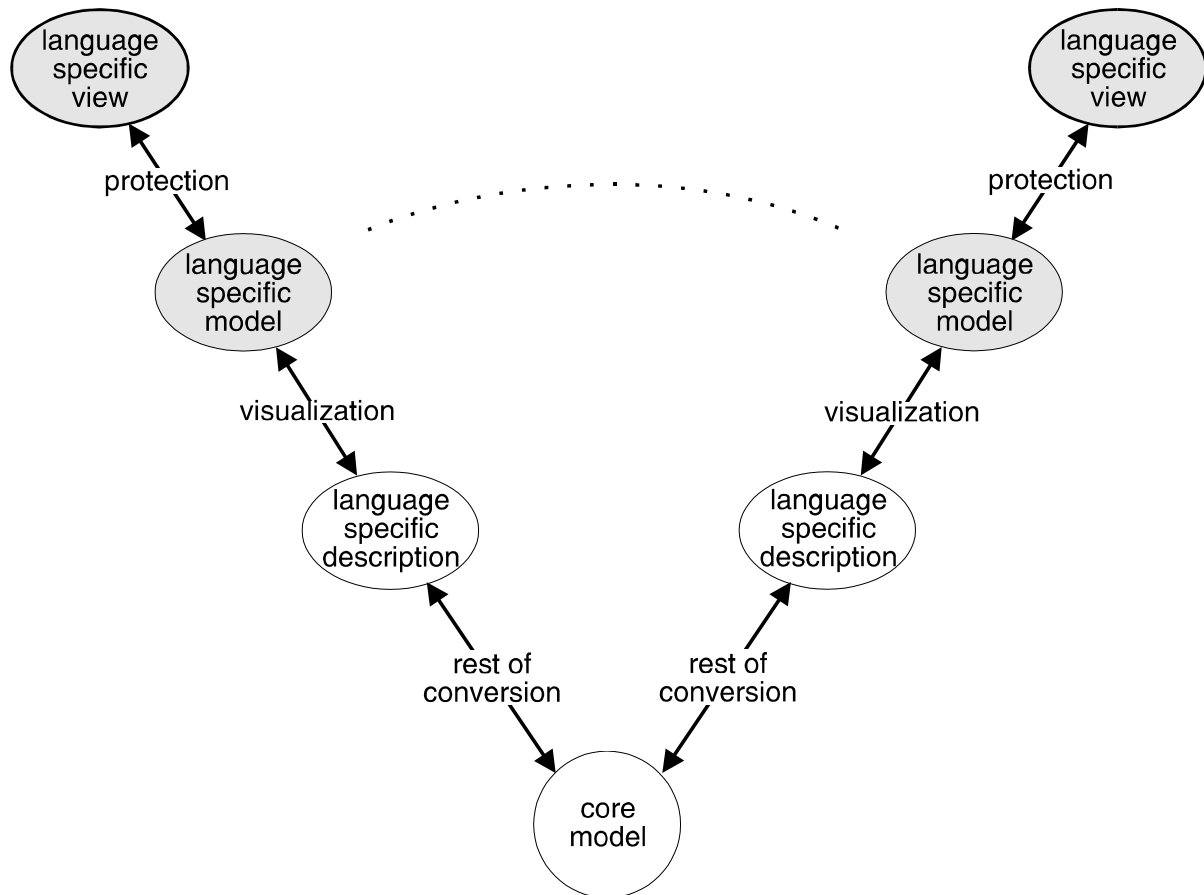


FIGURE 3.11 *Step 2: Visualization*

A way to prevent this is to devise a language-independent core model, through which coordination of separately maintained language specific descriptions is enforced (Tomiya et al., 1989; Aksit and Bergmans, 1992). For this solution to be feasible, the core model should contain all information that is needed to obtain any of the language specific descriptions, that is, it should integrate all information defining the actual model. So the ‘rest of the conversion’ in figure 3.11 should consist of a *translation and transformation* from core model into language specific view; the subsystems need to be translated, and the model structure needs to be transformed.

Based on the foregoing considerations, we propose to set up a system featuring multiple model formulations as depicted in figure 3.12.

Our proposition is in fact a specific implementation of the metamodel framework, a concept introduced by Tomiyama et al. (1989) that has gained considerable attention. Tomiyama et al. (1989) formulated three functions that this concept can serve. The first function is integration of several models. In our realization, this role is taken by the core model. The second function is to model physical phenomena in order to make different theories underlying different models compatible. The translations and transformations provide for this in the proposed system set-up. The final role mentioned by Tomiyama et al. (1989) is that of serving as a tool for describing the

evolving design object. Although the metamodel is necessary to enable this, we show in chapter 6 that the framework alone (and hence the proposed system set-up) is not sufficient to realize this function.

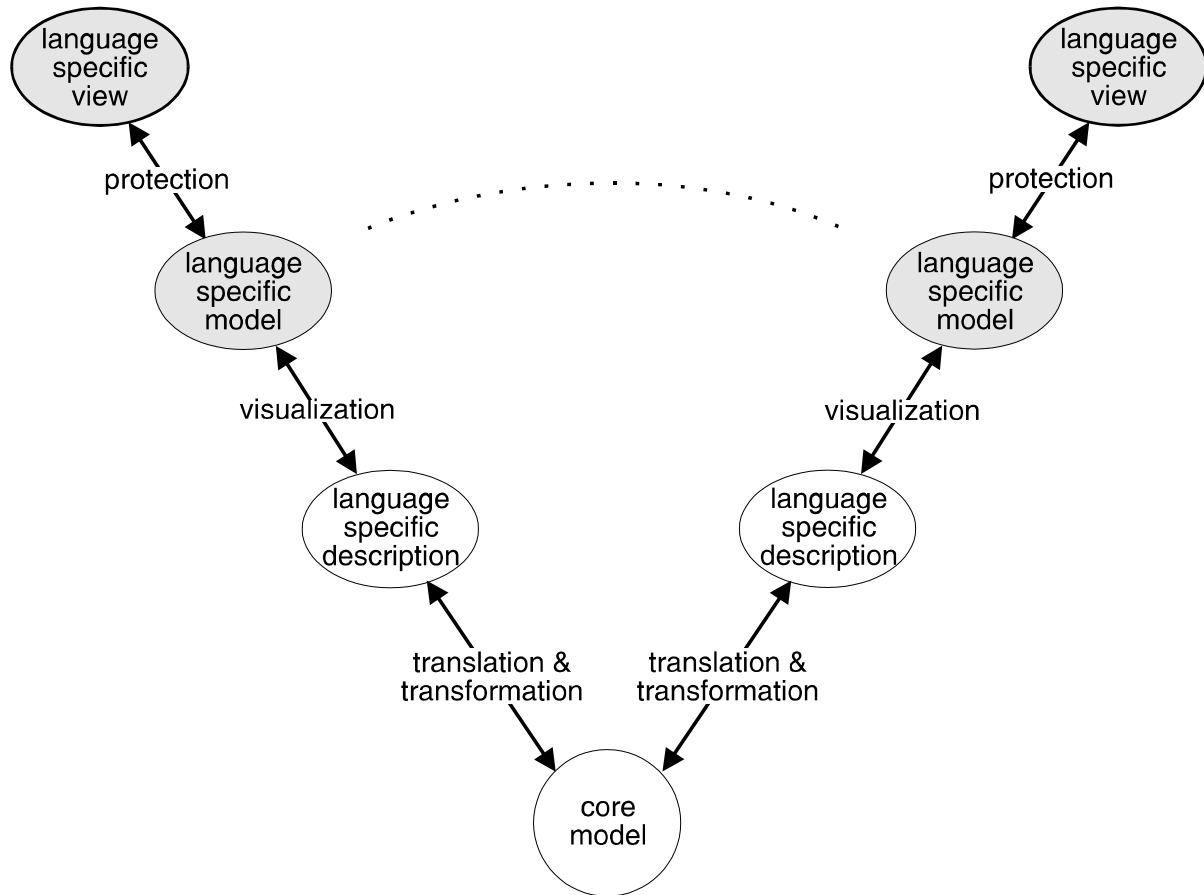


FIGURE 3.12 *Set-up for a system featuring multiple model formulations*

### 3.5 Realizability

A realization of the proposed system set-up is obtained when the following is incorporated:

- 1 a *mechanism for protection* of language specific models such that a language specific view is obtained, i.e. the Model-View-Controller paradigm (Krasner and Pope, 1988). This is a well-known and proven practice from the SMALLTALK-80 programming community. Details shall not be worked out here.
- 2 *methods for visualization* of the respective language specific descriptions to obtain language specific models, i.e. addition of notations and topographical positions. Proper notations for language elements in a certain formulation are simply a matter of definition. In the bond graph formulation, this definition has been well formalized on the basis of general rules (Breedveld, 1982) and generally is

followed rather closely. The same holds for the THESIS formulation (Wijbrans, 1993). In the iconic diagram formulation, the definition of notation is done individually for each subsystem, although some rules of appropriateness can be identified. Hence, the addition of notational information to language specific descriptions is rather trivial. The actual problem of visualization is to find a proper layout. Eades and Tamassia (1987) provide a fairly recent and complete bibliography on algorithms for automatic graph drawing. On the basis of this work, Wijsman (1992) has selected, modified and implemented an algorithm for automatic layout of iconic diagrams. Therewith, he has shown the feasibility of this part of the setup. Performance of this algorithm will be shown in the case study (see section 6.4). More details on the design and implementation of the algorithm are given by Wijsman (1992).

- 3 *methods for translation and transformation* from core model to the respective language specific descriptions and vice versa. Several publications have indicated the feasibility of converting functional models of dynamic systems into bond graphs (Ulrich and Seering, 1989; Finger and Rinderle, 1989; Redfield and Krishnan, 1992) and vice versa (Macfarlane and Donath, 1988; MacFarlane, 1989; Van Dijk et al., 1992). Hence, we have looked exclusively at the bond graph and iconic diagram model formulations. Algorithms for the conversions between these model formulations are presented and discussed in depth in chapter 4. Conversions to and from functional model formulations (i.e., THESIS) are not worked out here.
- 4 a *formalism* (language or data structure) for the core model, in which all information that is needed to obtain any of the language specific descriptions is defined unambiguously. This will be worked out in the next section.

### 3.6 Formalism for describing the core model

The formalism for describing the core model should comply to the following criteria:

- 1 it should be possible to integrate into the core model all information incorporated in any of the actual model formulations, and preferably nothing more. In other words, the *expressiveness* of the formalism should embrace the expressiveness of each of the model formulations as closely as possible
- 2 the formalism in which the core model is described will not be shown to the end user directly; it is purely a computer-oriented formalism. Therefore, apart from ambiguity, whether or not the formalism is powerful from the point of view of the designer is not of concern.
- 3 the system set-up has been designed such that maintenance of consistency and tractability are optimized. In line with this, redundancy in the information specified inside the core model should be avoided.
- 4 efficiency and performance of the system set-up are improved if conversion (i.e., translation, transformation and visualization) of the core model is readily possible. This implies that the main constructs from which the formalism for the core model is built up should be easily mappable to each of the model formulations. As all

formulations are instances of labeled and directed graphs, the core model formalism should also be based on such graphs.

An extensive treatment of all matters related to the selection of a proper formalism is beyond the scope of this thesis. Here, we just note that the SIDOX language family (Structured Interdisciplinary Description Of compleX systems, Wijbrans, 1993) is able to meet the above criteria. SIDOX originates from the SIDOPS language (Structured Interdisciplinary Description Of Physical Systems), which can be imagined as a special kind of textual description of bond graph models (Broenink, 1986). As a consequence, graphical images of bond graph (like) fragments can directly be translated to SIDOX constructs. Hence, we can indicate how a model fragment can be stored in the core model by presenting the appropriate ‘bond graph like’ fragment for it.

SIDOX is a language family, that is, it is a *set* of languages that have largely a common definition, but have been tailored to specific applications. The common part mainly entails the way in which models are structured, i.e., the use of ports, subsystems and connections. The specific parts of the languages deal with the definition of special purpose language elements and specialized grammars. Specific parts are designed such that correctness and consistency of models can be optimally checked. Currently, SIDOX languages are fully equipped to allow description of THESIS models and of bond graph models. Specific THESIS language elements are for example ‘merge point’ and ‘split point’. Specific bond graph elements are the ‘0-junction’ and ‘1-junction’.

Description of iconic diagram models using SIDOX requires that additional specific language elements and grammatical rules be incorporated in SIDOX. One obvious, trivial expansion that is required is that connections and ports be given a domain attribute, which can be checked upon consistency. More interesting problems are:

- *terminals* must be described in a bond graph like manner.
- *knots* must be described in a bond graph like manner.
- *global references* must be described in a bond graph like manner.

Extension of the SIDOX languages to solve these problems is straightforward if two new concepts are used: the  $\Delta$ -junction and the  $\Psi$ -junction, which are introduced in the next chapter. Using these concepts, iconic diagrams can be described in a bond graph like manner. This indicates that a proper formalism for describing the core model becomes available. The actual extension of SIDOX to enable description of iconic diagrams is currently being realized (Breunese, 1993).

### 3.7 Conclusions

An appropriate set of languages to be used during a certain design task includes languages which are suitable and necessary for the task, and which together are sufficient to complete the task successfully. For conceptual design of controlled electro-mechanical systems this implies that a set of graphical languages is required

that at least describes function, behavior and configuration. For functional descriptions, THESIS (a variant of the Modern Structured Analysis notation) is a good candidate. Bond graphs are a powerful language for behavioral descriptions. Finally, iconic diagrams (which denote physical phenomena) should be used for describing configuration. For conceptual design of controlled electro–mechanical systems, the set of THESIS, bond graphs and iconic diagrams is appropriate.

There are two central problems with formulating a model simultaneously in multiple languages:

- model formulations should be kept consistent.
- the information defining the model must be tractable.

Both problems are solved if a system supporting multiple model formulations is set up in the following way. The user has access to a language specific view of the model in an editor. This view is created by adding protection to a language specific model. Protection is needed in order to prevent the user from issuing commands that cannot be evaluated in the active model formulation. The (graphical) language specific model results from a visualization of a language specific description, i.e., a textual, computer–oriented codification of the model. A language specific description only stores intrinsic model properties, and not information like layout. Several language specific descriptions can be coordinated through a central core model, that integrally stores the information needed for all different model formulations. Language specific descriptions are linked with the core model by means of bi–directional transformations. These transformations make theories underlying different models compatible. Figure 3.12 depicts the system setup obtained in this way. It can be interpreted as a specific realization of the metamodel framework.

The SIDOX language family is suitable for describing the information incorporated in THESIS models and bond graph models in the core model. This formalism can straightforwardly be extended to allow description of iconic diagrams as well. Algorithms for transformations between language specific descriptions through this core model are discussed in depth in the next chapter, specifically concerning iconic diagrams and bond graphs. Automatic visualization of language specific descriptions to obtain a language specific model is feasible for iconic diagrams and bond graphs. Protection of a language specific model in a language specific view is also possible. Hence, the proposed system is realizable for the case of using bond graphs and iconic diagrams. No reasons can be foreseen why generalization of this result by including other formulations would be impossible.

## Iconic diagrams and bond graphs

### 4.1 Introduction

During the conceptual design of controlled electro–mechanical systems, both iconic diagrams and bond graphs can be applied fruitfully. A system set–up was proposed which supports the simultaneous formulation of models in multiple languages. This set–up specifies that multiple language specific models are coupled mutually by means of automated bi–directional conversions. Goal of this chapter is to *formalize* conversions between iconic diagrams and bond graphs, such that computer–based implementations can be realized.

There are three approaches towards conversion of graph–based models:

- *algorithmic* (sometimes referred to as computational); in this case, a deterministic procedure is available that completely specifies what steps have to be taken subsequently in order to obtain the output. For (sub)problems that clearly have good versus bad ways of solving, this approach is preferable, as it is computationally tractable and provides predictable results. Redfield and Krishnan (1992) use an algorithmic approach for converting a functional model into a bond graph model.
- *grammatical*; in this case, a set of formal ‘rewrite rules’ is given, that specify how a (fragment of a) description may be rewritten in another form, while maintaining correctness and consistency. This approach is preferable for problems that require *a* solution rather than *the* solution. Problems that do not have a well defined goal or optimum solution, and neither require the consideration of alternatives, lend themselves well for this. However, for complex problems, grammatical approaches generally do not perform well, as control at a strategic level is lacking. Finger and Rinderle (1989), for example, apply a grammatical approach for converting a functional model into a bond graph model.
- *search–based*; here, the solution is obtained by matching the input and/or the required output against a set of rules in a strategic way. This approach might work for problems where the previous two fail or are not suitable. However, it is computationally unattractive, and maintenance of consistency of the set of rules is hard for complex problems. The work of Ulrich and Seering (1989) on converting functional models into iconic diagrams is search–based.

The conversions required in the concept of multiple model formulations preferably should be realizable instantaneously; only then, multiple formulations of a model are truly simultaneously available. As no limit can be determined for the complexity of models that are considered during conceptual design, this implies that the conversions should be solved using an *algorithmic* approach.

The rest of this chapter is organized as follows. In section 4.2, terminals and knots in iconic diagrams are formally described and related to bond graph ports and junctions. Consideration is given to conversions between model formulations in general and between iconic diagrams and bond graph models in particular in section 4.3. After these preparatory discussions, the actual conversion algorithms are presented. Section 4.4 deals with the conversion from an iconic diagrams into a bond graph model, whilst section 4.5 deals with the reverse transformation. Section 4.6 summarizes conclusions.

## 4.2 Describing iconic diagrams

In section 3.3.2, iconic diagrams were introduced and the elements of this formulation were discussed shortly. We limited ourselves to an inventory, and did not go into a formalization of the language, that is, an unambiguous description of the language elements in formal terms. Before being able to specify a conversion algorithm, this has to be done. Because bond graphs are a formal language, this can be done by characterizing the elements of the iconic diagram language in terms of bond graphs.

The *interaction variables* of iconic diagrams, the across variable and the through variable, can be directly related to the bond graph interaction variables, effort and flow, as shown in table 4.1.

	Non-mechanical domains	Mechanical domains
across variable	effort	flow
through variable	flow	effort

TABLE 4.1 *Iconic diagram interaction variables in bond graph terms*

An iconic diagram *connection* consists of two constraints, one for the across variable and one for the through variable. For example, an electrical connection specifies that the voltage at one end equals the voltage at the other end, and that the current coming in at one end will go out at the other end. The same pair of constraints, only now in terms of effort and flow, is specified by a power bond. Because of this, we iconic diagram connections can be described by means of bonds.



The way in which electrical terminals of iconic diagrams can be described in terms of bond graphs has been indicated by Perelson (1975). Figure 4.1 depicts his proposition. Although this proposition outlines the approach, it falls short for our purposes for the following reasons:

- in this form, it cannot be generalized over domains. The terminals that are considered, paired terminals, specify across variables with respect to each other. In the electric domain, this indeed can be constructed by means of the bond graph junction structure fragment depicted in figure 4.1. However, in the mechanical domain, the dual fragment is needed, because there the across variable is a flow instead of an effort.
- only paired terminals are considered; the situation for autoreference terminals is not discussed. (Note: in the electrical domain, autoreference terminals are not used.)
- the dimension of bonds that are connected to the terminals should be equal to the dimension of the bonds contained in the junction structure fragment. This is not explicit in the proposition.
- Perelson rightfully notices that bond orientation is the key to specifying the direction of flow of the through variable and the polarity of the across variable, and examples are shown. However, general rules to obtain proper orientations are not formulated, and the difference between symmetric and asymmetric terminals (such as for the electric resistor and the diode respectively) is not worked out.

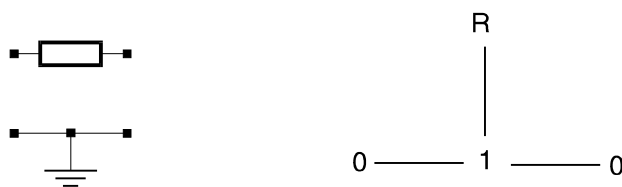


FIGURE 4.1 *Perelson's proposition for describing an electric resistor in bond graph terms (Perelson 1975, figure 9)*

Solution of these problems requires the introduction of two new structural elements that are *specifically for describing iconic diagrams*. These elements are the  $\mathcal{B}$ -junction and the  $\mathcal{M}$ -junction, and they can be defined in bond graph terms as shown in table 4.2. Note that the  $\mathcal{B}$ -junction and  $\mathcal{M}$ -junction are not equal to the  $\mathcal{S}$ -junction and  $\mathcal{P}$ -junction of Thoma (1975). First, the  $\mathcal{S}$ -junction and  $\mathcal{P}$ -junction are bond graph elements, whereas the  $\mathcal{B}$ -junction and  $\mathcal{M}$ -junction are used for describing iconic diagrams. Second, the  $\mathcal{S}$ -junction and the  $\mathcal{P}$ -junction do not have additional grammatical constraints such as the  $\mathcal{B}$ -junction and  $\mathcal{M}$ -junction have.

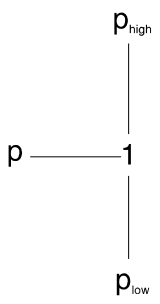
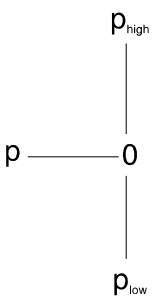
	Non-mechanical domains	Mechanical domains	Additional grammatical constraints
M-junction	0	1	dim(each connected bond) equal
8-junction			<p><math>\dim(p) = \dim(p_{high}) = \dim(p_{low})</math></p> <p>p always connected to a subsystem</p> <p><math>p_{high}</math> and <math>p_{low}</math> always connected to another junction</p>

TABLE 4.2 Definition of 8- and M-junction in bond graph terms

Using these junctions, the elements of the iconic diagram language can be formally describe. Figure 4.2 gives an illustrating example how this is done.

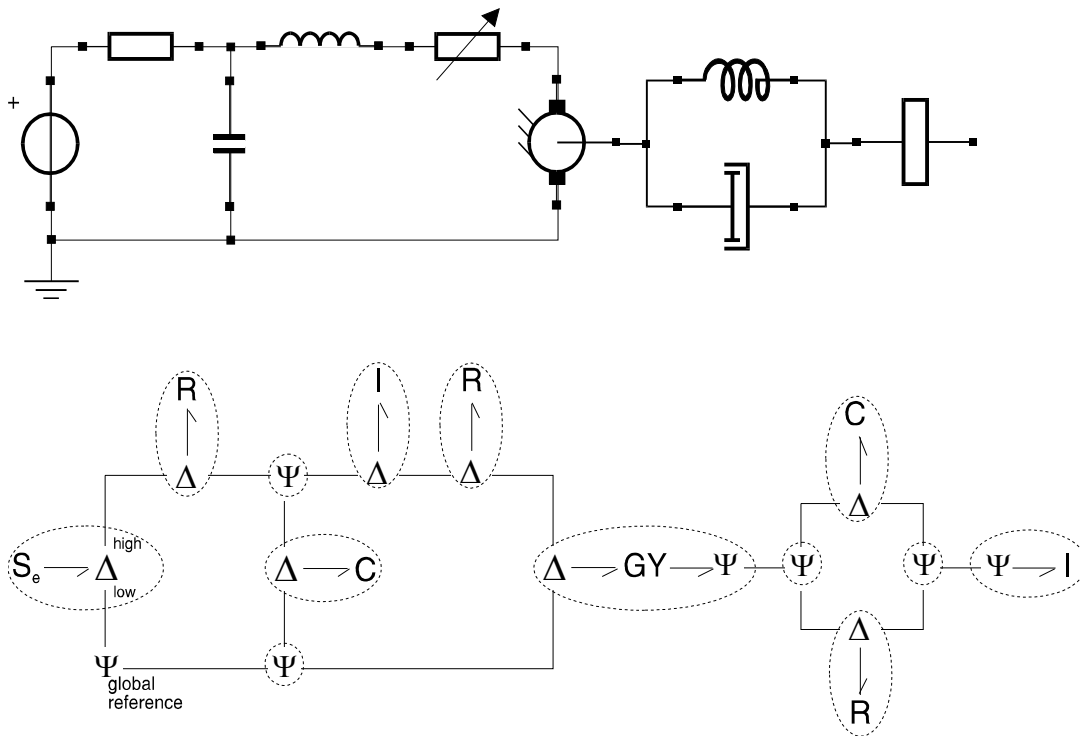


FIGURE 4.2 Example of describing an iconic diagram using 8- and M-junctions

The choice of the name 8–junction has been inspired by the fact that this junction models a *nodic* power port (Paynter, 1975; Hogan and Fasse, 1989), i.e. a power port that truly specifies a two–point (across) variable. The name M–junction has been chosen because this junction models an iconic diagram *knot*, i.e. a point where multiple connections are joined.

In figure 4.2 connections have not been given an orientation, because direction of flow of the through variables as well as polarity of across variables are not explicit in the iconic diagram. However, *constraints* on the orientation are explicit in the diagram by means of the character of the terminals. These constraints on orientation are maintained in the 8– and M–junctions. This is the main reason why the newly introduced junctions are useful; they properly formalize terminals in a local graph element. Generic formalizations of terminals and iconic diagram knots are shown in table 4.3.

	junction	orientation constraints
symmetric paired terminal	8–junction	orientation $p_{\text{high}}$ not equal orientation $p_{\text{low}}$
asymmetric paired terminal	8–junction	orientation $p_{\text{high}}$ not equal orientation $p$ orientation $p_{\text{low}}$ equal orientation $p$
autoreference terminal	M–junction	at least one in at least one out
knot	M–junction	indifferent
global reference	M–junction across var. = 0	preferred in

TABLE 4.3 *Language elements of iconic diagrams formally described*

### 4.3 General considerations on conversions

Two different formulations of one model are supposed to describe the same system, and hence should specify the same behavior. Therefore, conversions from one model formulation to another should be *behavior preserving*, that is, the describing equations of the total system should be the same. However, the partitioning and structuring of these equations are generally different. Referring to the example model of chapter 3, the structuring of the iconic diagram formulation (figure 3.4) and the bond graph formulation (figure 3.7) both contain knots and edges that are not present in the other one. However, the subsystems that are distinguished in either formulation are the

same. In general, this is not the case; the functional partitioning of a system for example generally differs from the physical partitioning (see e.g. Ulrich, 1988; Rinderle et al., 1989). Hence, conversions generally *do not preserve partitioning and structure*. So we conclude that conversion from one model formulation to another in general involves a repartitioning of the model and a transformation of the model structure.

In general, different model formulations imply:

- 1 different *subsystems*, i.e., equations or composite subsystems are grouped differently. This means that the port variables differ over formulations. As noted, iconic diagrams and bond graph models do not differ in this sense.
- 2 different *ports*, i.e., port variables are grouped differently. In the iconic diagram, port variables are grouped in terminals, whereas in bond graphs they are grouped in power (or signal) ports.
- 3 different *connections*, i.e., constraints on port variables are grouped differently. Iconic diagrams and bond graph models do not differ in this sense either; an iconic diagram connection specifies the same kind of constraints as a bond.
- 4 different *knots*, i.e., the connections of which ends are joined differ. For example, an electrical 1-junction in a bond graph joins bonds that cannot be joined by means of a knot in an iconic diagram. Conversely, global references are knots in an iconic diagram that generally are not present in bond graph models.

From this it follows that conversions between bond graph models and iconic diagrams are not the most difficult conversions; they do not affect model partitioning, and they do not require completely different connections. Hence, *translation of ports* can be considered independently from *transformation of structure* to deal with the differing knots.

There are situations for which it is desirable that the partitioning of a system in an iconic diagram differ from the partitioning in a bond graph model of the same system. This might be because one wants the iconic diagram to reflect the physical partitioning, whereas the bond graph model should reflect the functional partitioning. This means that the first issue mentioned above, different subsystems, also has to be addressed. The nice thing is, that the variation of partitioning can be considered independently from the conversion of structure. As long as elementary subsystems are not partitioned differently, varying the partitioning merely means that the model hierarchy is changed dynamically by contracting and unfolding composite subsystems, thereby creating and removing ports ‘on the fly’. Providing formalisms to support this is not covered in this thesis, and is considered as an interesting and important subject for future research.

## 4.4 Iconic diagram to bond graph algorithm

### 4.4.1 Available methods

Breedveld (1986) presents a systematic treatment of the conversion from Ideal Physical Models (i.e., iconic diagrams) to bond graphs. He shows that the underlying problems are the proper choice of one global reference for each domain that is present, and the proper choice of orientation for bonds in the bond graph. An explicit, human-oriented algorithm is given for the conversion. The approach taken in this algorithm is to create an almost proper junction structure right away by carefully inspecting the iconic diagram model. Van Dijk and Breedveld (1991) present an automated version of this algorithm. Regarding this algorithm, the following points can be made:

- Breunese (1992) demonstrated that the Breedveld algorithm does not explicitly identify the orientation constraint of *asymmetric paired terminals* (see table 4.3). As a consequence, a direct implementation of this algorithm can produce bond graph models with an abundant junction structure, due to the fact that orientations of bonds have been chosen wrongly initially. These abundant junctions cannot be removed using simple rules. When applying the algorithm manually, this situation seldom occurs, as humans immediately choose the bond orientation such that abundant junctions are not created.
- the Breedveld algorithm is based upon the assumption that the iconic diagram model is correct. Consequently, verification of this model by a proper *analysis* before actual translation and transformation has not been taken into account. In a design context, such a separate analysis phase is highly desirable in order to verify the design proposal.

Rinderle and Subramaniam (1991) also present a modeling system that is able to automatically obtain a bond graph model from an iconic diagram-like description. Their approach is basically to create a bond graph by composing bond graph fragments that have been predefined for each iconic diagram element. This leads to a graph with an abundant junction structure, that has some resemblance to our description with 8- and M-junction. By means of a fairly complete set of simplification rules, this abundant junction structure is subsequently simplified into a natural one. However, they do not systematically treat the main issue identified above, i.e., how to get rid of the references and orient bonds in a proper way. They seem to have solved this by manually providing bond graph fragments without references and with a proper orientation. Furthermore, they consider only mechanical iconic diagrams.

The above observations have lead us to conclude that conversion from iconic diagram should be preceded by an analysis phase, in which the iconic diagram is verified and in which proper orientation of bonds is prepared. The new algorithm presented hereafter follows this line of thought. It has been designed considering *single-dimensional*

*iconic diagrams* as input only, that is, iconic diagrams that only have power flow connections that represent a single effort and a single flow.

#### 4.4.2 Outline

Input to the procedure outlined here is a plain iconic diagram. Figure 4.2a depicts the system that is used to demonstrate the algorithm. The translation and transformation algorithm consists of the following steps:

- 1 Make a formal description of the iconic diagram by means of  $\mathcal{B}$ -junctions and  $\mathcal{M}$ -junctions. This converts the iconic diagram to what is shown in figure 4.2b.
- 2 *Verify* upon correctness and consistency of the formal description, i.e., check grammatical constraints of the junctions and make sure that all power ports and input signal ports are connected properly. If the model is not correct and consistent, stop conversion. In case of the example, the formal description is correct and consistent.
- 3 Perform an *orientation analysis*. During this analysis, the goal is to assign a power flow orientation to each power bond contained in the graph, such that orientation restrictions are met (as far as possible). This step will be worked out in more detail in the next subsection. If it is not completed successfully, the conversion algorithm is stopped. Figure 4.3 depicts the example system after this step in the conversion.

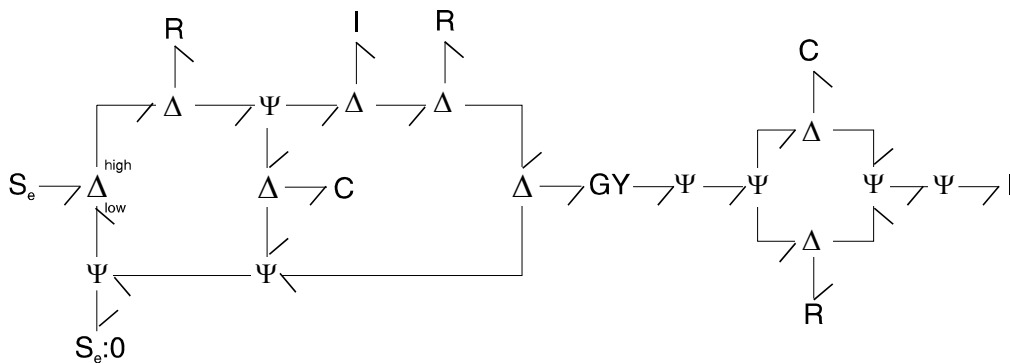


FIGURE 4.3 *Example system after orientation analysis*

- 4 *Remove* all *global references* and  $\mathcal{M}$ -junctions that are directly connected to these from the graph (not separately shown).
- 5 *Replace* all  $\mathcal{B}$ -junctions and  $\mathcal{M}$ -junctions by 1- and 0-junctions in the way specified by table 4.2. Figure 4.4 shows the bond graph model of the example that results from this.

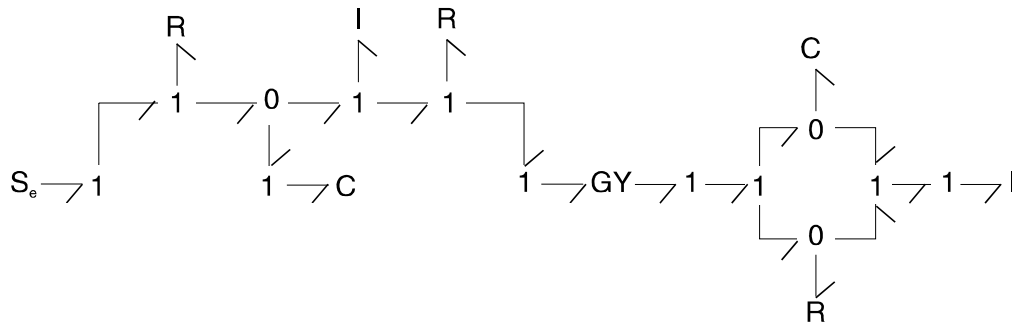


FIGURE 4.4 *Abundant bond graph that results after replacing iconic diagram junctions by bond graph junctions*

6 *Simplify* the resulting bond graph as far as possible. Appendix B lists common simplification rules. This final step leads to the bond graph depicted in figure 4.52.

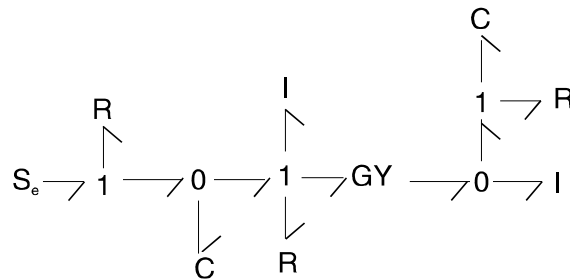


FIGURE 4.52 *Converted bond graph of the example system*

### 4.4.3 Orientation analysis

During orientation analysis, the goal is to assign a power flow orientation to each power bond contained in the formal description of the iconic diagram, such that orientation restrictions are met (as far as possible). Top (1993) was the first to describe an orientation assignment algorithm. The situation considered here is somewhat different, because orientation analysis has to be done in the iconic diagram rather than in the bond graph. Important differences in this respect are that the 8- and M-junctions have orientation restrictions that differ from those of bond graph junctions, and that the iconic diagram contains global references, whereas bond graphs do not. Therefore, Top’s algorithm is not applicable here.

Orientation analysis is completely comparable to causality analysis, which is done prior to converting bond graph models to functional models. Consequently, we have set up orientation analysis analogous to the standard procedure for causality analysis. To this end, a classification of orientation restrictions is needed and a procedure for the actual assignment of orientations must be given. We have identified the following types of orientation restrictions for ports of the formal description of iconic diagrams (e.g., figure 4.2b):

- 1 *fixed orientation*; this type of restriction is considered to hold for the ports of all subsystems except 8-junctions and M-junctions. The background of this is that there is an interplay between the sign of parameters in constitutive equations of an elementary subsystem and the orientation of the power flow through (one of) its ports (Perelson, 1975; Breedveld, 1986). In an automated modeling environment, it is virtually impossible to trace these interplays. A rigorous solution is to force fixed orientations upon ports of subsystems that can have constitutive equations with parameters. In practice, this limitation is not very restrictive, as one will seldomly want to use an orientation that conflicts with the fixed one.
- 2 *preferred orientation*; the M-junctions that represent global references have a 'preferred orientation in' for all connected bonds.
- 3 *orientation constraints*; this kind of restriction holds for the ports of a 8-junction and for a M-junction that represents an autoreference terminal, see table 4.3.
- 4 *indifferent orientation*; M-junctions that represent a knot have this type of orientation restriction.

Orientation may now be assigned using an algorithm derived from the causality assignment algorithm (Breedveld, 1986):

- 1 Assign all fixed orientations (type 1)
- 2 Propagate orientations by applying orientation constraints (type 3) as far as possible. If an orientation conflict (i.e., an unsatisfied constraint) occurs at this stage, the iconic diagram model is flawed and the orientation analysis is stopped unsuccessfully.
- 3 If the model is not completely oriented yet: assign a preferred orientation.
- 4 Propagate orientations by applying orientation constraints (type 3) as far as possible. If an orientation conflict (i.e., an unsatisfied constraint) occurs at this stage, try to solve it by incorporating a M-junction that represents a splitting point at the appropriate place.
- 5 Repeat step 3 and 4 until all preferred orientations have been assigned.
- 6 If the model is not completely oriented yet, two situations can be present: either the model does not contain an explicit global reference for each domain, or it contains a nodic subgraph (Paynter, 1975; Hogan and Fasse, 1989). In the first case, orientation analysis finishes unsuccessfully. In the latter case, continue by assigning an indifferent orientation.
- 7 Propagate orientations by applying orientation constraints (type 3) as far as possible. If an orientation conflict (i.e., an unsatisfied constraint) occurs at this stage, try to solve it by incorporating a M-junction that represents a splitting point at the appropriate place.
- 8 Repeat step 6 and 7 until all orientations have been assigned.
- 9 If the model is still not completely oriented yet, it does not contain an explicit global reference for each domain. Stop orientation analysis unsuccessfully. Otherwise, orientation analysis has been completed successfully.



Figure 4.6 depicts how orientation analysis proceeds for the example system. Bonds have been given a number in the order in which they have received orientation. Bond number 11 is the last one that has been oriented in step 1. After step 2, 16 bonds are oriented. When step 5 is completed, bond number 17 is also oriented. Two nodic substructures remain, as has been indicated in the figure. Through steps 7 to 9, these also become oriented.

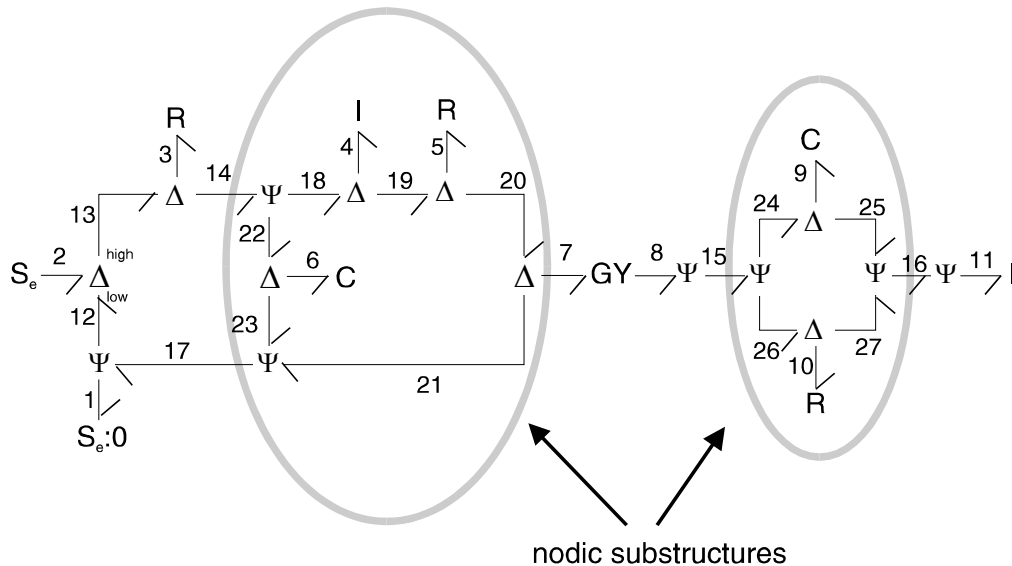


FIGURE 4.6 Process of orientation analysis

Besides nodic substructures, orientation analysis might also enable identification of other subtle structural properties of iconic diagrams. These properties do not surface when orientation is assigned according to Top's algorithm. However, further research is needed before more conclusions regarding the subject of orientation analysis can be drawn.

## 4.5 Bond graph to iconic diagram algorithm

### 4.5.1 Available methods

Several publications have indicated the utility and feasibility of converting bond graph (like) models to iconic diagrams, mostly in the context of automated conceptual design (Ulrich and Seering, 1989; Welch and Dixon, 1991x; Redfield and Krishnan, 1992; Welch, 1992). If details are given on the transformation, it appears that the conversions basically are realized as a *matching* process, i.e. they are search-based. Such approaches are known to be computationally unattractive, especially for complex systems. Furthermore, severe limitations are imposed upon the type of bond graphs that can be processed (e.g., all junction structures must be non-cyclic). As explained in section 4.1, we require a conversion *algorithm*, which is capable of transforming more

complex bond graph models. As far as we know, no such algorithm has been developed.

### 4.5.2 Inquiry

The basic approach we take in the transformation algorithm is to iteratively expand bond graph junctions into series- and parallel constructs of 8- and M-junctions. For example, we try to expand the electrical bond graph model of figure 4.7 into a [series construct of an  $S_e$ , an  $R$  and a [parallel construct of a  $C$  and a [series construct of an  $I$ , an  $R$  and a  $C$ ]]].

domain = electrical

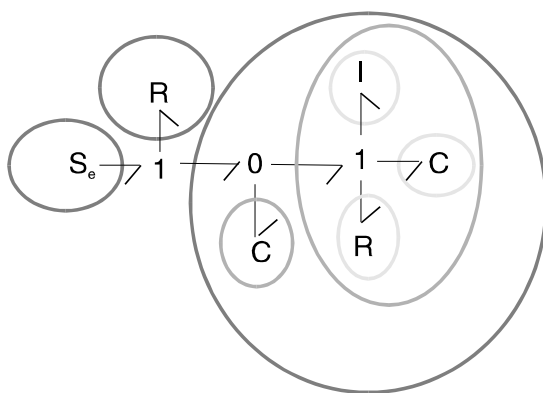


FIGURE 4.7 *Interpretation of a bond graph as a nesting of series and parallel constructs*

### Terminology

When multiport elements are present, the whole model cannot be converted in this way. Consider for example the model of figure 4.8. The electrical and the mechanical junction structure need to be processed separately. We use the term *unit* for the partial junction structures that are processed separately. So each largest non-disjoint junction structure fragment of a graph is said to form a unit. Note that all ports contained in one unit always have one and the same domain. Therefore, we can speak of an electrical unit, a mechanical unit, etc.

A second point is that in non-mechanical domains, the 0-junction is expanded in a parallel connection, whereas in mechanical domains this is the case for the 1-junction. This implies that we cannot speak of one junction type that will become a parallel construct in the iconic diagram. This can be overcome by switching over to the terminology of P-junctions and S-junctions, as introduced by Thoma (1975). The *P-junction* is equal to the 0-junction in non-mechanical domains, and equal to the 1-junction in mechanical domains. The *S-junction* is the dual of the P-junction. This has been indicated in figure 4.8.

Thirdly, it is handy to have a word to denote the model fragment that is treated as one piece in a series or parallel construct. We will use the term *branch* for this, see figure 4.8. Each branch has one unique *entrance connection*, i.e. the connection between the model fragment of the branch and the rest of the unit. Note that the entrance connection is considered *not* to be part of the branch.

The last piece of terminology that is introduced here is an *auto-reference port*: this is a port of a bond graph subsystem that will be represented in the iconic diagram by an autoreference terminal. In other words, it is a port of which the across variable is defined with respect to the global reference. Typical examples of autoreference ports are the rotation port of a motor, the port of a translational mass or of a rotational inertia and the port of a rotational friction element.

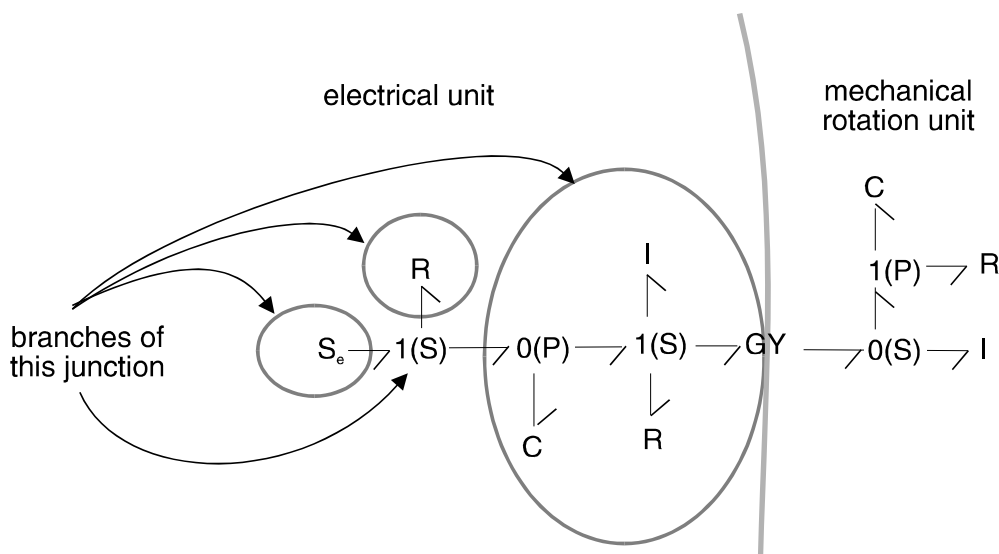


FIGURE 4.8 Units,  $P$ -junctions,  $S$ -junctions and branches

The above definitions only introduce terminology; they provide a shorthand for the scope of the algorithm presented hereafter. They are not used to actually describe a graph. This is unlike the previously introduced concepts of  $\mathcal{B}$ -junctions and  $\mathcal{M}$ -junctions. Furthermore, the definitions of auto-reference ports,  $P$ -junctions, and  $S$ -junctions are to be used in relation with bond graphs only.

### 4.5.3 General characterization of expansions

Using the above terminology, we can characterize the series-parallel expansion more concisely. The iterative expansion consists of three elements:

- expansion of a *simple branch*, i.e. a branch of which the entrance connection is connected to the single subsystem port of the branch.
- expansion of a *series branch*, i.e. a branch of which the entrance connection is connected inside the branch to an  $S$ -junction

- expansion of a *parallel branch*, i.e. a branch of which the entrance connection is connected inside the branch to an P-junction

These are worked out successively.

### **Expansion of a simple branch**

Simple branches are processed as specified in table 4.4.

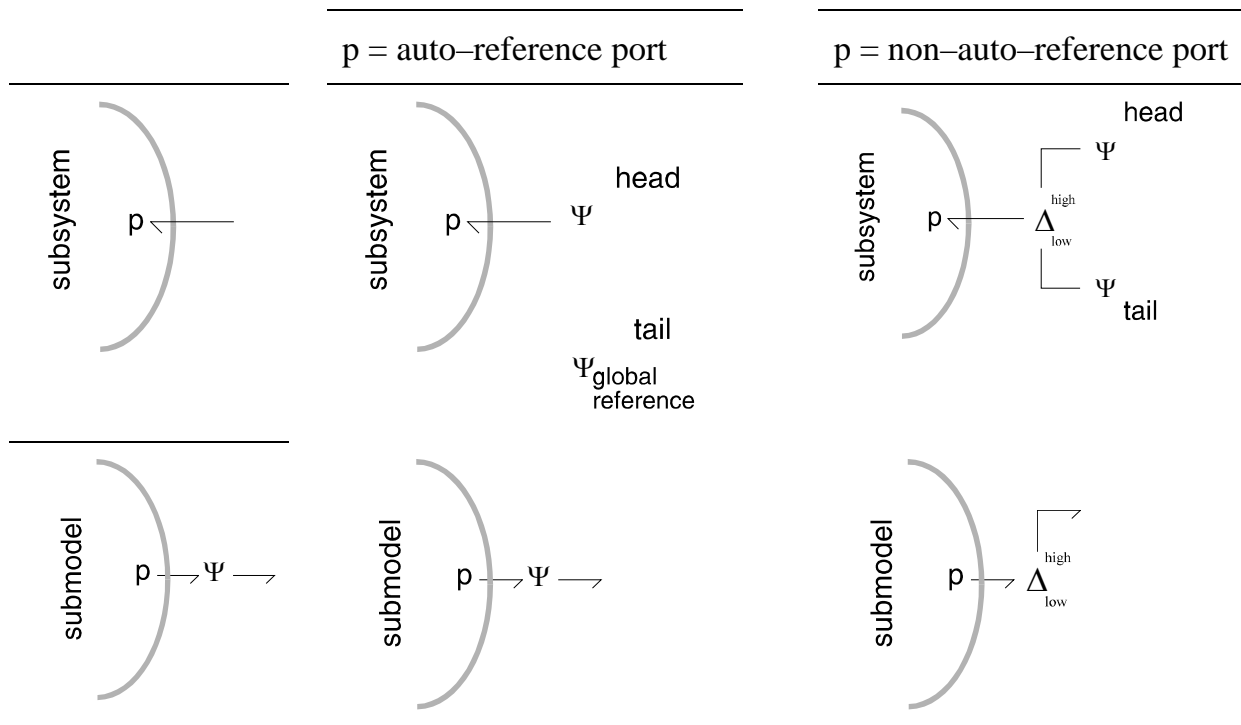


TABLE 4.4 *Expansion of simple branches*

Note that this expansion is a translation from bond graph ports to iconic diagram terminals. The heads and tails are called *dangling knots* as long as no other connections are made to them than the ones specified in table 4.4. Dangling knots are added to processed constructs so that it is clear whether a head or tail of a construct has been further processed in an embracing construct; if this is not the case, then the head or tail has just one connection.

### **Expansion of a series branch**

The standard expansion from S-junction to series construct is characterized in figure 4.9. There is an additional issue that needs attention during this expansion: the ordering problem.

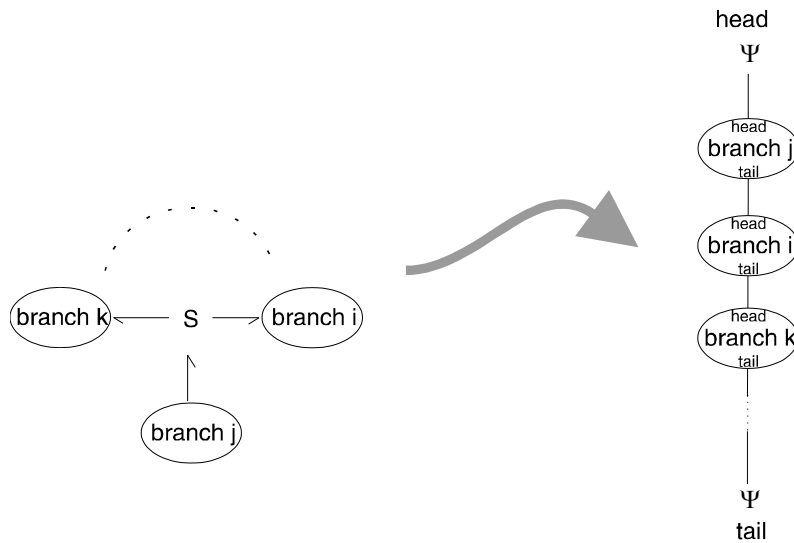


FIGURE 4.9 *Standard expansion of an S-junction into a series construct*

### Ordering

The way in which branches connected to an S-junction need to be ordered in the series connection is not explicitly specified in the bond graph. Hence, there is some freedom in choosing the order of branches in a series connection. For example, in figure 4.9, we have chosen the order ‘head — branch j — branch i — branch k — tail’. The order ‘head — branch j — branch k — branch i — tail’ could equally well have been chosen.

However, ordering of branches is not completely random; the following ordering rules are followed:

- 1 place branches with entrance connections oriented towards the S-junction closer to the head of the series construct than branches with entrance connections oriented away from the S-junction.
- 2 let branches of which the head or tail is a global reference appear at the appropriate outer end of the chain.
- 3 locate branches connected to the S-junction that represent sources or transducers as close to the appropriate outer end of the chain as possible.

Hence, the ordering problem is: how to order the branches of a series connection in a repeatable fashion conform the set of ordering rules. To solve this, the collection of bonds connected to a S-junction in the bond graph has to be ordered in some way, in accordance with the rules. There are two ways of doing this. The one that is straightforward and currently applied is to attribute a numeric order to the bonds connected to a junction. A major drawback of this is that there is no natural way in which the designer can control the order. A more intuitive way of ordering seems to be to derive the required order from the relative topographical position of the branches

connected to the S-junction; the first criterion is left to right, the second (in case horizontal position is about equal) top to bottom. The fact that applying this ordering principle to natural bond graph drawings leads to an ordering that complies well with the formulated rules gives confidence that this principle is indeed valid and preferable. However, for this solution to be repeatable, the automatic visualization (see section 3.5) of bond graph models should work accordingly. This makes implementation of this solution non-trivial.

### *Expansion of a parallel branch*

The expansion from P-junction to parallel constructs has been generally characterized in figure 4.10.

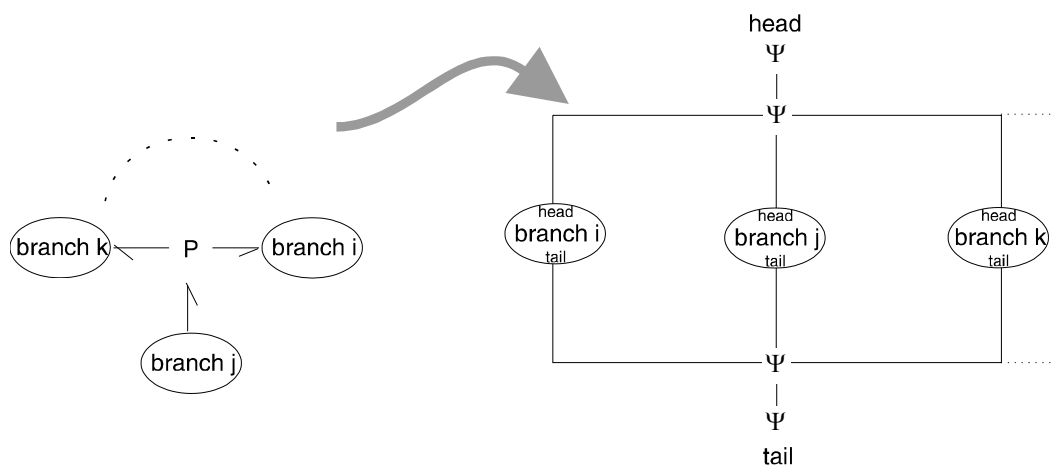


FIGURE 4.10 *Expansion of a P-junction into a parallel construct*

If the head or tail of any of the processed branches of the P-junction is a global reference, the head or tail of the newly created construct also becomes this global reference.

### **4.5.4 Remaining issues**

There are two remaining issues concerning the expansion of individual branches:

- 1 the cycle treatment problem, and
- 2 the reference placement problem.

#### *Cycle treatment*

Consider the electrical bond graph model of figure 4.11, which contains a cycle.

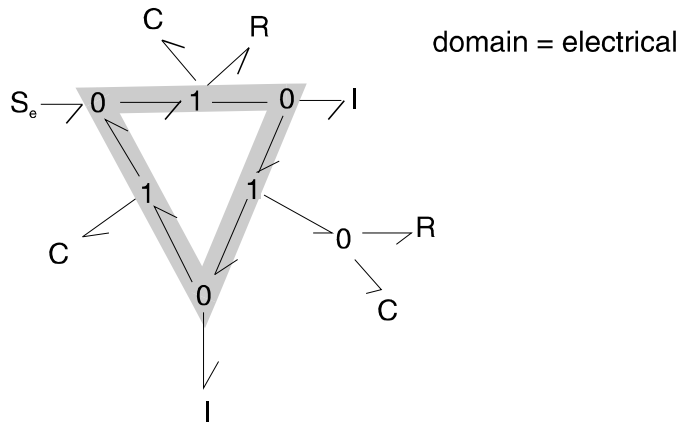


FIGURE 4.11 *Cyclic bond graph junction structure*

The iterative solution indicated above succeeds if expanded junctions deliver a head and a tail. For junctions contained in a tree-like bond graph fragment, this is the case. However, if there are cycles in the graph consisting of junctions and bonds only (like here), the approach fails. Because of the cycle, a head and a tail of a junction in the cycle cannot be found without special measures. Hence, junctions in cycles cannot be treated in the same way as junctions not contained in a cycle.

If this problem is examined more closely, three main observations can be made:

- a P-junction contained in a cycle (i.e. the 0-junctions in the cycle of figure 4.11) always represents a knot.
- an S-junction contained in a cycle always represents a series construct of all branches not contained in the cycle, i.e., all connected fragments adjoint to the junction which are not contained in a cycle.
- a bond contained in a cycle of a fully simplified bond graph always represents a connection between a knot (from the P-junction to which the bond is connected) and the head or tail of a series construct (from the S-junction to which the bond is connected).

Hence, we can process cycle junctions in the following way. We remove all bonds contained in a cycle, and convert the remaining (tree) bond graph fragments using the series/parallel approach. We make sure that a P-junction contained in a cycle will be properly transformed to a knot, and an S-junction to a series connection. We obtain the correct iconic diagram by restoring the removed bonds between the corresponding transformed model fragments. Figure 4.12 depicts how the example bond graph model with cycle is processed.

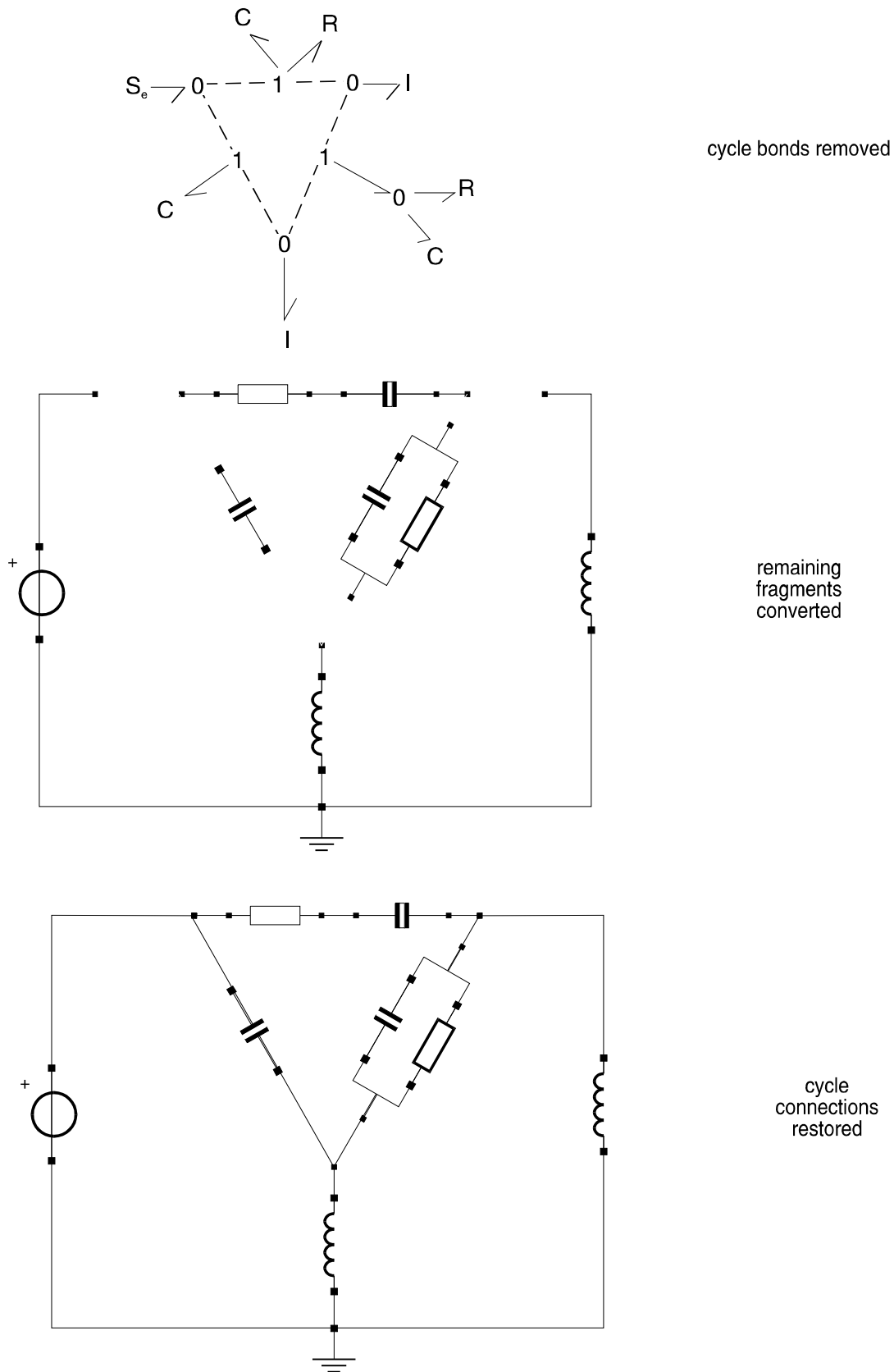


FIGURE 4.12 Processing of the bond graph with cycle



### Global reference placement

A properly defined iconic diagram includes global references for each unit. When examining these diagrams more closely, two things can be noted with respect to the places at which global references occur:

- there are specific places where a global reference cannot occur, namely in a series construct that is contained in a cycle, in a series construct that begins and ends with autoreference terminals and in a parallel construct between a knot and a global reference.
- global references are preferably located next to the lower terminals of sources and transducers.

The transformation algorithm should comply to these points. So the problem of global reference placement is determining the terminals to which a global reference should be connected such that a correct natural iconic diagram is obtained.

Consider a correct iconic diagram. It is always possible to interpret the place of the global reference in such a diagram as the closure of a series connection of a particular subsystem and other constructs. Figure 4.13 presents an illustrative example of this.

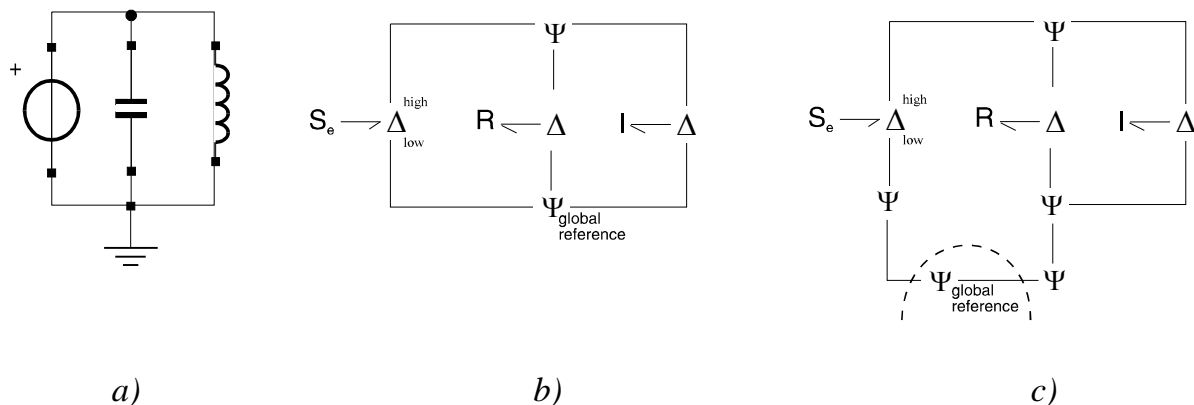


FIGURE 4.13 *Interpretation of global reference*

a) *original iconic diagram*

b) *straightforward formal description*

c) *as a series construct of a voltage source and a parallel construct*

This interpretation has two implications. One is that when the global reference is removed, places to which the global reference was connected remain recognizable as *dangling knots*. The second implication is that the location of the global reference is coupled to a *particular subsystem* with which the series connection starts or ends.

The way in which the conversion algorithm is started has not been specified yet. The above analysis points out that this should be utilized to solve the global reference placement problem:

- 1 start conversion of a bond graph unit in terms of a series connection. This guarantees that the dangling terminals that remain after expansion should be connected to a global reference.
- 2 choose the places from which conversion starts equal to subsystem ports or junctions to which a global reference definitely should not or preferably should be connected. This gives control over the places where global references will occur.

Therewith, all issues related to the transformation from bond graph to iconic diagram have been addressed.

#### 4.5.5 Outline

The conversion algorithm consists of the following steps:

- 1 *Verify* the bond graph formulation upon correctness and consistency, i.e. make sure that it is connected properly and is physically realizable (see Perelson 1975a).
- 2 *Sort ports* according to preference as starting point:
  - 1 ports of junctions contained in cycles involving bond graph junctions only
  - 2 autoreference ports
  - 3 ports of sources and transducers
  - 4 remaining ports
- 3 *Remove bonds* contained in *cycles* involving bond graph junctions only.
- 4 Take the first unprocessed port of the sorted list of ports. If this port is not contained in a junction, then *translate* the bond graph *port* to iconic diagram terminals according to table 4.4 and obtain the connected branch.
- 5 *Process* the *branch* as follows. Label the ports of the branch ‘processed’. If the branch is a
  - simple branch: translate the bond graph port into iconic diagram terminals according to table 4.4.
  - not a simple branch: obtain the connected subbranches. For each of these branches, do step 5 (i.e. *iterate*). Collect the heads and tails of the subbranches. If the branch is a:
    - + series branch: compose a series construct as specified by figure 4.9, thereby accounting for ordering.
    - + parallel branch: compose a parallel construct as specified by figure 4.10.
- 6 If there remain unprocessed ports: *goto* step 4.
- 7 *Restore* the *cycle bonds* that have been removed in step 3 between the now transformed model fragments.
- 7 *Add global references* to each unit and connect them to all dangling knots in the corresponding unit.
- 8 *Simplify* the resulting bond graph as far as possible. Appendix B lists simplification rules.

## 4.6 Conclusions

For describing iconic diagrams formally, two new concepts were introduced:

- the M-junction, suitable for describing a knot.
- the 8-junction, suitable for describing a difference between two knots.

These concepts are sufficient to capture all specific information incorporated in iconic diagrams, as was shown in table 4.3.

Conversions between model formulations should be behavior preserving. In general, conversions will not preserve structure and partitioning of the model. Conversions between iconic diagrams and bond graphs are a somewhat special case, as these conversions can be realized while preserving model partitioning.

When transforming an iconic diagram into a bond graph model, the main problem is how to properly remove all references. This involves the proper choice of one global reference for each domain and the proper choice of orientation for bonds in the bond graph. The solution to this problem is to incorporate an orientation analysis in the transformation. During orientation analysis, the goal is to assign a power flow orientation to each power connection contained in the iconic diagram. This analysis is comparable to causality analysis. It allows detection of subtle structural properties of iconic diagram models, like nodic substructures. Using orientation analysis, iconic diagrams can automatically be analyzed upon correctness and transformed into bond graph models.

Conversion from a bond graph model to an iconic diagram can be done iteratively by expanding bond graph junctions into parallel and series constructs. Three issues need attention in this:

- ordering; the order of subsystems that appear in a series connection of an iconic diagram is constrained. Information about this order is not explicitly available in a bond graph and needs to be generated appropriately. Rules for this are indicated.
- cycle treatment; bond graph junctions contained in a cycle consisting of junctions and bonds cannot be interpreted straightforwardly as representatives of series or parallel constructs. This can be solved by removing cycle bonds before processing and restoring them after processing.
- global reference placement; a bond graph generally does not include global references. Hence, these need to be generated during transformation and included correctly in the iconic diagram. By strategically choosing the way in which conversion of bond graph fragments is started, this can be controlled.



---

## Polymorphic modeling of engineering systems

### 5.1 Introduction

Building a model of a real world, continuous-time system involves the characterization of this system by a set of state variables  $x$  and by a set of (possibly time-varying) relations  $f(\cdot, t)$  on these state variables and environmental variables  $u$ . These relations are supposed to be satisfied at all time instances. By definition, the model that results is an abstraction of the real world system, in the sense that it intends to only incorporate properties of the system that are of interest and relevant, given the problem context. If this intention is met, i.e. if we have a *competent* model of the system, we can reason about and draw conclusions for the real world system based on the model. There are two main purposes (types of problem contexts) to do this:

- to understand an existing system (the *modeling* context)
- to define a system to be created (the *design* context)

For both purposes, it is important to be able to evaluate the competence of the model. In this thesis, the emphasis will be on the design context. However, the material presented in this chapter also has relevance in a modeling context.

Stated informally, the above implies that model building is the designation of a system in terms proper for the problem context. Therefore, the heart of model building is formed by three aspects, as can also be shown in a theoretic model of the modeling process (De Vries and Breedveld, 1992):

- the *decomposition* of the system into interrelated subsystems
- the *classification* of these subsystems and relations
- the *representation* of the resulting model.

Here, a subsystem is not necessarily a physical, concrete part, but it can also be of a conceptual nature. Also, it should be noted that decomposition and classification are dialectic concepts, that is, one cannot be considered without the other. In order to make a proper decomposition, the resulting parts need to be classified usefully, and conversely, in order to be able to classify, one generally has to distinguish a part that is considered.

Inside a computer-based modeling system, the decomposition, classification and representation of a model is determined by the data structures in which the model is maintained. Data structures are designed on the basis of structuring principles or *implementation techniques* that are available in computer science. Hence, the form that models have in a computer-based modeling tool is influenced by implementation techniques incorporated in the system. This statement is analyzed in detail in this chapter. The analysis is illustrated by showing what form a particular example system has in a system that incorporates the implementation techniques considered. The example system is largely the same as the one of chapter 3: a dc-motor, that is fed by a dc power supply, and that drives a gearbox that is connected to a flywheel (the load). The dc-motor model incorporates electrical inductance and resistance as well as mechanical inertia and friction. Different forms of this model are derived in Appendix D.

When a real world system is properly decomposed, when the subsystems have been meaningfully classified and when an insightful representation is made, the purpose for which the model was created is more easily fulfilled and evaluation of competence is less difficult. Consider as an example two different forms of models, namely the (mathematical) system characterization and bond graphs.

The *system characterization*  $f(\mathbf{x}, \mathbf{u}, t)$  can be regarded as the most straightforward and plain form of a model. A model with such an *internal structure* does not explicitly incorporate the decomposition into subsystems and their classification, and it is represented by means of mathematical equations. This is depicted in figure 5.1, both in general terms and for the example system. While this form of a model may be powerful for *solving* problems by means of calculation, it is not very suitable for the step ahead of that, namely for *obtaining* a good model of a complex system; it will not provide much insight and will not be easily understood. This holds for both a modeling and a design context.

$$f(\mathbf{x}, \mathbf{u}, t) \quad \dot{\mathbf{x}} = \begin{bmatrix} -0.003 & 2.78 \\ -0.027 & -33 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ S(t) \end{bmatrix}$$

a) b)

FIGURE 5.1 *Plain model*  
 a) *in general terms*  
 b) *worked out for the example system*

*Bond graphs* (Paynter, 1961; Breedveld et al., 1991) by contrast have proven to be well suited in this respect, at least in a modeling context. When using this language, the criterion on which to base the decomposition is clear and straightforward, namely the energetic behavior. Furthermore, the classification of the resulting elements is natural and rigorous, and most of all it is relevant from the perspective of engineering. Thus

when dealing with energy related aspects such as dynamic behavior, bond graphs support the making of good decompositions and classifications. At the same time the notation provided by Paynter and others (Karnopp and Rosenberg, 1968; Breedveld, 1982) features a powerful graphical representation. It depicts decomposition in a easily understandable way by means of a network, and properly reflects the classification of the elements at the same time. Finally, the bond graph representation allows flexibility in its attributes: it can range from qualitative to quantitative, from non-causal to causal, from conceptual to physical, and from non-oriented to fully oriented. Hence, bond graphs incorporate powerful concepts for decomposition, classification and representation. Therefore, they are helpful in building good models.

From the foregoing it follows that support systems for model building should incorporate implementation techniques that enable the use of powerful decomposition, classification and representation concepts. Available techniques that have been exploited in this sense are parametrization, typing and port-based interfacing; they are discussed in section 5.2.1 to 5.2.3 respectively. A detailed evaluation in section 5.3 shows that contemporary modeling systems do not support classification properly. Therefore, a new concept, polymorphic modeling, is introduced in section 5.4 to resolve these shortcomings. Section 5.5 presents examples of features that a polymorphic modeling system has. The following sections, 5.6 and 5.7, are devoted to discussion of system design and application issues, respectively. Conclusions are listed in section 5.8.

Except for the system design and application issues, the material of this chapter basically has been discussed by De Vries et al. (1993).

## 5.2 Existing techniques

Because bond graphs form a powerful model building environment, and also because the ideas presented here were largely based on experiences with bond graph modeling, this modeling environment is taken as a reference for required support of decomposition, classification and representation. However, this does not imply that application of the presented material is limited to a bond graph environment, as becomes clear in chapter 6. Implementation concepts are also evaluated with respect to reusability of models and to possibilities for organizing the model library, as these two aspects are important measures of the quality of support for model manipulation and model maintenance.

### 5.2.1 Parametrization

A first and rather obvious way of improving model specification is to *parametrize* the system characterization. In a parametrized model, (most) constant numerical values are specified in terms of symbolic entries (typically letters) that are separately given values. This is expressed in figure 5.2a by means of the parameter vector  $\theta$ . In a

system like TK SOLVER (Universal Technical Systems, 1988) models are specified in this form. It is interesting to see that, although parametrization seems straightforward, it took a long time before commercial CAD systems became available that incorporated this technique.

$$f(\mathbf{x}, \mathbf{u}, \boldsymbol{\theta}, t) \quad \dot{\mathbf{x}} = \begin{bmatrix} -\left(\frac{R_m}{n^2} + R_{load} + I_m\right) \frac{1}{I_{load}} & \frac{1}{n} \frac{1}{L_e} K_m \\ -\frac{1}{n} \frac{1}{I_{load}} K_m & -\frac{1}{L_e} R_m \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ S(t) \end{bmatrix}$$

a) b)

FIGURE 5.2 *Parametrized model*  
 a) *in general terms*  
 b) *worked out for the example system*

Introduction of this technique is important, as it demarcates the transfer from models that describe one particular instance of a real world system to models that describe a *class* of instances. Therewith, parametrization introduces a primitive form of classification. A second major improvement is that parameters represent the meaning of an attribute, rather than the value. Also, it results in models that show more structure, as expressions in terms of parameters are, contrary to direct numerical values, not worked out. At most, these combinations of parameters are replaced by a new parameter, which is then representative for the combination. Finally, parametrization of course leads to models that are more reusable.

In summary, parametrization is beneficial, because:

- it provides for a primitive form of classification.
- it improves the representation by showing the meaning instead of the value of attributes, and by showing more of the model's internal structure.
- it facilitates reuse of models.

### 5.2.2 Typing

Early systems used to support modeling, of which ACSL (Mitchell and Gauthier, 1976) is a well known example, were mainly based on a technique called “macro-modeling” (see figure 5.3). A macro is a *definition* of a subsystem, and consists of two parts: the heading and the body. The body describes the internal structure of the subsystem. The heading specifies the macro name and two kinds of formal arguments: the parameters  $\mathbf{e}$  (i.e., value-arguments in computer science terms), and the inputs and/or outputs of the subsystem (var-arguments, also called reference arguments). Input and output variables are denoted with  $\nu^{i/o}$  here. A subsystem defined in this way can be incorporated in a model (*instantiated*) by invoking the macro name with the actual



arguments only. The top level model might be viewed as a special kind of subsystem, namely one where the var-arguments are equal to environmental variables.

Macro-modeling can be regarded as a specific form of what is generally known as *typing*. Typing is the categorization of objects according to their usage and behavior. The key characteristic of typing in computer science is that the definition of an object (in this case a subsystem) is separated from its usage. This is done in order to *enforce correctness* of object usage and to *encapsulate local information* of an object (Cardelli and Wegner, 1985). To enable this, the type definition of the object consists of two parts: the interface part (the heading in case of macros) specifies the information that has to be provided at the moment of instantiation, and the implementation part (the body) contains the encapsulated local information of the object.

When a modeling tool allows to define a subsystem type, it means that this subsystem can be instantiated and treated as a single part. Consequently, the internal structure of a model can be expressed in terms of subsystems. In other words, it can be specified as an *aggregation* of parametrized, encapsulated (lower level) subsystems. As a result, the decomposition of the model becomes explicit. Not only the top level model, but also subsystems can be expressed in terms of lower level subsystems (i.e., the “boxes within boxes” idea). This implies the complete model takes the form of a *‘part-of’* hierarchy. Tools that feature this are therefore said to support hierarchical modeling. These effects of typing on the form of a model are depicted in figure 5.3a. In this figure and later on, the rectangles around subsystems symbolize typing.

With the advent of SIMULA (Dahl and Nygaard, 1966), the technique of typing was given an important extension: SIMULA allowed a type definition (named a class here) to be expressed as a specialization (a subclass) of an other type definition. This introduced the notion of *inheritance* into computer science: a type automatically inherits the properties of its supertype, which may then be extended or overridden. While this appeared very powerful in a programming context, it had only moderate influence on modeling tools. The reasons for this are examined in section 5.3. (Note: subtyping should not be confused with the possibility to define a type as a composition of other types. This construct does not introduce inheritance, because it lacks means for extension and overriding of attributes or behavior.)

Application of typing in computer based modeling tools is useful, because:

- it makes decomposition explicit by enabling the model to be organized in a part-of hierarchy.
- it extends classification by giving a type name to subsystems.
- it improves representation of models by encapsulating the internal structure of subsystems.
- it provides for reusability of complete subsystems.
- it enables the subsystem library to be organized around (related) types.

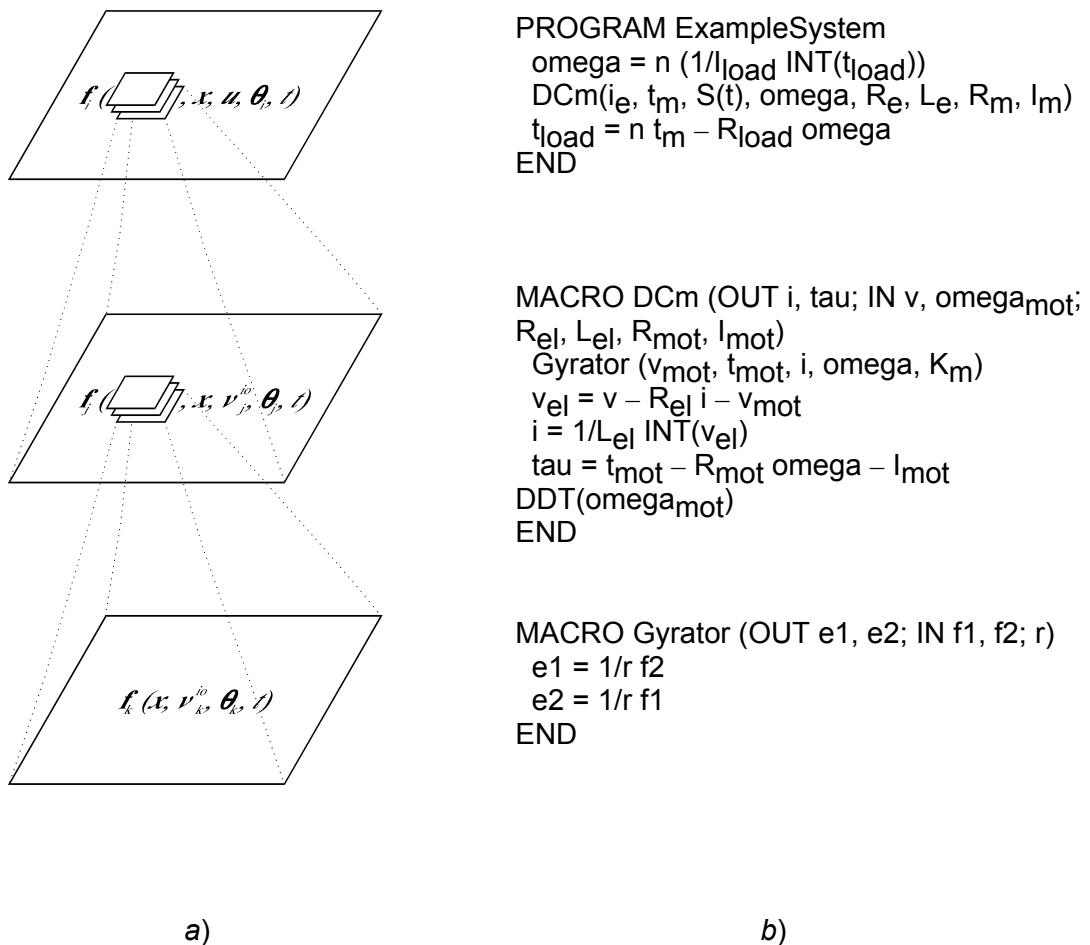


FIGURE 5.3 *Model in terms of encapsulated subsystems*  
 a) in general terms  
 b) worked out for the example system

### 5.2.3 Port based interfacing

The interface part of macros very much resembles those of subroutines in ‘conventional’ programming languages (e.g., PASCAL). When instantiating a subsystem in an aggregation, the inputs, outputs and parameters (actual arguments) are variables that are declared in the aggregation in which the subsystem is embedded, i.e. the ‘superpart’. In other words, the actual arguments are passed from the superpart down to the subsystem and processed locally.

In case of *port based interfaces*, the inputs and outputs of a subsystem (var-arguments) are declared as a special kind of variable, namely as *ports* (or also terminals). These variables are denoted as  $v^p$  here. When instantiating a subsystem in an aggregation, the inputs and outputs (actual arguments) are port variables that are declared

in

the

subsystem. In other words, port variables can be incorporated in relations of the subsystem's superpart. The conceptual difference with non-port based interfaces is that the variables  $v^p$  are 'popped up' from the subsystem to the superpart, instead of passed down from the superpart to the subsystem (as the variables  $v^{i0}$  in macro-modeling).

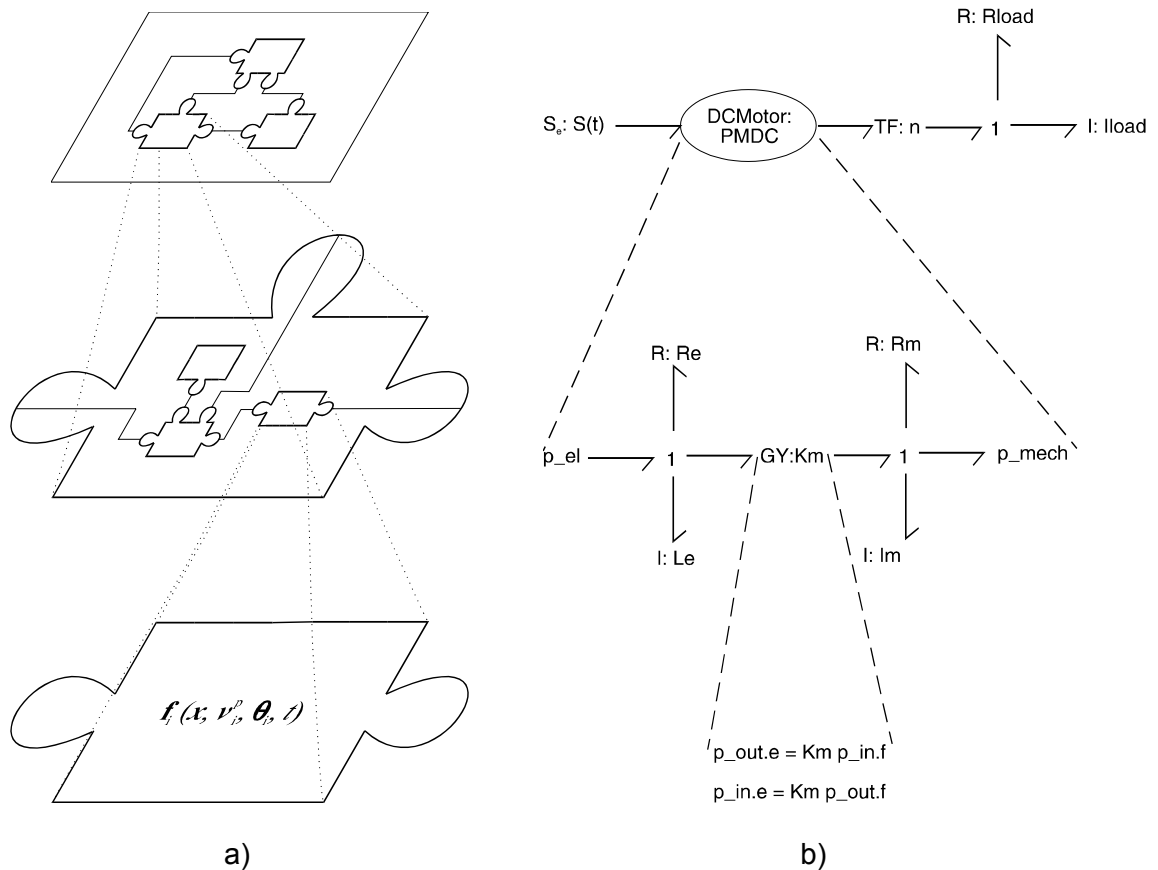


FIGURE 5.4 Model in terms of a network of subsystems  
 a) in general terms  
 b) worked out for the example system

Due to this change, an aggregation of subsystems is simply described in terms of connections, for a connection is a relation stating that the variables of the two involved ports are equal. It is useful then to restrict the internal structure of subsystems to be either a pure aggregation of lower level subsystems, called a *submodel*, or a pure set of equations, called an *elementary model* (Broenink, 1990). When this is done, it is

possible to represent a submodel as a *graph* or network of interconnected subsystems. In case of an elementary model, the internal structure will define the *constitutive equations*. This is depicted in figure 5.4a, where the ports are reflected by ‘bulbs’.

When using non-port based interfaces, subsystems are incorporated in models in the form of procedure calls which process the input and deliver output. This reflects a *process-oriented approach* towards model building: a model is built by stating the *processes* that take place. This approach is appropriate for dynamic, elementary models, for these indeed are characterized by processes. However, model building is generally concerned with the decomposition of a subsystem into smaller subsystems that are interconnected in some way, until some set of basic *components* is found. This typically reflects an *object-oriented approach*, and port based interfaces have enabled modeling tools which support this. The first modeling tool that incorporated port based interfaces was DYMOLA (Elmqvist, 1979). Most of the currently available bond graph oriented systems (for example ENPORT (Rosencode Associates, 1990) and CAMAS (Broenink, 1990)) have included a variant of this technique.

Note that Paynter’s *reticulation* (Paynter, 1961) provides a ‘smooth transition’ between the object-oriented and the process-oriented approaches. The basic bond graph elements specify idealized physical behavior (phenomena), and therefore ‘internally’ require a process-oriented approach. But by classifying these as conceptual elements, it becomes possible to treat the idealized physical phenomena as ‘objects’, which are incorporated in a conceptual network. Such a network in its turn mostly describes a (tangible) component, i.e. a submodel in a word bond graph. Word bond graphs typically are aggregations of components, and are made using an object-oriented approach. So the network of basic elements forms an intermediate layer between the model part that requires a process-oriented approach (inside the elements) and the part that requires an object-oriented approach (the aggregation of components). Typing and port based interfacing together provide computer-based model building tools the capability to fully support reticulation.

Port based interfaces improve computer-based modeling support, because:

- they improve decomposition by enabling an object-oriented model building approach.
- they allow representations of a model in the form of a network, which can be depicted graphically.
- they enable the subsystem library to be further organized according to port attributes.

### 5.3 Evaluation

Application of parametrization, typing and port based interfacing enables computer based modeling tools to fully support model decomposition and representation as done in bond graph modeling. Systems such as CAMAS and ENPORT exemplify this. CAMAS

also supports classification to some extent: each subsystem is declared as an instance of a certain class. However, in these (and comparable) systems, there is a one-to-one correspondence between a subsystem's type and its internal structure. This means that for each single subsystem type there is a unique form of instances. Stated more abstractly, types are *monomorphic* (of one form) in these systems. This characteristic is the cause for the fact that it is not possible to *define generic subsystem types* in available modeling systems. For example, it is impossible to define a general model type 'DCmotor' that can be incorporated in a network model and that covers detailed models of both Permanent Magnet DC-motors (PMDC) and wound DC-motors. Another example is that a type 'MechanicalFriction' that can represent both Coulomb friction and viscous friction cannot be defined. This is due to the difference in their internal structure. In (bond graph) modeling however, the invocation of generic models that are specified by different internal structures is a common procedure. Above the impossibility to define generic subsystems, it is also impossible to *represent a subsystem as a specific instance* of a more general class. Being able to do this is desirable, as generalization is utilized often when reasoning about a model. These two limitations show that available modeling systems do not support classification properly.

One might guess that the use of type inheritance as introduced in SIMULA would solve for this problem. Type inheritance means that the definition of some types, the subtypes, automatically includes ('inherits') definitions made for a designated other type, the supertype. In the subtype, additional definitions are made and/or inherited definitions are overridden. Hence, applying type inheritance allows the creation of generic types that capture common properties of multiple subsystem types, and it implies that a 'kind-of' relation can be described between types. This shows that type inheritance allows defining types in terms of other types, i.e., type inheritance is a form of *subtyping*. In SIMULA and other object oriented programming languages, the inheritance primarily involves operations defined in an object class.

In modeling, the major part of a type definition involves the description of the internal structure. Thus it seems attractive to enable inheritance of internal structures. However, internal structures of two subtypes with a common supertype are often not shared, so that inheritance is hardly advantageous. Coulomb friction and viscous friction, for example, have completely different constitutive equations, i.e. different internal structures. But even if internal structures are largely shared, type inheritance is not very powerful. Consider for example the internal structures of a PMDC motor and a shunt motor, see figure 5.5. The electrical parts of these internal structures are different. It is not so much a matter of extending or overriding the electrical part of an internal structure defined in a common supertype, but rather a matter of *reconfiguring* such a structure, using additional parts. Inheritance is not suitable for this (Aksit and Bergmans, 1992).

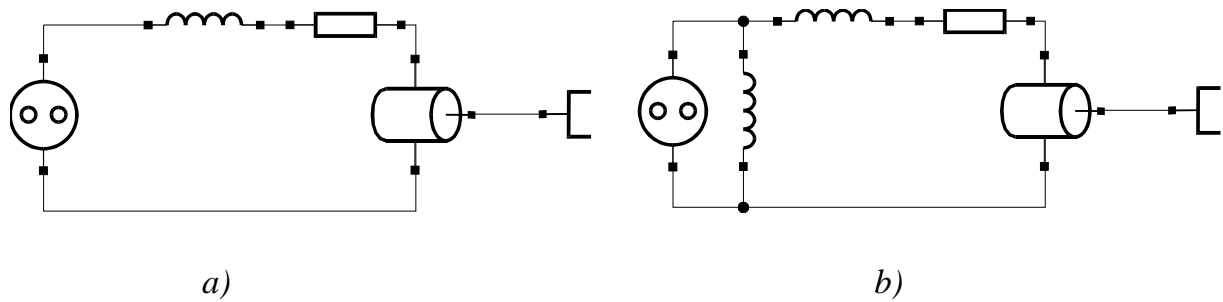


FIGURE 5.5 *Differing internal structures for a DC motor*  
 a) *PMDC motor*  
 b) *shunt motor*

Hence, generic subsystem types created for the purpose of type inheritance can in general not define a useful internal structure. Instead, they specify subsystem types which define some properties but have no internal structure. In computer science terms: type inheritance provides for *abstraction* of subsystem types. In other words, such abstract subsystem types allow to gradually define common properties of subsystems, and thus enable subsystem libraries to be organized in a kind-of hierarchy. This is a useful feature for development purposes (Rosenberg et al., 1992). From a modeling perspective however, we think it is less valuable. A model containing a generic (i.e. abstract) subsystem type is only partially specified; it can for example not be simulated, and things like main eigen frequencies cannot be determined. Also, this form of inheritance leads to a hierarchy in which an abstract type may have two different kinds of subtypes: subtypes that are instantiations of the abstract type, and subtypes that are refinements of the abstract type. Mixing these two different kinds of type relations is not desirable, because it obscures classification. Therefore, we conclude that type inheritance still does not give computer-based support the capability to capture classification properly. For it still is impossible to define, represent and manipulate fully specified subsystems which have a generic type. In the next section, a solution for this problem is proposed.

## 5.4 Polymorphic modeling

The basic reason why classification is not supported properly is that the inheritance mechanism enables abstraction, but leads to types that are not fully specified. In other words, the combination of abstraction and typing is too restrictive. This has been shown above. Once this is clear, the solution is easy: an abstraction barrier should be defined, which separates properties of (generic) subsystem definitions that are inherited by subtypes from properties that will not be inherited. This usage of an abstraction barrier is analogous to a technique known as *modularization* in computer science. Modularization is a program structuring principle that was most consequently applied in MODULA-2 (Wirth, 1982).

The abstraction barrier is defined here as the separation between essential properties and incidental properties of a subsystem. *Essential properties* are the properties of a subsystem that are ‘typical’, i.e., which are necessary to classify the subsystem. Essential properties are defined in the subsystem type (as before), and are inherited by subtypes. By contrast, *incidental properties* of a subsystem are not typical, and may take varying forms. Incidental properties are no longer defined in the subsystem type and therefore not inherited, but are defined in a *specification* of the subsystem type. This indicates that a complete model will exhibit two choices for each subsystem: its essential properties, reflected by the type, and its incidental properties, reflected by the specification. Furthermore, it follows that one type may have more than one specification. In figure 5.6, the consequences for the model of the introduction of an abstraction barrier are reflected by the dashed lines.

There are two main reasons why a type may have more than one specification. Firstly this occurs because one specification describes the behavior of a subsystem in more detail than the other, i.e. they have a *differing resolution*. In the case of the PMDC motor for example, both a specification containing only a gyrator can be defined and a specification that includes mechanical and electrical ‘parasitic effects’. The second reason is that the type has instances with *differing behavior*, although the difference in behavior is not essential. For example, Coulomb friction and viscous friction are described by different constitutive equations, but both specify mechanical friction. Which specification should be chosen for a subsystem in a model then depends on the context for which the model is used.

By combining subtyping with modularization, it becomes possible to define instantiations of an abstract type by means of a specification, and to define refinements of an abstract type by means of a subtype. Hence, this combination improves classification; it enables many different forms of instantiations of a typed subsystem. In other words, it supports polymorphic models of a subsystem. (Note that the polymorphism relates to description of the subsystem, not to its formulation.) This leads to the following definition.

*Polymorphic modeling* is the combined application of modularization and subtyping during model building, that is, the division of a subsystem description into a subsystem type and a subsystem specification, and the expression of a subsystem type in terms of one or more designated other types.

In this thesis, a particular form of subtyping is considered, namely single inheritance: a subsystem type is a specialization of one other subsystem type, and inherits its definitions. Single inheritance is the most simple and straightforward form of subtyping.

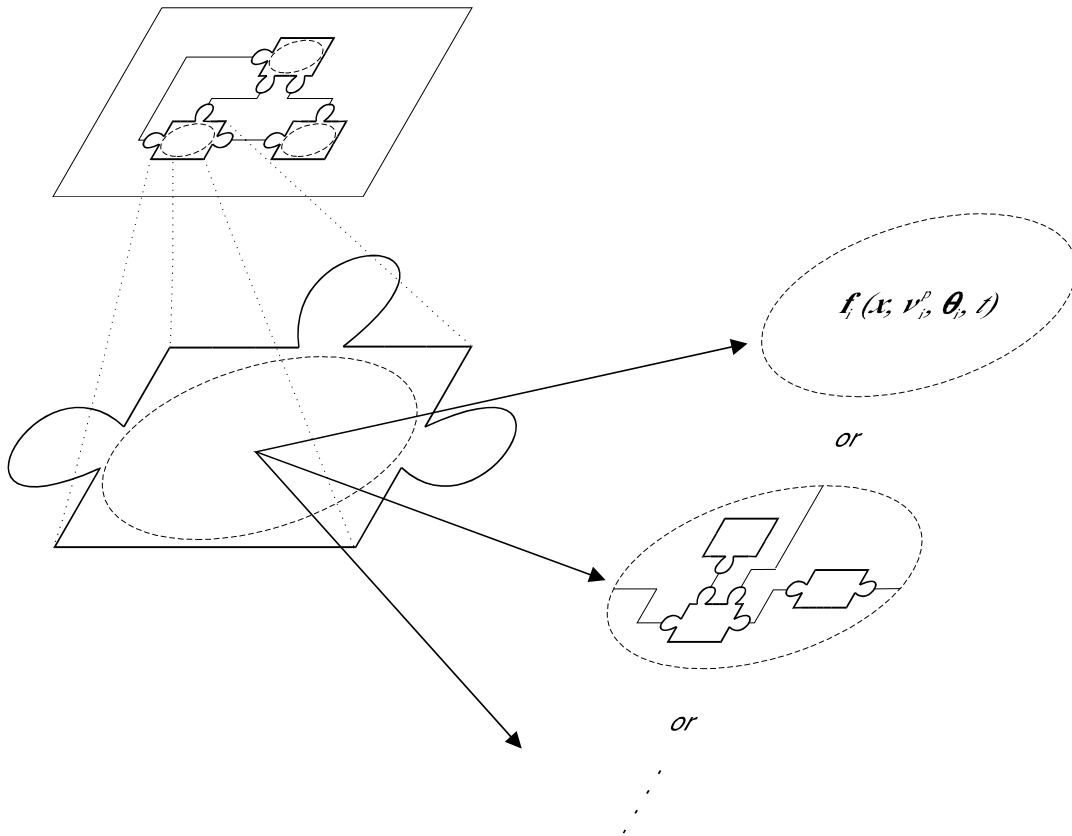


FIGURE 5.6 Polymorphic modeling: model in terms of modular, classified components

The concept of polymorphic modeling is useful, because:

- it improves classification of subsystems by means of generic as well as specific typing.
- it completes representation of models by separately depicting essential and incidental characteristics.
- it further enhances reuse, because subsystem types and subsystem specifications can be reused separately.
- it enables the subsystem library to be organized in a kind-of hierarchy, such that subsystems are specialized incrementally downwards. (That is, in case of single inheritance.)

In the next section, these features are illustrated.

## 5.5 Consequences

In this section, examples are given which show the consequences of applying subtyping and modularization during model building.



### 5.5.1 Support for evolutionary model building approach

Although modeling and design are recursive and interactive processes, the global working direction is downwards: one usually starts with decomposing the top level model into subsystems, and in the next step it is further developed by decomposing these subsystems again, and so forth. Thereby, the number of hierarchical layers of the model expands gradually. This maps exactly to the modularization introduced above: in the first step, the subsystems types of the top level model are determined, in the next step the proper specification for each of these subsystems is described in terms of lower level subsystem types, etc. A polymorphic modeling system separates types and specifications, and thus is well suited to support a top-down approach.

In this way, one uses the property that types can have specifications which differ in resolution. The strategy is to initially define properties at a non-detailed level, and only later be more specific. In case of monomorphic types this approach cannot be supported, because in order to determine the type of a subsystem, one has to make a choice for the internal structure. One could initially ignore the internal structure (i.e. use a type with an empty or completely wrong internal structure), and adjust it later. But this is more like a way around the lack of support! One has to make too much of an effort to obtain only the reusable properties of a related type. Also, in case of a new internal structure one has to create a new type with a new identity in order to not lose the empty or old type.

Using polymorphic modeling, the model builder can equally well start describing a specification of some subsystem, and after that select the type to which this specification belongs. In other words, a bottom-up approach is possible as well. Moreover, a *mix* of working in top-down and bottom-up fashions is possible. For example, the model builder can start top-down by developing the top level model and proceeding with specifying one of the subsystems (i.e. into several hierarchical levels). On basis of the insight obtained thusly, it can be decided to continue bottom-up by adjusting the top level model again, i.e. making a new type for the just defined specification, and so forth. In practice, any model building process will require such a mixed approach due to its evolutionary nature. A real process will never be purely top-down or bottom-up, and computer-based tools should be adjusted to that. Polymorphic modeling supports such an evolutionary approach.

### 5.5.2 Hierarchical subsystem library

Modularization and subtyping in the form of inheritance has two important consequences for libraries of computer-based modeling tools: the types are organized in a hierarchy, and for each type available specifications are listed. In this way, polymorphic modeling imposes a coherent structure on the collection of subsystems contained in the library. For illustration purposes, it is shown how a subsystem type hierarchy might be made for bond graph elements in figure 5.7. Other researchers have presented implementations of similar hierarchies (MacFarlane, 1989; Rosenberg et al., 1992). Note that other ways of organizing the hierarchy of bond graph elements can be

thought of (e.g. Breedveld, 1984). The reasons for choosing this particular organization are given in section 5.7.

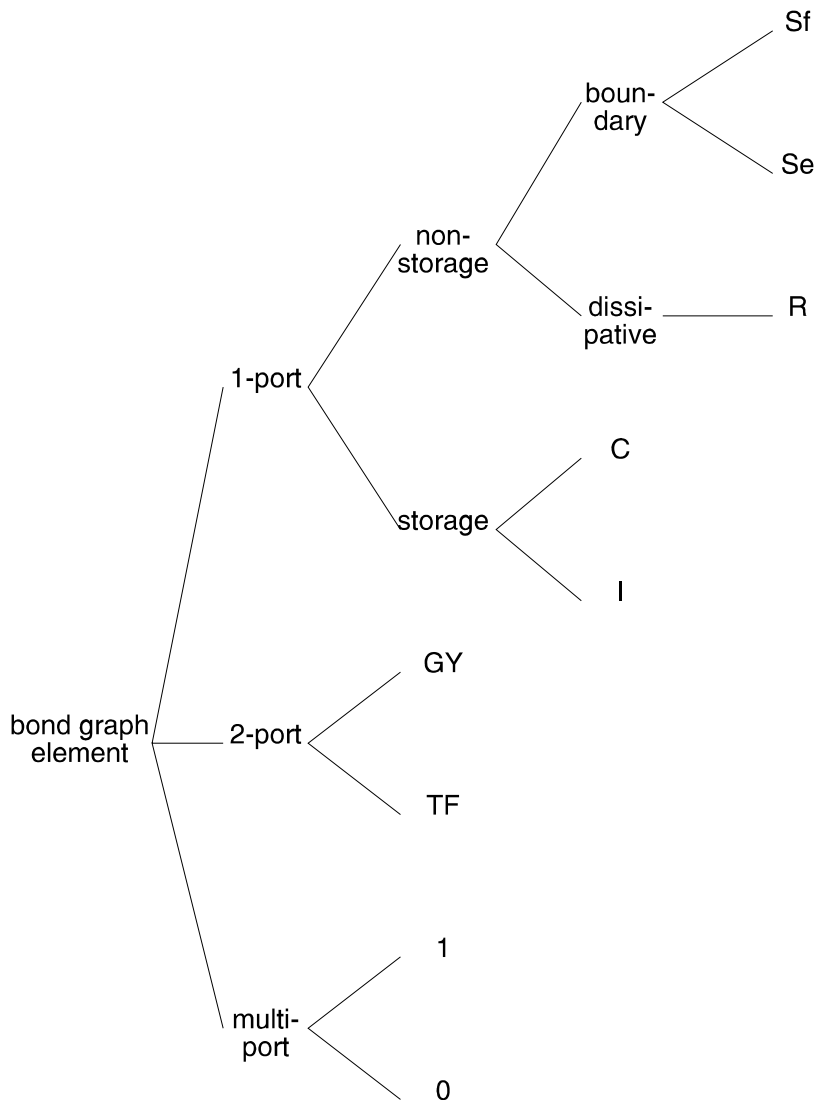


FIGURE 5.7 *Partial hierarchy of bond graph elements*

Analogous to the situation in programming (Cardelli and Wegner, 1985), we expect that the ease of reusing common properties is incidental to the clarity and conceptual parsimony provided by the coherent structure. In terms of model libraries, the main advantage of polymorphic modeling is not the additional reusability, but rather that the library is structured such that it is easy to understand and maintain by means of a hierarchy. Especially when the number of models in the library is large, such as for example in the OLMECO project (Olmeco Consortium, 1993), this is important. However, it does require that the hierarchy be built carefully. It should factor out

common properties of subsystem types stepwise, such that meaningful generic subsystems arise. We will further treat this issue in the section 5.7.

### 5.5.3 Creation of alternatives

Above, it was shown that polymorphic modeling leads to a hierarchy of subsystems, which can be depicted in a tree. An example of this is shown in figure 5.7. This implies that apart from the root, each subsystem has a more general supertype, and apart from the leaves, each subsystem has more specialized subtypes. Now an interesting perspective opens up: subsystems can be *generalized* and/or *specialized*. These manipulations can be done without actually changing the network of the model. Also, one can *vary between specifications* of a subsystem. Both options are possible only when polymorphic types are available, and they are interesting in order to create alternative models (design solutions).

Consider, for example, the system depicted in figure 5.8a. Suppose that the bond graph elements are incorporated in a hierarchy as proposed in figure 5.7. In this model, the second transformer is specified as a gearbox (figure 5.8b). However, if a specification for a harmonic drive is also available, we can create an alternative model without changing the network by choosing this specification (figure 5.8c). Alteration of the top level network is not needed for that. Also, the model can be partially dualized by subsequently generalizing and specializing the elements of the involved model fragment (figure 5.8d). The partial dualogue of the original model is a useful alternative, see figure 5.8e. For no elements shall the number of ports change, thus again no alterations of the network are needed. Furthermore, we see that the knowledge needed to create a dualogue is embedded in the structure of the hierarchy.

## 5.6 Design issues

Incorporating polymorphic modeling in computer-based systems gives rise to some system design issues that need to be discussed before implementation can be considered. This is the topic of this section.

### 5.6.1 Modularization

One way of looking at a type is to say that it is the interface layer between the inner world of a subsystem, i.e. its internal structure, and the outer world, i.e. the model in which it is incorporated (Simon, 1981). The role of a type, then, is to enforce correct interaction between these two worlds. If interaction is completely based on ports, the characteristics incorporated in a type definition always relate to the ports of a subsystem. Thus a convenient place for the abstraction barrier in a subsystem description is between the port definition and the remaining part of the subsystem definition. This is also suggested in figure 5.6.

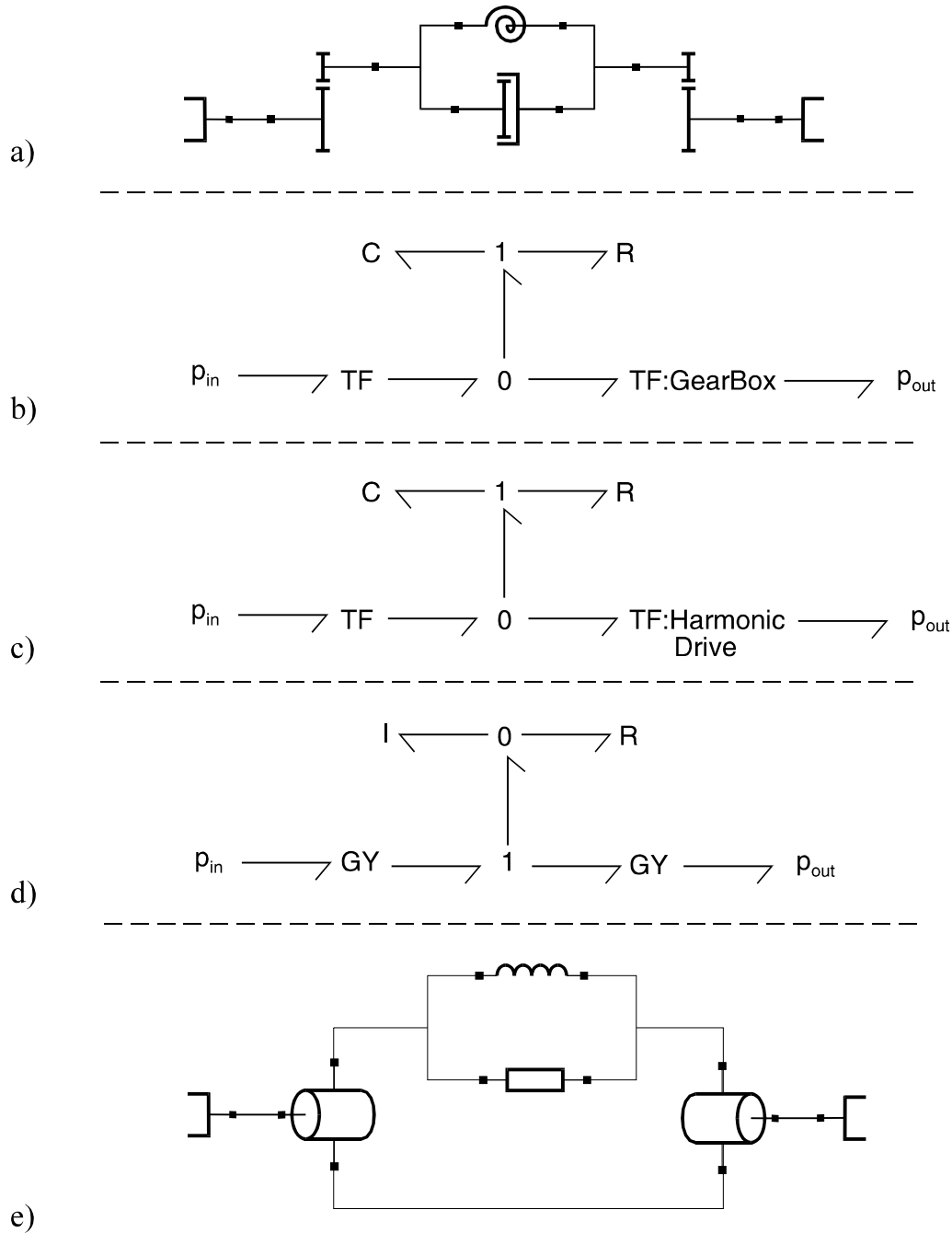


FIGURE 5.8 *Manipulating the model using subtyping and modularization*  
 a) *iconic diagram of original design proposal*  
 b) *corresponding bond graph*  
 c) *specification of second transformer changed*  
 d) *dualogue*  
 e) *iconic diagram of dualogue*

Consequently, the description of a specification is directly analogous to a non-modularized subsystem description, except that port definitions are replaced by a type declaration.

Type descriptions capture the essential characteristics of a subsystem. There are two kinds of essential characteristics: *connection characteristics*, which constrain the interaction from the point of view of the outer world, and *structural characteristics*, which constrain the interaction from the point of view of the inner world. Structural characteristics are a direct consequence of modularization; they are not explicitly present in non-polymorphic modeling systems. Therefore, type descriptions are not the same as the port definitions of non-modularized subsystem descriptions, and are elaborated shortly hereafter.

### ***Connection characteristics***

Connection characteristics should enforce correct usage of subsystems in models. They always relate to attributes of one or more complete ports, such as number, kind, dimension, (preferred) orientation of flow, etc. Therefore, it is easy to express these characteristics; it merely involves a proper definition of the ports of a subsystem. Also, the checking of these characteristics at the moment a subsystem is used is not difficult, as it comes down to verifying that the connections of a subsystem can be matched with its ports for all relevant attributes.

### ***Structural characteristics***

Structural characteristics should enforce specifications to have the correct form. Essentially, they involve functional constraints on port variables, like ‘the net power flow equals zero’, ‘effort  $e$  no function of flow  $f$ ’, or ‘port variables  $x$  and  $y$  are reciprocal’. These simple examples show that expressing these characteristics otherwise than in natural language requires a ‘meta-language’ that contains powerful mathematical constructs (like reciprocity). Even were such a language available, the formal verification of such characteristics at the moment a type and a specification were combined would be far from trivial. The only thing that can be verified without this meta-language is whether the variables incorporated in the ports of a type properly match those of the specification.

However, there are a few additional measures that can be taken, although they are not sufficient. It is relatively simple to define in a type description that its specifications have to contain a component of a certain type, or a parameter of a certain kind, or a specific function that is supposed to implement the required functional constraint(s). Verification of such characteristics is straightforward. In this way, the set of subsystems in the library that have not been verified for structural characteristics is significantly reduced. For these remaining subsystems, a mechanism can be devised that verifies structural characteristics right after a simulation run. At that moment, quantitative values of variables are available over a certain time interval, so that functional relations between variables can be explicitly checked, be it in a specific situation and for a limited interval.

No powerful means for describing and checking structural type characteristics have been developed; further research is needed.

### 5.6.2 Subtyping

The hierarchy shown in figure 5.7 is one featuring a specific form of subtyping, namely *single inheritance*: each type has only one supertype (except the root of course). Single inheritance is the simplest form of inheritance one can think of, yet it also restricts expressiveness. An example best clarifies this.

Suppose we want to refine the model of the dc-motor subsystem by replacing the gyrator with a two-port storage element. The electrical port of this element behaves like an electrical Inductance. The mechanical port that acts as a mechanical Capacitance. Hence, we have to define a new ‘IC’ type for the two-port storage element. This definition requires selection of an appropriate supertype. Two options spring to mind: the Capacitance or the Inductance. But which one to choose? Arbitrarily selecting one of the options is not satisfactory. The problem in fact is structural: we would like to be able to express that a subsystem’s type is related to more than one other type.

We could consider solving this by allowing *multiple inheritance* (i.e. enabling a type to have more than one supertype). However, this is not a good solution, mainly for two reasons.

Firstly, the use of multiple inheritance introduces non-trivial implementation problems. For example, the name of the port of the I-type is inherited from the type I-port, just as is the case for the name of the port of the C-type. Therefore, if we define the new type IC as a subtype of both I and C, the new type inherits two different ports that have the same name. In other words, due to multiple inheritance, one and the same attribute (like the name of the port in the example) is inherited more than once, along different paths. Therefore, priority rules or other measures are needed to guarantee consistency. However, the problem is even more complicated; it is not clear beforehand whether the attribute that is inherited more than once relates to one and the same feature, or to different features. The reason behind these problems is that the type inheritance as used here is a form of inheriting state specifications, which is known to cause inconsistencies or conflicts in case of applying multiple inheritance (Aksit and Bergmans, 1992).

Secondly, it is questionable whether multiple inheritance is *appropriate* here. We could rightfully say that the IC type is a kind of an Inductance if this behavior dominates, and if the second port merely modulates this behavior. However, equally well the reverse can be true; this depends on the problem context. Even more, it might be the case that the Inductance and Capacitance behavior are about equally important. What at least is true in all of these three cases is that the IC type transduces energy from one port to the other. In other words, the IC type is best characterized as a kind of Transducer, that consists of the combination of a Capacitance-like port and an Inductance-like port. Expressing the IC type as a kind of I and a kind of C is desirable from the reusability point of view, but from a modeling point of view the multiple inheritance is less defensible. This is also true in general: in a hierarchy concerning

one particular aspect (like dynamic behavior in our case) that is carefully built, multiple inheritance is mostly not appropriate.

The above remarks lead us to conclude that what single inheritance lacks is the ability to describe a new type as a *combination* of other types. To solve that, it must be possible to express, besides the ‘kind-of’ relation, that a part of the newly described type is ‘defined-as’ some designated existing type. For example, the IC type would be described as a ‘kind-of’ Transducer (i.e. a subtype thereof), with the electrical port ‘defined-as’ an Inductance and the mechanical port ‘defined-as’ a Capacitance. In that way, occurrences of problems analogous to those of multiple inheritance can be prevented, while still gaining expressiveness. However, in line with the characterization of designing as evolutionary, we think it is useful to first gain experience with polymorphic modeling in its simplest form, i.e. with only single inheritance.

## 5.7 Application advice

When making a new description of a subsystem, the following questions need to be answered by the model builder:

- what are the relevant properties of the subsystem?
- does the subsystem require a new type, or is it merely a new specification of an existing type? In other words, are the essential properties of the subsystem already captured in an existing type? A concrete example: is a PMDC motor a specification of the type DCMotor or is it a subtype of this type?

The first question is not specific for polymorphic modeling; it is the basic question underlying abstraction. For this reason, it is beyond the scope of this thesis. The second question, however, is a direct consequence of modularization: is a new subsystem definition an instantiation or a subset of an existing type? There is no general answer to this question, but some helpful rules are the following.

A subsystem is *not* a specification of the closest matching type, but requires a new type if:

- 1 it has a different number of ports
- 2 it differs in essential attributes of any of the ports
- 3 its phenomenological representation (‘iconic diagram’, see chapter 4) differs from the one of the considered type
- 4 multiple instantiations of the subsystem (that for example differ in resolution or in parameter values) should be contained in the library.

Of these rules, the first two indicate a difference in connection characteristics, whereas the latter two signal different structural characteristics.

Suppose now that a new type needs to be created. It already has been noted that the type hierarchy should be built up carefully. Common properties should be factored out stepwise, such that meaningful generic types arise. Furthermore, it has been shown that the structure of the hierarchy, if set up properly, can capture knowledge such as how to build dialogues. Finally, it should be user-oriented in the sense that it is clear immediately in which part of the tree a certain type can be found. To realize this, the following issues need to be addressed:

- 1 what is a proper set of subtypes for a certain type; or conversely, what is the proper supertype for a certain type? For example, in the hierarchy of figure 5.7, the subtypes of ‘bond graph element’ are ‘1–port’, ‘2–port’ and ‘multiport’. Instead, a primary distinction of ‘storage’ versus ‘non–storage’ could be considered (i.e. ‘energetic’ versus ‘non–energetic’, Breedveld, 1984).
- 2 if multiple aspects are available according to which subtypes can be distinguished: what order are we going to choose? The example hierarchy first subtypes according to the number of ports, then to ‘(non–)storage’, but it might also have been done otherwise.
- 3 what name should be given to a type? The bond graph hierarchy uses the term ‘2–port’, but this type might also have been called ‘transducer’. Also, the type that has been designated ‘multi–port’ could perhaps have been better called ‘junction’. The consequences of these choices will especially surface when one starts extending the hierarchy.

Useful rules to answer these questions are the following:

- 1 for each main engineering aspect (such as signal processing, dynamics, geometry), a separate hierarchy should be created. Consequently, a hierarchy will generally be organized around one type of port (respectively signal ports, energy ports or ports for geometrical connections, i.e. ‘solids ports’).
- 2 it is preferable to first subtype according to connection characteristics, and then to structural characteristics. This implies for example that the subtypes of the root of the hierarchy should involve a distinction based on the number of ports of the type. The reason for this is twofold: 1) connection characteristics are generally readily identified, structural characteristics are not; 2) connection characteristics are better reusable than structural characteristics.
- 3 a type should designate an artifact or a set of artifacts that is meaningful and relevant from the point of view of the main aspect of the hierarchy. Also, it should be meaningful and relevant from the perspective of the envisioned user of the hierarchy. For example, if the user of the system is someone with a bond graph background, the name ‘2–port’ is meaningful, but for someone not knowing bond graphs, ‘transducer’ is probably better.
- 4 a type should be a proper description for *all* the types that inherit its properties. Conversely, it should belong to *all* the sets that are defined by its supertypes, i.e. it should truly be ‘a kind of’ all its supertypes.



- 5 a supertype should preferably differ from its immediate subtypes in only one attribute. This attribute should have the default or no value for the supertype. Also, there should be one subtype for each of the values the attribute can have, that is, if its range is small enough (e.g., not exceeding the magical number 7).
- 6 the combined operation of generalizing a type one or a few levels and subsequently specializing the same amount of times along a different path should provide for transformation of a type which is meaningful and relevant from an engineering perspective.

## 5.8 Conclusions

Application of parametrization and typing in computer based modeling tools is useful, because these techniques enable hierarchical modeling and provide for encapsulated, reusable subsystems that are explicitly classified. In addition to this, port based interfaces improve modeling support by allowing an object-oriented instead of a process-oriented approach in modeling, and by making it possible to represent models as networks of interconnected subsystems. Modeling tools that incorporate parametrization, typing and port based interfaces can therewith fully support the reticulation as is done in bond graph modeling.

However, the classification as common in bond graph models is not supported adequately. This is due to the fact that subsystem types are monomorphic in these systems, i.e. generic subsystems can not be defined. The use of subtyping and inheritance, such as introduced in SIMULA and common in object oriented programming, does not solve this, because the internal structure of subsystems can generally not be abstracted into generic subsystem types. As has been shown, generic subsystems can only be described if subtyping is combined with modularization. Modularization means that a subsystem definition is divided into two parts: a type that defines essential properties, and a specification that defines incidental properties. By allowing one type to have more than one specification, subsystem types become polymorphic. Therefore, the combination of modularization and subtyping is called polymorphic modeling.

Polymorphic modeling enables computer-based systems to more adequately support classification. Also, the representation and reuse of subsystems is enhanced. Polymorphic types result in a hierarchical subsystem library. The importance of this is mostly that it provides the library with a conceptually clear and coherent structure. Furthermore, they give modeling systems the possibility to conform to the evolutionary nature of model building. Finally, they facilitate the manipulation of models, like the creation of alternatives and dialogues of bond graph models.

The system design of the modularization mechanism is based on the observation that a type will describe two kinds of essential properties: connection characteristics and structural characteristics. Both these characteristics relate to ports, and therefore a type contains the (extended) port definitions of a subsystem. Consequently, the description

of a specification is directly analogous to a non-modularised subsystem description, except that port definitions are replaced by a type declaration. Type description has become a separate activity. It appeared that the system design of subtyping should be based on single inheritance. Multiple inheritance is not applied for two reasons: it causes implementation problems, and it is often used for inappropriate reasons, namely for expressing an 'is defined as' rather than an 'is a kind of' relation.

When applying polymorphic modeling, the model builder is confronted with two new issues: 1) is a new subsystem definition an instantiation or a subset of an existing type? and 2) how should the hierarchy of subsystem types be built up?. Questions related to these issues have been identified globally, and useful rules on how to deal with them have been given.

## MAX, a mechatronic modeling environment

### 6.1 Introduction

The ultimate justification of engineering research lies in that it leads to artifacts that enhance human capabilities. In chapter 1, we identified that designing with an integrated problem solving approach is an activity that requires enhancement. In chapter 2, we presented the result of a theoretical investigation into design, and claimed to have found a model that can help to develop better support systems. In chapters 3, 4 and 5, we analyzed particular problems related to model building in the context of design, and proposed solutions that can help to overcome these. What remains to be done is to justify our claims, i.e. to show that:

- the model of design presented in chapter 2 can be used fruitfully when developing a design support system
- the concepts of multiple modeling languages (chapter 3 and 4) and polymorphic modeling (chapter 5) can be implemented and made to work in a system
- a system that is based on the model of design and that incorporates multiple modeling languages and polymorphic modeling enhances designing with an integrated design approach

For these purposes, the prototype computer-based design support system called MAX (Modeling and Analysis eXpert, Van Dijk and Breedveld, 1991; Van Dijk et al., 1992) was (further) developed. MAX is a *model building* environment; it supports the user in creating models and evaluating them by means of structural analyses. From that perspective, it is comparable to systems like QUBA (Top, 1993) and SCHEMEBUILDER (Sharpe and Bracewell, 1993). It does not provide functionality for model usage by means of calculation, like simulation or optimization. Rather, our philosophy in this is to create bidirectional links with existing computer-based environments that provide such functions, such as CAMAS (Broenink, 1990). Also, it does *not* implement an *automated model synthesis* strategy, i.e. it does not automatically compose models on the basis of design specifications, initial models or model assumptions. Therewith, it differs from most other automated modeling environments (Stein, 1991; Falkenhainer and Stein, 1992) as well as from systems aimed at automated conceptual design (e.g., Ulrich, 1988; Welch, 1992; Redfield and Krishnan, 1992; Malmqvist, 1993). The longer term objectives underlying MAX can be formulated as follows:

- create an extendible modeling kernel for a mechatronic designer's workbench.
- enable evaluation of modeling tools, concepts and methodologies in practical environments like industry (for commercial purposes) and university (for educational purposes). This implies that the system should have an appearance and performance that is beyond the prototyping phase that usually suffices for research projects.

The MAX system is presented in more detail in this chapter. We start with giving an overview of the development in section 6.2. Next, we discuss the state of the art of MAX (section 6.3). An example model building session is presented in section 6.4 as to give an impression of how the model builder generally interacts with the system. Subsequently, an evaluation is presented (section 6.5), and conclusions are summarized (section 6.6).

## 6.2 System development

### 6.2.1 Main issues

It was stated that one of the main objectives of MAX is to provide a modeling kernel of a mechatronic designer's workbench. Such a workbench or *integrated design environment* (Roozenburg, 1993) is of great importance to generate high quality output in minimum time. In an integrated design environment, all tools, techniques and methods used by the designer can be applied in an integrated way.

Support for different tasks is generally realized in separate (sub)systems in order to optimize its use, like Finite Element Analysis, parameter optimization or simulation. Therefore, a design environment is usually composed of a (possibly large) number of subsystems, each one with a specific supporting task. Integrated application requires that these subsystems are able to cooperate without user intervention. This implies that it should at least be possible to transfer information about the problem at hand and the proposed solution between subsystems. Hence, integration involves accessability of a common problem description and the possibility to coordinate support.

So when creating a designer's workbench, two main issues appear. First, it must be decided which support should be at the designer's disposal. This is treated in the next subsection (6.2.2). Second, it should be clarified how this support is to be integrated. This requires the description of a framework which defines the following:

- the subsystems to be incorporated in the environment and their functionality
- the interactions that are allowed between subsystems
- the data that is exchanged between subsystems

The framework should allow for new subsystems to be developed more or less independently and for existing subsystems to be incorporated, while still maintaining integration. In other words, the framework is the means to create an open, extendible

environment. In section 6.2.3, the framework within which MAX is developed is described.

Note that this section (6.2) discusses the functionality and organization of a mechatronic designer's workbench in general. Only parts of it are actually implemented in MAX; these are presented in section 6.3.

## 6.2.2 Identification of required support

In chapter 2, a model of designing was formulated. Here, it is shown that this model enables to identify in a systematic way what kind of support is required. For this purpose the basic model (figure 2.7) is used.

### *Support in the symbolic world*

The nature of the symbolic world is one of linguistics. Descriptions, while they exist, need some kind of easily accessible storage place. Support must be offered here by means of data- and knowledge bases. Their task is to store the descriptions properly and to supply the information when queried for it. In the model of designing, two separate elements exist in the symbolic world: the design object and described design knowledge. The design object forms the description of the current problem state, and the database holding this is referred to as the *repository*. By having one central repository, an integrated design environment provides for the availability of a common description of the problem at hand. The database storing described design knowledge is called the *library*.

Descriptions are only valid when they have an interpretation, i.e. when they are correctly 'spelled' and are meaningfully 'composed'. Inside computers, descriptions are expressed in two distinct forms: as a data-structure and as text in terms of a (computer) language. A computer language has a formal structure with rules for syntax and semantics. Descriptions expressed in a language easily allow a check for validity, as the syntax and the semantics of the language must apply to them. Descriptions expressed as a data-structure are much more effectively manipulated, however. In other words, both forms are useful, and therefore it needs to be possible to convert one form into the other, i.e. to (de-)compile models. Syntax- and semantics checks are usually performed right before compilation. Therefore, the support of syntax- and semantics checking and (de-) compilation, is logically combined in *parser-compilers*.

A design object or design knowledge generally contains several descriptions, that have to be consistent with each other, meaning that they do not describe incompatible facts. A change of the design object or design knowledge which would make it inconsistent should be detected before actually realizing it. In case of inconsistency, the system should react in an 'intelligent' way, i.e. in a coordinated manner. To this end, a *manager* is needed. The manager manages changes made to the design object and described design knowledge and coordinates different areas of support. It should therefore also support the checking of consistency.

Finally the validity of the descriptions depends on the goal for which they are used, which is the pragmatics aspect. Checking pragmatics is concerned with the completeness and applicability of a description to answer an information request (or rather observation request in terms of the model). It generally involves some kind of reasoning to do this checking, as information might be available in an implicit form. If so, the description has to be changed, either automatically or by the user. Because the manager takes care of the coordination of changes to descriptions, the support for checking pragmatics is logically incorporated into this subsystem. To this end, it should contain an AI-like *inference engine* that can do the reasoning needed for checking pragmatics.

Note that the manager is in fact devoted to supporting the interface between the symbolic world and the real world. It is concerned with the modification and observation of the design object and design knowledge, i.e. with the arrows between the symbolic world elements and the real world elements of the model of design (figure 2.7).

A summary of the subsystems required for supporting the symbolic world and their supporting task is given in table 6.1. All the tasks related to the symbolic world part of designing have been delegated to subsystems incorporated in the framework of the designer's workbench. Consequently, the symbolic world part of designing is completely contained in the support system; none remains the responsibility of the designer.

subsystem	function
repository	store the integrated design object
library	store described design knowledge
manager	check consistency, completeness and applicability coordinate different areas of support
parser-compiler	check syntax and semantics (de-)compile

TABLE 6.1 *Support in the symbolic world*

For the design object, extra structure was stated in the model by means of the four axes along which the descriptions may vary (c.f. section 2.4.3). This structure can be used to design the language definitions of the parser-compilers and the data- and file structures of the repository. Described design knowledge was classified into four categories. These categories can be used during the determination of a useful representation form and a structure for the library.

***Support in the real world***

The model of designing includes three processors in the real world: design automatons, descriptors and observers. The design automaton and the descriptor modify the design object and design knowledge, and the observer inspects both the design object and design knowledge. The environment has to provide the means for performing these manipulations. To modify descriptions contained in the design object and described design knowledge, ‘pencils’ and ‘erasers’, e.g. *writing-materials* are needed. To enable observation actions, some kind of display tools or *viewports* are needed, again for both the design object and described design knowledge.

The interaction between designer and environment is in general complex. The designer therefore needs guidance in how to interact with the system. This guidance can be separated into two areas: help on what stage (s)he is in now and how the system will react on possible interactions, and advice on how to proceed with the session. These functions can be combined in an *informer* which (on request) tells the designer what state the system is in at the moment, explains briefly how it got there, suggests what the designer can or should do next etc.

Letting the support system perform some of the design activities as well involves automation of design tasks. Subsystems which realize this automation are called *design automatons*. Design automatons automatically generate or modify descriptions of the design object. An example of such a subsystem is a simulator, which generates values of output variables for a particular input situation and time interval.

In table 6.2 the subsystems and their main function are summarized. All non-automated activities in the model of designing are driven by the designer, and therefore it remains his or her responsibility to realize them. Thus it can be concluded that the design process as incorporated in the model of design is distributed over the designer(s) and the environment.

subsystem	function
writing-materials	enable manual design actions
viewport	enable observation actions
design automaton	automate design actions
informer	provide information about the state of the system explain possibilities for continuation give advice on how to proceed

TABLE 6.2 *Support in the real world*

Structure is given to the design process in the model by means of elementary design actions. This set of actions can be used as a reference to define how writing–materials should be able to modify the design object, and to decide which of these actions are to be performed automatically (which tasks to automate in the automatons). The characteristics of observation actions can be applied in a similar manner.

### ***Support in the conceptual world***

Every human being has one private conceptual world, inside his or her mind. Consequently, this world can only be supported indirectly by the design environment, through the ‘sensors’ and ‘actuators’ of the person. In other words, mental processes cannot be influenced directly. Support cannot be concerned with concepts straightway, but deals with the creation of an environment in which the designer can concentrate on the conceptual world, i.e. mental processes. This means that support is given on the level of the interface to the conceptual world, i.e. the arrows from and to this world. Hence, the environment should allow the designer’s concepts to be applied straightforwardly in driving activities, and should stimulate concept forming by the designer through interpretation and inference.

Realization of the support mentioned above primarily concerns the *user interface* of the design environment. In literature, this aspect has been given a lot of attention. This is easily understood, for this world is the place where the “creativity” of designers has its roots. As this is a key factor in design, support of the conceptual world is very important. The task of the user interface can be summarized as to present the functionality of the environment in a form adapted to the conceptual world of the designer, such that (s)he is not bothered with matters that do not contribute to designing. Thus interaction between the design environment and the designer needs to be straightforward from the designer’s point of view, and not vice versa.

subsystem	function
user interface	adapt the system to the conceptual processes of the user

TABLE 6.3 *Support for the conceptual world*

In section 6.2.4, it is shown how the model of design can be applied in the development of the user interface of a support system.

### **6.2.3 Organization into system framework**

Above, the subsystems which should be incorporated in the environment have been identified. These were: repository, library, parser–compilers, manager, writing–materials, viewports, informer, design automatons and user interface. In this section, these subsystems are organized in a framework. From the viewpoint of system development, this organization should preferably be a layered structure. In that case it



is easier to keep the interactions between subsystems well-defined and manageable. This improves extendibility and maintainability characteristics of the system.

To get a framework with a layered structure, one needs to define some kind of hierarchy of subsystems. Part of the needed hierarchy is present in the three world model, see figure 2.2. The subsystems related to the symbolic world are ‘under’ the subsystems related to the real world, which in turn are ‘under’ the user interface. Additional hierarchy, inside the support offered within one world, remains to be defined. This follows in a natural way by considering which services a subsystem needs in order to perform its task and by evaluating which other subsystems offers these services. This will be shown hereafter.

Highest in the hierarchy is the user interface. The user interface presents the functionality of the system to the designer. To do so, it needs the service of the subsystems which actually realize this functionality. These are the subsystems which enable, guide or perform design actions and observation actions. Thus the user interface layer is on top of the writing-materials, the viewports, the informer and the design automatons. These subsystems take input from and present output to the user interface. Therefore, they are called the devices available to the user, and this layer is called the *devices layer*.

These devices all operate upon the common problem description. They enable modification or observation of the design object and described design knowledge. These tasks require coordinated access to the problem state. Service from the manager is needed to realize this. Therefore, the manager is under the devices layer in the framework.

The manager coordinates the interactions with the problem state and its manipulations and performs checks on consistency, completeness and applicability. These tasks are only concerned with the management of the access to the problem state, not with actual storage of information. For storage, the services of the repository and the library are required. So the layer containing these, called the *bases layer*, is under the manager.

The problem state and described knowledge stored in the bases layer are contained in data structures. Importing and exporting this information generally takes place at the language level. This requires the ability to (de-) compile the descriptions and to check correctness. For this purpose the bases layer requires the services of the parser-compilers. The layer formed by these parser-compilers is called the *language layer*. Hence, the language layer is under the bases layer. The language layer is the bottom layer of the hierarchy; it does not need the service of another subsystem. In the hierarchy obtained now, all the subsystems have indeed been incorporated. The complete framework, with the subsystems and their ordering, is presented in figure 6.1

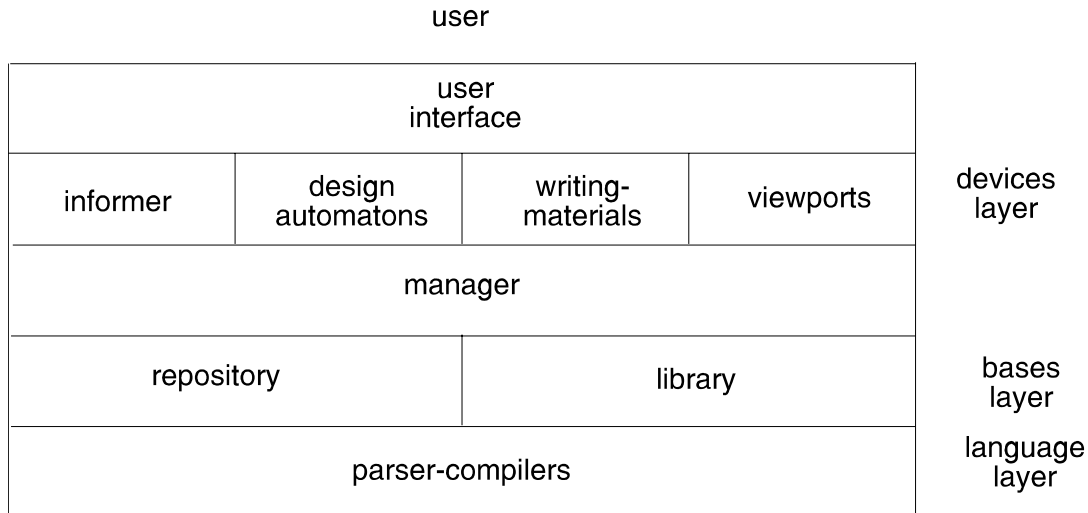


FIGURE 6.1 *Complete framework*

#### 6.2.4 Presentation to the user

The function of the user interface is to adapt the system to the conceptual processes of the user. The framework of figure 6.1 states that the functionality of the system is available to the user interface in the form of devices. So apart from appearance issues, the main question of user interface design is how devices should be made available as tools to the user.

The following three observations can be made :

- 1 the descriptor and the observer are generally simultaneously active on one and the same model. Hence, the user interface should present the designer a tool that integrates this model with the devices for description and observation. In other words, the user should interact with a tool that combines a writing-materials device, a viewport, and the involved model. This proposition maps exactly to what is known as the “Model-View-Controller” user interface paradigm (MVC-paradigm, Krasner and Pope, 1988), which originates from the SMALLTALK-80 programming community. This structuring principle has been shown to lead to conceptually clear, well maintainable, extendible interactive software.
- 2 the smallest unit of knowledge that directly drives activities is an operator (part of the procedural knowledge). Therefore, tools should provide functionality in terms of operators such as have been identified in the TEA model (Ullman et al., 1988). This especially implies that design automatons should be configured so that they represent a single operator, and that they can be activated in a tool within which the result can also be evaluated.
- 3 which tools should be available depends on the way in which the planning and control takes place during design; a switch of tools is preferably done only at the

moment that a new primitive goal is chosen. The way in which planning takes place depends on the mode of design, and thus the set of tools that a designer should have at his or her disposal is determined also by the mode of design:

- in the explorational mode of design, the typical procedure followed by a designer is: inspection of the problem, browsing for relevant known solutions and selecting one, adaptation of this solution to the particular situation. Hence, in the explorational mode, the tool set should consist of a ‘problem inspector’, a ‘principle solutions browser’ and a ‘proposed solution editor’.
- in the systematic mode of design, the typical procedure is something like: decide which path to follow in the design plan, perform next procedure in this path. Therefore, the tool set should incorporate mainly a ‘method controller’, and ‘processors’ for each procedure in the plan.
- in the problem solving mode of design, the typical procedure is: specify the goal, search and apply relevant transformations (rules) that might bring the goal closer, evaluate result. In this mode, the tool set should thus incorporate a ‘problem specifier’, a ‘planner/inference engine’, an ‘explainer’ and a ‘rule editor’.

If modes are to be applied in combination, also combined tool sets should be available. We have decided to concentrate initially on a system that supports the exploratory design mode.

### 6.2.5 Multiple model formulations and polymorphic modeling

In previous chapters, two concepts to support model building were developed: multiple model formulations (chapter 3 and 4) and polymorphic modeling (chapter 5). In order to utilize these concepts, they have to be fitted into the framework.

To implement *multiple model formulations*, the following rules should be applied:

- 1 all subsystems in the framework that process models in some way should separate the *description* of a (sub)system from its *representation*
- 2 the repository should store the core model
- 3 translations and transformations from and to the core model (e.g., the algorithms presented in chapter 5) should be implemented in a design automaton
- 4 visualization of language specific descriptions should be implemented in a design automaton
- 5 the manager should take care that translations and transformations and visualization are activated properly
- 6 writing-materials for a language specific model should be designed such that only model manipulations are available that result in visible model changes in the accompanying viewport (protection)

Implementation of *polymorphic modeling* requires the following rules to be complied to:

- 1 the library should separately store explicit definitions of subsystem types and subsystem specifications
- 2 the repository should contain subsystems that are instantiated from definitions contained in the library
- 3 writing materials should be designed that allow to create and modify definitions of subsystem types and subsystem specifications
- 4 writing materials should be devised that allow to vary types and specifications of subsystems contained in the core model

Summarizing, implementation of multiple model formulations and polymorphic modeling requires two structuring principles for model storage: separation of subsystem description and subsystem representation, and separation of subsystem type and subsystem specification. When these two principles are combined, one issue needs special attention: in how far should a differentiation be made between *components* and *elements*. This differentiation is important from the point of view of understanding models, but also has a direct practical relevance; a type that represents an element can only have a specification in terms of equations. (A type that represents a component should preferably be further reticulated until elements are obtained, but this is not necessary.) In the bond graph formulation, the differentiation between component and element is generally explicitly represented: components are depicted by ellipses (i.e., as a word bond graph subsystem), whilst elements are represented by mnemonic codes. In iconic diagrams, the differentiation is usually not made explicit. There are three ways to test whether a polymorphic subsystem is a component or an element:

- 1 *type-based*: if a type inherits from a bond graph element, then the subsystem is an element, otherwise it is a component. Hence, the model builder decides on whether the subsystem is a component or element when the type is chosen, which will usually be at the moment of subsystem creation.
- 2 *specification-based*: if the specification is a graph, then the subsystem is a component, else it is an element. The model builder thus decides on whether the subsystem is a component or element when the internal dynamics of the subsystem dynamics are determined.
- 3 *type and specification-based*: if the type inherits from a bond graph element and if the specification is a set of equations, then the type is an element, else it is a component. The decision on component or element behavior is, like in the previous case, taken when the internal dynamics of the subsystem dynamics are determined.

The first solution has two drawbacks; this option implies that the number of types drastically increases, and it might not be clear at instantiation time whether a subsystem is a component or an element. The second proposal is undesirable because the differentiation can give wrong results; a type can have an equational specification while not being an element. In the last option, the chances that such wrong results occur are diminished, although not completely eliminated; even if a type inherits from an element and has an equational specification, it might still not be an element.

However, this will occur infrequently. Therefore, we assert that the last solution is preferable; the drawback of this solution is less serious than those of the first solution.

## 6.3 State of the art

For illustration and in order to prepare the case study of section 6.4, the state of the art of the MAX system is described in this section. However, it should be kept in mind that this is a momentary impression; MAX is continually being modified and expanded. As such, it proves that design indeed is evolutionary. We start by giving an overview of the complete system, and then look at the subsystem type hierarchy in more detail.

### 6.3.1 Overview

The system has been implemented in SMALLTALK-80. This is an object oriented language (Goldberg and Robson, 1989) and programming environment (Goldberg, 1989), that is well suited for incremental development of prototypical software applications. The organization of the software of MAX complies to the framework of figure 6.1 and the rules stated in section 6.2.5.

The user interface of MAX was set up largely according to the OPEN LOOK guidelines (Sun, 1990), see Stet (1993). Consequently, MAX works with a windowing environment that meets the major goals of a good interactive application: it is simple, consistent and efficient.

MAX contains the following tools.

#### *Graphical image editors*

MAX features two graphical image editors, namely the 'IPM editor' and the 'Bond graph editor'. These editors allow entry, analysis and modification of main models and specifications of subsystems in the iconic diagram and bond graph formulation respectively. Initial implementations of these editors have been described by Breunese (1992), but major revisions have since been done in order to more rigorously incorporate the separation between model description and model representation. The appearance and functionality of these editors will be shown in the next section.

#### *Equation editor*

The 'Equation editor' supports the entrance and modification of specifications of subsystems in the form of equations, i.e. elementary model specifications. Details about the design and implementation can be found in Evers (1993) and in Breunese (1993b).

#### *Library browser*

The 'Library browser' enables the user to browse through the libraries of subsystems and components. Furthermore, it is the means by which the user can modify the organization of the library. Figure 6.2 depicts the tool. Meindertsma (1992) discusses

implementation issues related to this. The library of design solutions plays a major role in supporting explorational design. Therefore, the library incorporated in MAX will be discussed in more detail hereafter.

### ***Icon editor***

By means of the ‘IPM image editor’, the designer can create and modify the icon that is to be used for a type in the iconic diagram formulation, including terminal definitions and allowable orientations of the icon. Currently, the IPM image editor is a bit map editor, but work is being done to migrate it to a vector-based drawing tool. The design and implementation of this tool has not been given particular attention; it merely has been created on a pragmatic basis.

### ***Type editor***

Using the ‘Type editor’, the model builder can describe a subsystem type in a representation-independent way. As explained in chapter 5, all attributes of a type are port-related, except for parameters. The port-related attributes that can be defined in the type editor are: port name, dimension, orientation preference, causal preference and domain. Implementation details are described by Meindertsma (1992).

### ***Organizer***

The ‘Organizer’ is the part that informs the user about the current problem state. To that end, it can show to the user the part-of hierarchy of the core model and the model contained in some editor(s). Also, this tool is the central point for detailed error messages, and it lists which other tools are active. Finally, all other tools except for the help system can be activated from here.

### ***Help system***

On user request, a hyper-text based ‘Help system’ (part of the informer) provides context-sensitive help about how to interact with MAX. It does not give advice on how to proceed with the session. Implementation of this tool is described by Evers (1993).

Herewith, all tools that are contained in MAX have been outlined. In addition, design automatons are available that are activated by means of issuing commands from tools. The following kinds of automatons are implemented:

- ‘*graph transformers*’. These translate and transform the core model to language specific descriptions and vice versa. Currently, this is realized for two model formulations: bond graphs and iconic diagrams. In chapter 4, the algorithms underlying the implementation were discussed in detail.
- ‘*visualizers*’. These realize the visualization from language specific description to language specific model (see chapter 3). Only for the iconic diagram formulation is a visualizer fully operational, although it still needs improvement. For the bond graph formulation, a visualizer is under construction. Wijsman (1992) provides information about implementation.
- ‘*constraint propagation machines*’. These are mainly applied for performing causal analyses of bond graph models. Causal analysis is a non-trivial issue that

needs attention, also in a design context (Van Dijk, 1994). Constraint propagation machines also are used for the purpose of orientation analysis (chapter 4). Nevenzel (1992) and Geertsema (1993) discuss the implementation of causality analysis using constraint propagation machines in detail.

The manager that is implemented in MAX mainly takes care that models are consistent throughout the system. This subsystem also coordinates the available tools. The bases layer has been implemented straightforwardly in the form of a simple file system, both for the repository and the library. For a library of a size required for real design applications, this is definitely not sufficient. The language layer, finally, includes parser-compilers for importing models from and exporting models to the CAMAS modeling and simulation environment (Broenink et al., 1992).

### 6.3.2 Hierarchy of subsystem types

The hierarchy of subsystem types in MAX is such that creation of useful alternative models by means of sequences of ‘generalize’ and ‘specialize’ upon a type is supported. Furthermore, types were given names from a design perspective, i.e., functional and not physical. This means that a component is expected to *mainly* behave conform the stated *function*, although other physical effects may be incorporated in the specification. Finally, emphasis was laid on types for (controlled) electro-mechanical subsystems.

Three hierarchies are currently available:

- 1 *signal processors* hierarchy: this hierarchy contains subsystems of which the functionality mainly is to process signals in some way. This also implies that these subsystems mostly only have signal ports.
- 2 *components* hierarchy: in here, models of physical components are contained. This is the most extensive hierarchy. Figure 6.2 depicts an illustrative part of this hierarchy.
- 3 *special* hierarchy: in this hierarchy, model parts that are specific for a certain formulation are contained. These are typically knots. Also, ports are incorporated in here. This hierarchy will not expand much; only if new formulations or new physical domains are considered does it need adaptation.

The type hierarchy was designed largely according to the guidelines given in section 5.7.2.

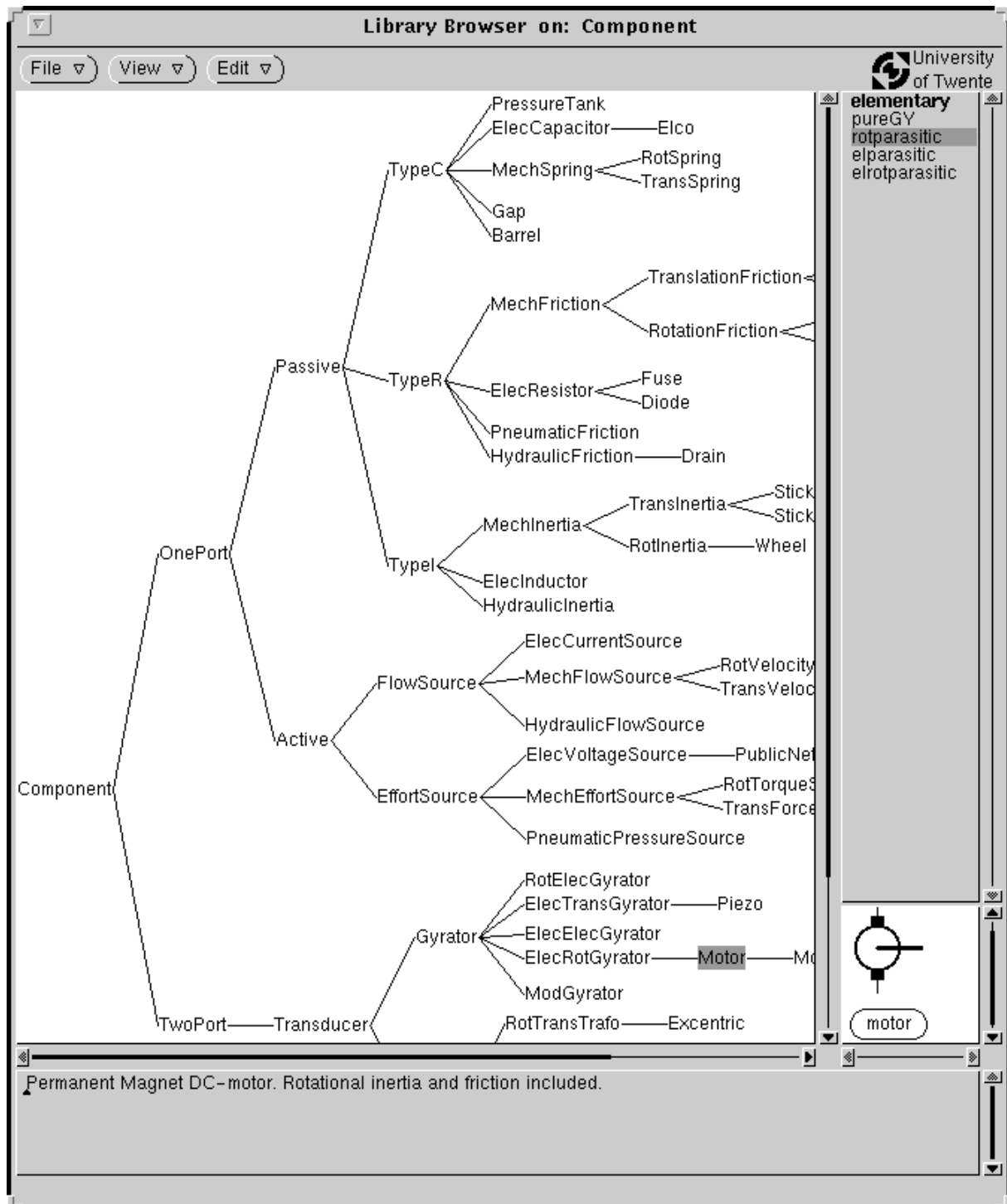


FIGURE 6.2 *MAX library browser*  
 Top left: hierarchically organized subsystem types  
 Top right: list of specifications of the selected type  
 Middle right: available representations of the selected type  
 Bottom: comment about selected type and specification



## 6.4 Case study

Hereafter, a relatively small design problem is presented that shows how MAX can be applied in a design context. This is sufficient to demonstrate the functionality and utility of the system. More realistic test cases, i.e. for larger problems in an industrial environment, are currently being studied. Preliminary results can be found in Kleijn (1993).

The problem was to design a mechatronic drive system to be used by third year master students to let them gain some experience in the design and implementation of a practical controller. This problem was given to two third year master students, who had to come to a documented conceptual design proposal within the period of 250 hours (Bolks, 1993). The following list gives the main requirements:

- low cost
- no maintenance
- clearly observable difference between controlled and non-controlled performance
- robust to misuse
- safe
- variable stiffness and load
- choice between linear and non-linear behavior of the load

Figure 6.3 depicts the initial design proposal, entered in the iconic diagram editor.

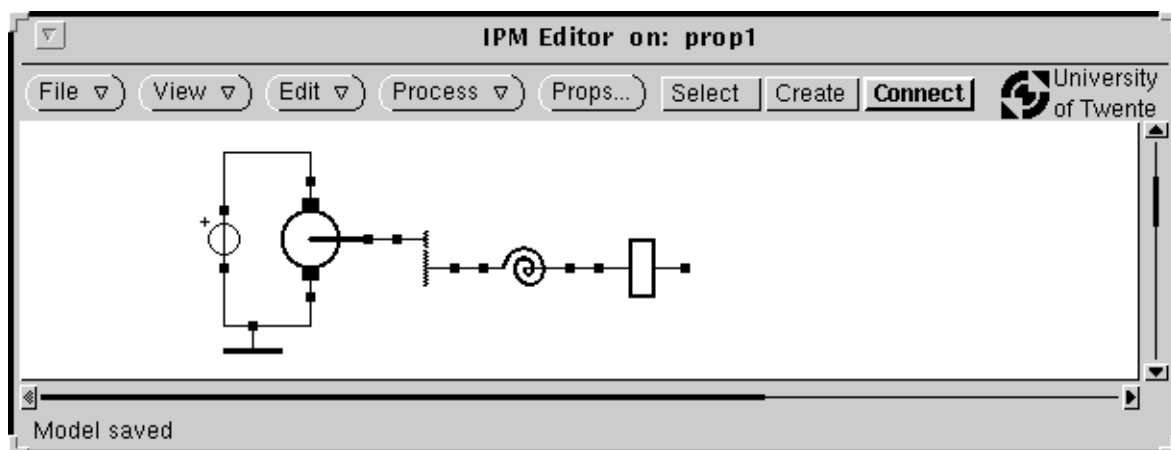


FIGURE 6.3 *Initial design proposal*

Subsequently, the model was transformed to a bond graph formulation, see figure 6.4.

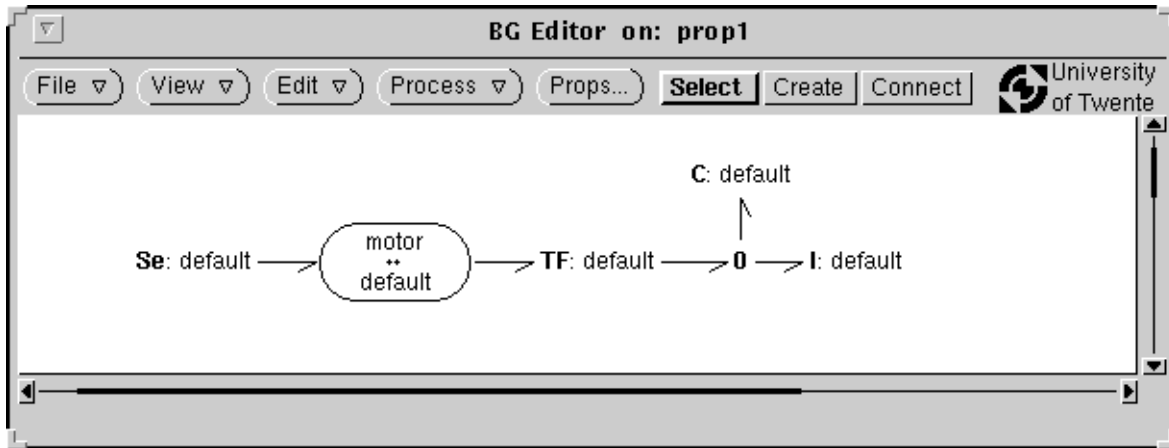


FIGURE 6.4 *Initial proposal in bond graph formulation*

Alternative solutions, whereby parts of the system are realized in other domains, were systematically found by subsequently generalizing and specializing parts of the bond graph. A good candidate for this operation was the flexibility in the transmission (the C that represents torsional stiffness): it might be easier to vary the flexibility if it was realized in the translation mechanical domain. This solution was created from the available one by introducing a transformer between the torsional stiffness and the load, and after that changing the domain of the bond graph fragment between the two transformers to the mechanical translation domain. Figure 6.5 depicts the result.

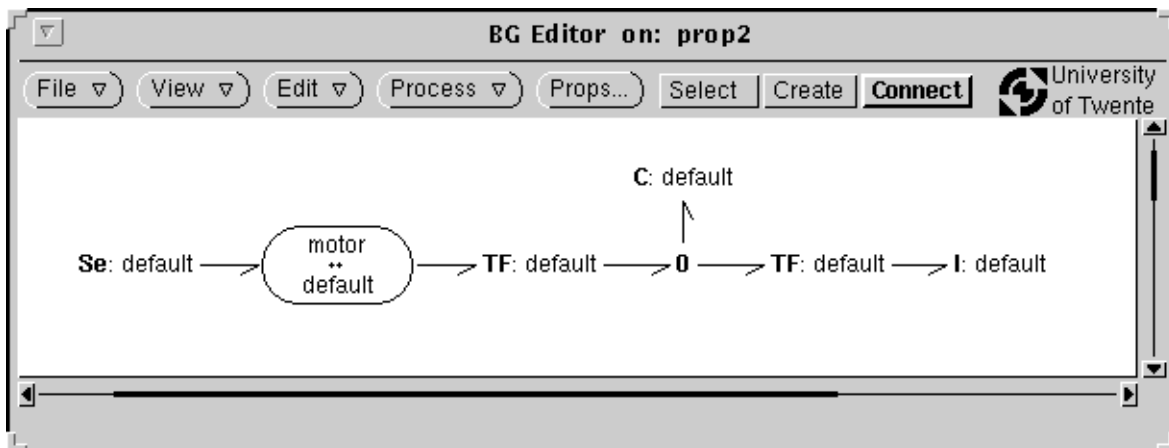


FIGURE 6.5 *Alternative solution, systematically created by means of subsequently generalizing and specializing model parts*

The newly created proposal was transformed back again to the iconic diagram formulation, see figure 6.6. It showed that the solution could be realized as two pulleys coupled by means of an elastic belt. This solution had certain advantages compared to the initial proposal; variable stiffness and load can be obtained easily by incorporating

two pairs of pulleys with differing transmission ratios, and/or by varying the number of elastic belts that couple the pulleys. This seemed an elegant way to also satisfy the robustness, safety and cost requirements. Therefore, this solution was selected to work out further.

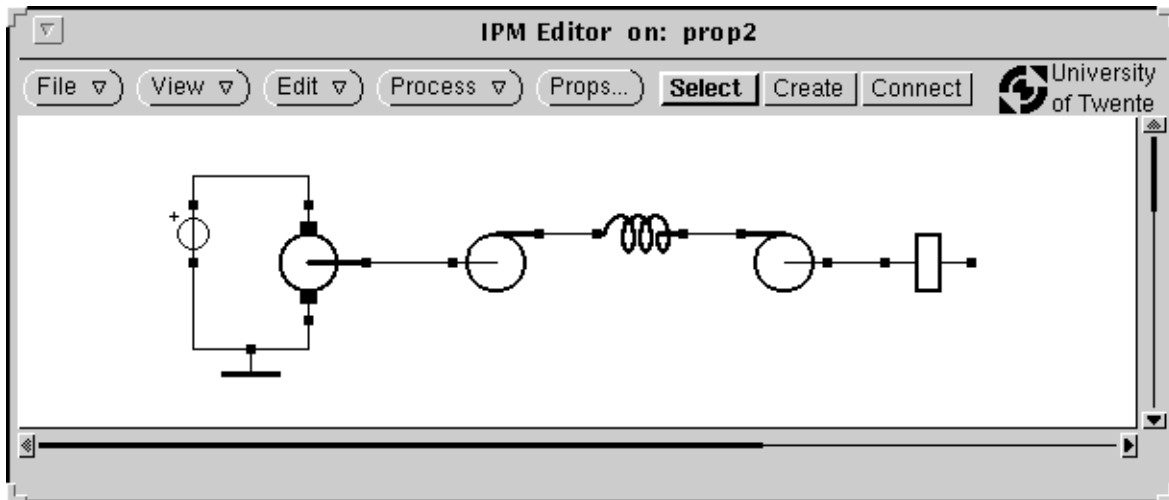


FIGURE 6.6 *Alternative solution, iconic diagram formulation*

Now that the major configuration were known, a proper motor could be chosen. Again, alternative solutions were systematically found, this time by inspecting and creating proper motor specifications with differing behavior. Specifications for a Permanent Magnet DC-motor and various types of wound motors were loaded in the library. Because the PMDC motor is the least expensive and meets the performance requirements, this option was chosen. A more detailed model of the design proposal resulted (see figure 6.7).

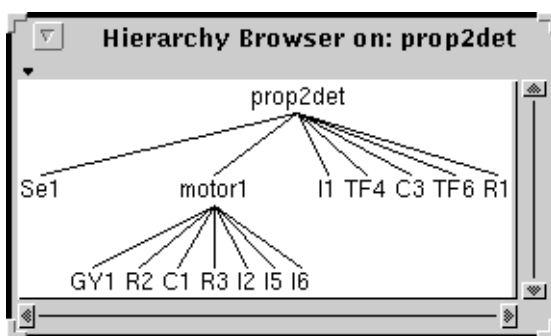


FIGURE 6.7 *Motor selected systematically by specifying it to a PMDC*

The idea was to come to a resonance frequency of the load that was clearly visible, i.e. about 3 Hz. On basis of this, parameters were given a roughly estimated value. To evaluate the system response in the time domain and the sensitivity to parameter

variations, simulation experiments could be done. To this end, the refined model was transformed to a bond graph, and made causal (figure 6.8).

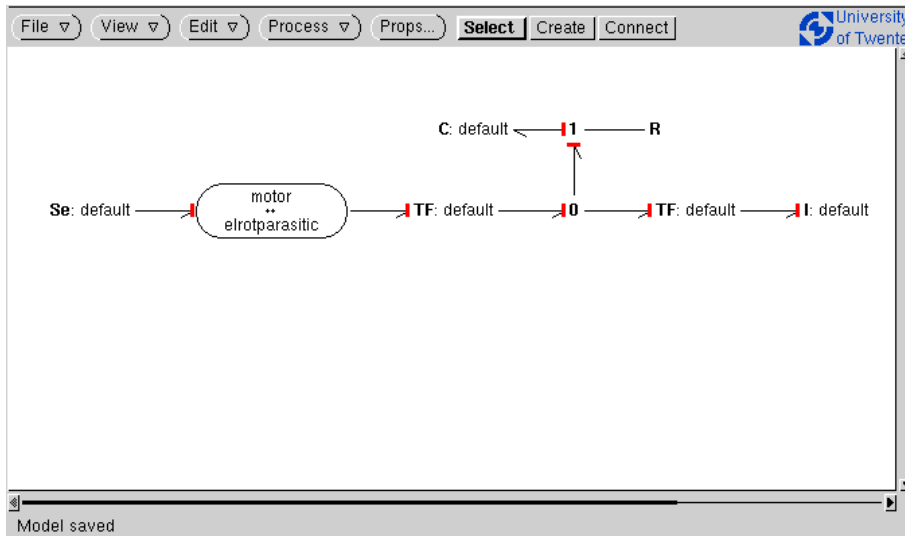


FIGURE 6.8 Causal bond graph model of the selected alternative

This causal model was exported to CAMAS, and simulations were done (figure 6.9).

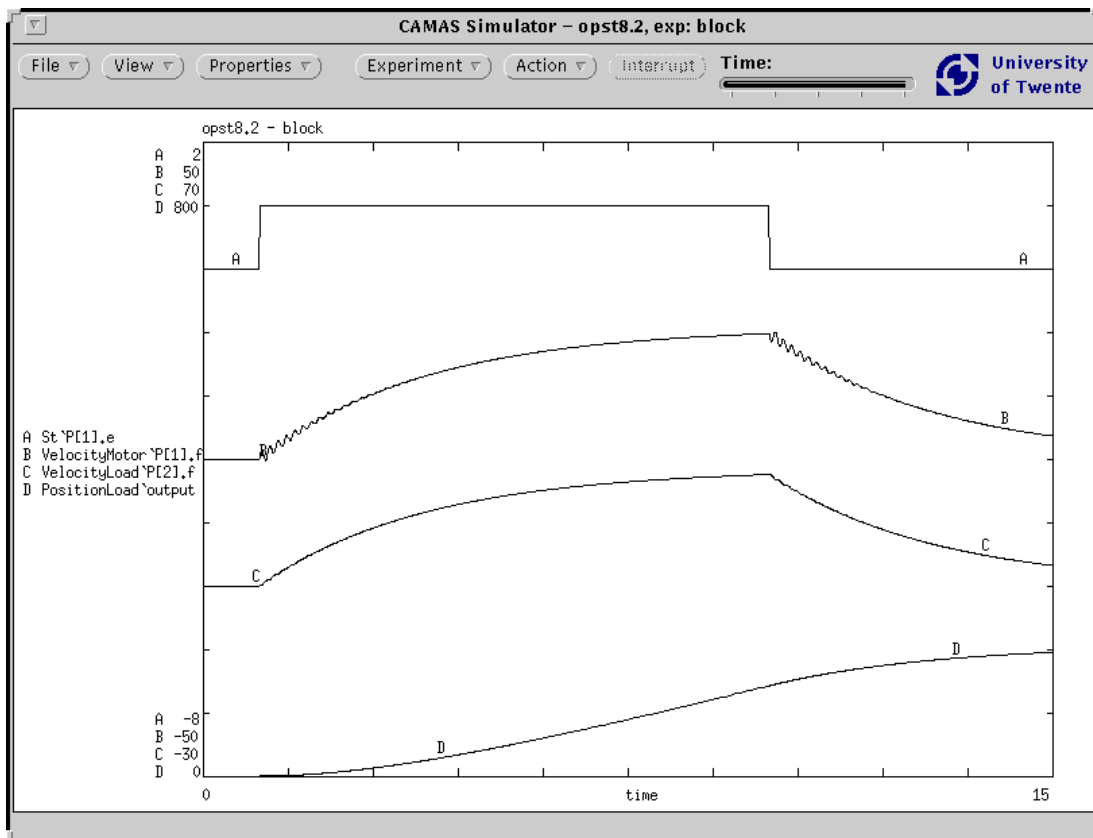


FIGURE 6.9 Initial simulation experiment for the selected alternative

The first conclusion following from these experiments was that the selected parameter values indeed gave the system the desired resonance frequency. However, the second conclusion was that the simulations do not match with the behaviour that was expected for this system: the inertias at the motor side were vibrating, rather than at the load side.

Closer inspection of the causal bond graph model of the motor (figure 6.10) revealed the cause of this: the gyrator inside the motor, which transduces the energy from the electrical to the mechanical rotation domain, was acting as a torque source (i.e., current controlled), rather than as a velocity source. The cause of this unwanted causality was the fact that the electrical inductance of the motor was incorporated.

FIGURE 6.11 *Causality of the specification of the motor*

By using a true servo amplifier, the bandwidth of the actuator can be enlarged such that it becomes negligible. In that case, the electrical inductance can be left out of the model of the motor. The motor model was adjusted accordingly by changing its specification. Simulation experiments with this refined model confirmed the expected and desired dynamic behavior (figure 6.11). Hence, detailed design was started for this design proposal.

## 6.5 Evaluation

In this section, strengths and weaknesses of the current implementation of MAX are outlined. Also, desirable extensions are indicated.

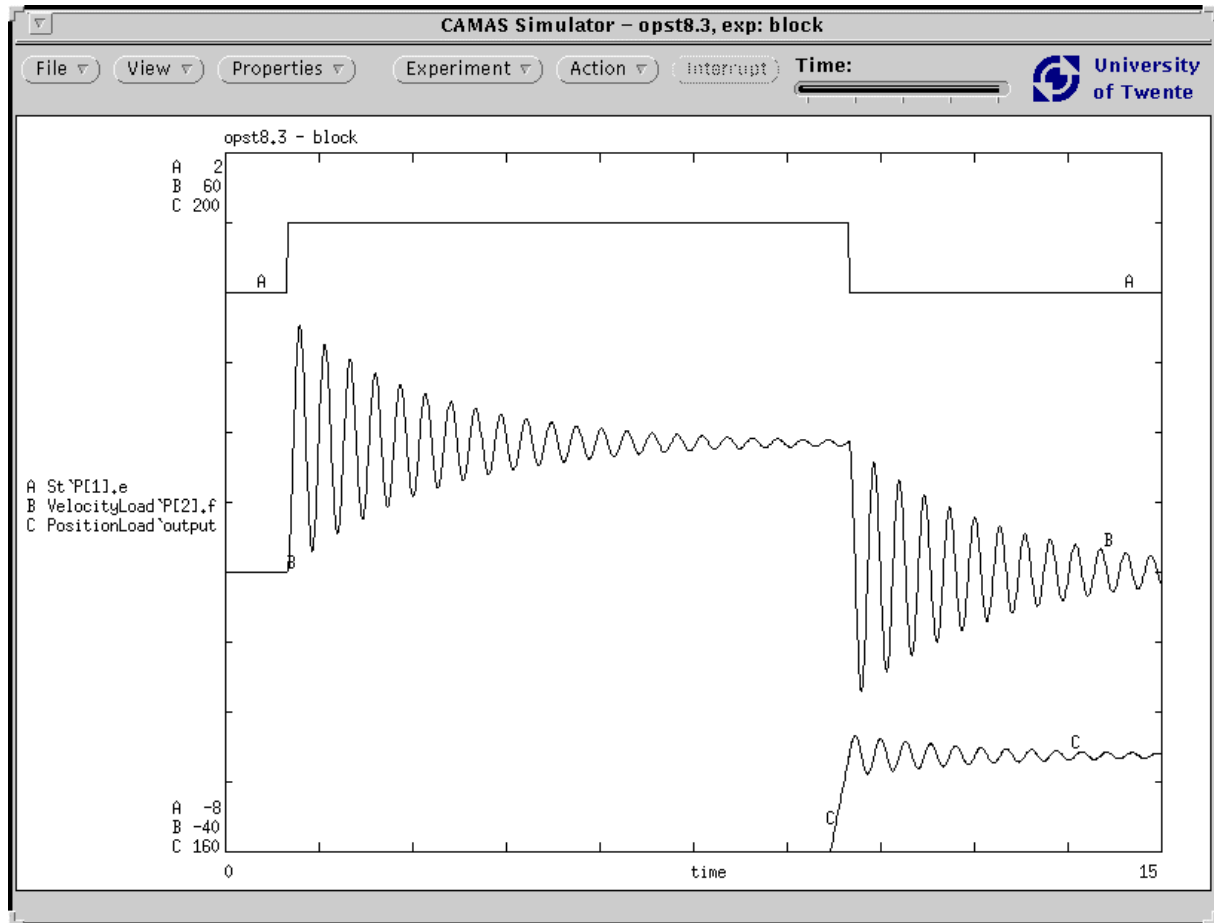


FIGURE 6.11 *Simulation experiment with the causally correct model*

### 6.5.1 Favorable points

#### *MAX works*

The first point is that MAX actually works in the way that was envisioned. It is ready to be submitted to further tests in industrial design practice and in educational environments.

#### *Use of the model of design*

With respect to the typical deficiencies of computer-based design support systems (see chapter 2), the following can be noted. The MAX system provides an environment for model building that is conceptually clear and does not constrain the designer to particular model building methods. Moreover, it features a clearly defined, modular internal structure that is well suited for incremental extension of its functionality and for integration with other systems. Finally, major development choices have been backed up by theoretical investigations done in a design context, and not in a tool building context. We assert that these benefits are due to the fact that the system was based on a formal model of designing that both reasonably well describes our understanding of the design process, and is well suited for the development of support.

### ***Modeling the conceptual design object***

MAX allows the designer to describe and observe the design object at different levels of concreteness, ranging from the functional level (causal word bond graph) to the initial form level (iconic diagram model). Within this range, models can have any resolution. Also, multiple alternatives can be considered. The only requirement for fully supporting a model of the design object during conceptual design that is not met is the possibility to vary the precision; parameters have either no value or an exact value. What is especially interesting for designing with an integrated problem solving approach is that MAX enables users to simultaneously inspect the design object from multiple perspectives by means of opening multiple editors on one and the same model.

### ***Polymorphic model refinements***

In the case study, it has been shown that polymorphic refinements of the model (i.e. generalize–specialize and specify) can be fruitfully applied during model building. These mechanisms allow the user of MAX to incrementally expand and revise the model. Consequently, the model builder can start solving the problem with a simple model of a partial solution, and gradually add relevant detail. This achievement is more significant than the fact that extensive models can be built quickly, using off–the–shell components from the library. Several researchers have claimed that building extensive models with little effort would be a major benefit of computer–based design support systems, as more detail could be considered earlier on in the design process (Roozenburg, 1993). Rather, the contrary is true; one of the major dangers of computer–based design support is that such systems stimulate or demand the consideration of too much (irrelevant) detail, which merely obscures the real issues and is a waste of precious time.

### ***Multiple model formulations***

The case study also has illustrated that MAX fully integrates the iconic diagram formulation and the bond graph formulation of a model by means of bidirectional transformations. The model builder has complete freedom in choosing the formulation he or she desires for a particular task, without having to later re–enter the same information. It appears that iconic diagrams are particularly useful for synthesis–oriented tasks. This is probably due to the fact that this formulation has a non–static vocabulary and is to some extent context–sensitive; these characteristics indicate that in the iconic diagram formulation one is confronted automatically with the question of whether a certain model fragment is appropriate for the problem at hand or not. The bond graph formulation is especially powerful in analysis–oriented tasks. Hence, the integration of these two formulations is an important contribution, as it implies that models suitable for synthesis tasks have been integrated with models for analysis tasks.

### ***Problem understanding***

Looking back at the overall process of the case study, it can be concluded that the intermediate results of transformations and polymorphic refinements and the structural analyses that can be done in MAX have helped to learn quickly and efficiently about

the actual design problem and proposed solution. That is, they have helped to properly classify the design problem and to verify correctness, consistency and suitability of the proposed solution in early stages. As a result, more informed decisions could be made earlier on in the design process. Hence, MAX improves problem understanding throughout the process.

### ***Enhancing communication***

Due to the availability of a well-structured, extendible subsystem library and the flexibility of model manipulation, little effort and time is required for model building in MAX. Consequently, the designer is less discouraged to make a well-defined, formal model of the solution that is proposed. Communicating a solution with other designers is easier on the basis of such a formal model, especially since these other designers are not forced to use the same representation. Therefore, it can be concluded that MAX also enhances communication. As a result, more alternative solutions can be considered and proposed alternatives can be better judged on their merits than before.

### ***Evolutionary nature of design***

Systems have been described in literature that are comparable to MAX in terms of objectives and functionality, such as SCHEMEBUILDER (Bradley et al., 1993; Sharpe and Bracewell, 1993), QUBA (Top, 1993) and SYSFUND (Tomiyama et al., 1989). All these are interesting and to some extent have been influential on the ideas incorporated in MAX. However, none of these systems is able to support the evolutionary nature of design as well as MAX does. This is because these systems have not combined polymorphic modeling and multiple model formulations (or comparable concepts). It is this combination that gives MAX the possibility to conform to the evolutionary design process, and to suggest a systematic search for alternative design solutions.

## **6.5.2 Weaknesses**

### ***Constructing the subsystem type hierarchy***

A weak point of MAX is that classification of subsystem types into a kind-of hierarchy in the library is not straightforward. In chapter 5, advice has been given on how to deal with this matter. However, this advice is of limited value; it does not suggest a rigorous classification for simple subsystems, let alone for complex subsystems with many ports (Aalbers, 1993). However, obtaining a good type hierarchy is important, as it has a major influence on which refinements can be easily made by means of the generalize-specialize mechanism and which are more tedious. The explanation is that unique classification is not possible; it depends on user and problem context. Top (1993) asserts that the problem of making a proper classification in MAX is caused by the fact that different ontologies (conceptual knowledge structures, Alberts, 1993) are forced into one hierarchy. However, even when considering one ontology (bond graphs for example), classification is hard (Karnopp and Rosenberg, 1968). Top's remark is correct to the extent that it is not fruitful to force subsystems that do not have ports of the same kind into one hierarchy. In MAX this is also supported, because different



hierarchies can coexist. For example, subsystems that process signals and hence have only signal ports are contained in another hierarchy than components having energetic behavior.

There are three ways to deal with the difficulty of classification:

- 1 don't restrict relations between types to subtyping and single inheritance. This is in fact what was suggested in chapter 5. However, it is not obvious which alternatives should be included and how this will influence the model refinements that can be made. This solution needs further research and experimental evaluation.
- 2 automate the classification of types, for example on the basis of optimizing reuse of type definitions. At first glance, this seems to be an attractive solution, but it is expected that it will eventually lead to bad type hierarchies.
- 3 allow each user to make a personal (or even problem class specific) hierarchy, and make sure that communication across systems of different users and across different projects remains possible. This can be imagined as a way of customizing MAX to individual preferences. It is a 'solution' that does not require any action, and it allows us to gain some experience before deciding on definitive measures. Therefore, this option is preferable for the time being.

### ***IPM image editor***

When working with the system, one soon learns that the major bottleneck in MAX is currently the IPM image editor. Drawing icons with this tool is tedious and time-consuming. The tool suffers from the same deficiencies as conventional CAD systems: it is not based on a proper understanding of the task it supports. Consequently, it works in a system-oriented way rather than in a designer-oriented way. This is mainly due to the fact that the design of this tool has not yet been given much attention.

### ***Required level of expertise***

MAX indeed has become an expert system, in the sense that it gives valuable feedback to knowledgeable users. It is expected that more ignorant users are overwhelmed quickly because of the seemingly unstructuredness that is the result of the flexibility offered by MAX. Also, the high level of abstraction on which MAX communicates with the user demands knowledgeable users.

## **6.5.3 Desirable extensions**

Many useful extensions to MAX can be imagined. Here, only a few of the more important ones are indicated.

- the subsystem library should be enlarged extensively with models for components and principle design solutions for relevant areas (for example those given by Koster (1993) for mechanical constructions). Work in this area is currently being done (Olmeco Consortium, 1991).

- additional modeling languages should be added to the system, to start with a functional language.
- the modeling kernel of MAX should be extended such that multi-dimensional bonds are supported as well.
- MAX should be equipped with tools such as are needed for other modes of design, so that it will not force the usage of the explorational design mode.
- manipulation and comparison of alternative design solutions is currently not supported well; improvements are desirable here.
- to make MAX more suitable for the explorational mode of design, a documentation system should be added to the subsystem library. This can be combined well with the above two extensions (Prins and Olthoff, 1993).

## 6.6 Conclusions

The model of designing as formulated in chapter 2 has been useful while developing MAX in the following ways:

- it has enabled a systematic identification of support required in a designer's workbench
- it suggested an organization of the identified subsystems into a framework
- on basis of the model, a useful way of combining subsystems into tools has been found, and a reference for development choices for individual subsystems has been provided.

Polymorphic modeling and multiple modeling languages can be implemented within the framework by incorporating two structuring principles for model storage in the system: separation of subsystem type and subsystem specification, and separation of subsystem description and subsystem representation.

Contributions of MAX are the following:

- 1 it provides the designer a means to concurrently consider the integrated system at different levels of abstraction, ranging from not concrete (i.e., functional) with low resolution to concrete, with high resolution.
- 2 it gives the designer an unequaled flexibility to manipulate the description of the design. Yet, these manipulations are well-structured and require little effort and time.
- 3 it allows the designer to build descriptions of the design object in synthesis-oriented terms, and to evaluate this same design object by means of powerful analysis tools, such as causality analysis, orientation analysis and simulation. In other words, it integrates models suitable for synthesis tasks with models suitable for analysis tasks.

Intermediate results of transformations and polymorphic refinements are essential in order to learn about the actual design problem, that is, to check validity and completeness of the problem statement and to verify correctness, consistency and

suitability of the proposed solution. As a result, more informed decisions will be made earlier on in the design process.

The little effort and time that is required for realizing transformations and polymorphic refinements is essential in order to communicate about the actual design problem, that is, to describe and represent the proposed solution. As a result, more alternative solutions can be considered and proposed alternatives can be better judged on their merits than before.

The improved support for learning and communication will lead to a shortening of the design process and an improved final design. These benefits will especially become apparent when an integrated design approach is applied.

A weakness of MAX is that classification of subsystem types into a kind-of hierarchy is hard, especially for complex subsystems with many ports. The underlying problem is that classification is context-dependent, and hence an optimal classification cannot uniquely be determined. For the time being, it is proposed to deal with this by allowing each user to make a personal (or even problem class specific) hierarchy, and to make sure that communication across systems remains possible.

Overall conclusion is that MAX is a powerful model building environment that is well adapted to usage by designers. Hence, it complies well to its main development objective; it is suitable to serve as an extendible modeling kernel for a mechatronic designer's workbench. MAX is not a full size design system yet; to become that, it should be further extended.



## Discussion

### 7.1 Synopsis

The engineering design process was analyzed in detail for the purpose of developing advanced computer-based design support. This analysis has been motivated by the hypothesis that computer-based systems will be better able to enhance designing if they are developed on the basis of a model of designing that explicitly reflects what is understood of the design process.

A model of designing was obtained in the following way. We characterized designing as a contextually situated, evolutionary process. We categorized existing models of designing by means of a taxonomy and reviewed a well-known model contained in the most suitable category, namely the Task/Episode Accumulation model (Ullman et al., 1988). In this review, we identified reasons why this model is not straightforwardly applicable for our purpose, and subsequently formulated a new, more applicable model that incorporates the TEA model.

The new model of designing made clear that the conceptual design task is of crucial importance when designing with an integrated problem solving approach. Also, it clarified that abstractions contained in the design object play a crucial role in this task. However, systems that properly support the maintenance and manipulations of these abstractions are lacking. Therefore, we have firstly addressed the question in what terms models should be made, i.e. in what way we should formulate models in a computer-based system. Secondly, we have investigated how computer-based systems can (should) support the creation and modification of abstractions.

The language in which a model is formulated determines what information can be expressed in a model, and whether it is easy or not to observe and interpret the information that is incorporated in the model. It has been argued that when using an integrated design approach, it should be possible to simultaneously formulate one model in multiple languages, in such a way that the model can be manipulated in any of the formulations. This concept was called multiple model formulations. We have devised a system setup that enables multiple model formulations and yet will keep different formulations of a model consistent and tractable. This setup specifies that different formulations are coupled to each other through a central core model. We have

taken bond graphs and iconic diagrams as an example set of formulations for investigating whether the setup is realizable.

To support model building, computer-based systems should provide powerful means for decomposition, classification and representation. It was shown that there are three implementation techniques currently exploited in this sense: parametrization, typing, port-based interfacing. These techniques together give modeling tools the capability to support decomposition and representation of models by means of powerful concepts, such as reticulation and hierarchical modeling. However, adequate support for classification has not been available. Improving this requires the use of additional techniques, namely modularization of subsystem descriptions into a type and a specification, and subtyping of subsystem types, i.e. expressing a type as a specialization of a more general type. It was explained that the combined application of modularization and subtyping (called polymorphic modeling) leads to hierarchical subsystem libraries and gives modeling systems the possibility to conform to the evolutionary nature of model building. System design issues related to modularization and subtyping were discussed and some general advice on how to apply polymorphic modeling was given.

Finally, we discussed the development and functionality of the MAX system. MAX is a model building environment for controlled electro-mechanical systems; it supports the user in creating models and evaluating them by means of network-based analyses. A main objective underlying MAX is to form the modeling kernel of a mechatronic designer's workbench. Therefore, its development was based on the model of designing developed in this thesis. Multiple model formulations (i.e. bond graphs and iconic diagrams) and polymorphic modeling are incorporated in the system. The state of the art of MAX was presented and the utility of the environment was shown by means of a case study.

## 7.2 Conclusions

In the course of the reasoning outlined above, the following points have been made.

### *Model of designing*

Designing has to be viewed as a contextually situated, evolutionary process, because:

- design is context dependent.
- design problems are ill-structured and incomplete.
- design involves an initiation of change, to be realized within time-constraints

A descriptive, object-oriented and concurrent model of designing is best suited for explaining how designers are able to design. However, such models in general and the TEA model specifically cannot be applied in the development of design support straightforwardly, because:

- inconsistencies may arise in systems based on these models (e.g., the design object is modeled in a way that does not match with design actions the user wants to take).
- the evolution of the design state has not been worked out (e.g., it is unclear in what way the information incorporated in the design state develops in the course of designing).
- context-dependent and context-free parts are not distinguished.

The newly proposed model of designing explicitly presents an overall view upon designing, which helps to prevent inconsistent or incompatible assumptions. It describes a pattern how context might influence the design process and how the evolution can be given a framework. It consists of a context-free basic model which relates the design object, the design process and design knowledge (figure 2.7). This basic model clarifies that designing involves a specific form and combination of learning and communication. Additional context-dependent features have been incorporated in the model by further characterizing the design object and propagating the obtained features through the rest of the basic model. Evaluation of the new model showed that it is of a unifying nature, that it clarifies how designers can deal with the context-dependent and evolutionary nature of design and that it enables to explanation and prediction of how designing can be enhanced.

There are three different ways to enhance designing: by formalizing design knowledge, by automating design activities on basis of formalized knowledge and by creating practical computer-based systems that incorporate formalized knowledge and automated activities. Computers do not possess the communication- and learning capabilities that humans have, and therefore cannot tackle realistic design problems autonomously. This implies that enhancing design by means of practical systems requires systems that support designers in their communication and learning processes. In communication and learning, the use of abstractions plays a major role. Therefore, modeling capabilities of design support systems need to be improved, specifically for support of the conceptual design task.

### ***Multiple model formulations***

In order to keep multiple model formulations consistent and tractable, a system should be set up in the following way (figure 3.12). The user has access to a formulation through a language specific view of the design object in an editor. This view is created by adding protection to a language specific model such that the user can only issue commands that have visible effects in the view itself. The (graphical) language specific model results from a visualization of a (textual) language specific description that only stores intrinsic model properties. The language specific description finally is linked with a central core model by means of a bidirectional transformation. The core model integrally stores the information needed for all different model formulations, and coordinates the different language specific descriptions.

For conceptual design, a set of graphical languages is required that at least enables description of function, behavior and initial form. In the domain of controlled electro-

mechanical systems, the set consisting of iconic diagrams, bond graphs and the THESIS formalism (a variant of the Modern Structured Analysis notation) seems appropriate. Feasibility of the system setup has been shown for iconic diagrams and bond graphs.

### ***Transformations***

To be able to transform an iconic diagram, it was necessary to identify formal rules that are incorporated in this formulation. These formal rules relate to the usage of references and terminals. When transforming an iconic diagram into a bond graph model, the main problem is to properly choose orientations for bonds in the bond graph. This requires an orientation analysis prior to the actual transformation. During orientation analysis, the iconic diagram is checked upon correctness and the power flow through the connections of the diagram is given an orientation.

During transformation of a bond graph model to an iconic diagram, the following three issues appear:

- ordering: the order of components that will appear in a series connection in an iconic diagram is constrained, but not explicitly available in a bond graph. Therefore, it needs to be generated appropriately.
- cycle treatment: bond graph junctions contained in a cycle have special properties, that have to be regarded.
- global references placement: bond graphs generally do not include knots that represent global references. Hence, these need to be generated during transformation and included correctly in the iconic diagram.

An algorithm that deals with these issues and automatically generates a correct iconic diagram from a bond graph was described.

### ***Polymorphic modeling***

Computer-based modeling tools that incorporate parametrization, typing and port-based interfaces allow building models as networks of encapsulated, reusable subsystems that are explicitly classified. However, generic subsystems cannot be defined in such systems. The use of subtyping and inheritance, such as common in object-oriented programming, does not solve this, because the internal structure of subsystems can generally not be abstracted into generic subsystem types. Generic subsystems can only be described if subtyping is combined with modularization. Modularization in this context means that a subsystem definition is divided into two parts: a type that defines essential properties, and a specification that defines incidental properties. By allowing one type to have more than one specification, subsystem types become polymorphic (figure 5.6). The combination of subtyping and modularization results in a hierarchical subsystem library that has a conceptually clear and coherent structure. Furthermore, it facilitates the manipulation of a model, like the creation of analogues of models and the variation of detail incorporated in models.

### ***MAX***

Contributions contained in the MAX system are the following:



- 1 it provides the designer a means to concurrently consider the integrated system at different levels of abstraction, ranging from functional and with a low resolution to configuration and with a high resolution.
- 2 it gives the designer an unequalled flexibility to manipulate the description of the design. Yet, these manipulations are well–.
- 3 it allows the designer to build descriptions of the design object in synthesis–oriented terms, and to evaluate this same design object by means of powerful analysis tools, such as causality analysis, orientation analysis and simulation. In other words, it integrates models suitable for synthesis tasks with models suitable for analysis tasks.

Conversions between model formulations and polymorphic refinements enable learning about the actual design problem. Little effort and time is required for realizing conversions between model formulations and for polymorphic refinements, which enhances communication about the actual design problem. This improved support for learning and communication leads to a shortening of the design process and improved quality of final designs. These benefits especially become apparent when an integrated design approach is applied. Major difficulty of using polymorphic modeling (and thus of MAX) is that classification of submodel types into a tree–like kind–of hierarchy is hard, especially for complex subsystems with many ports. However, the overall conclusion is that MAX is a powerful model building environment that is well adapted to usage by designers.

### **7.3 Suggestions for future work**

In chapter 2, a new model of designing was presented. We claimed that the model is of a unifying nature, without working this out in detail. An interesting and theoretically relevant direction for further work would be to extend the new model by more explicitly relating it to available models of designing. More specifically, this should be aimed primarily at understanding how the conception of a design problem and its initial solution come into being. This requires the conceptual world part and the observer contained in the model to be described in more detail, for example by looking into models from the area of cognitive psychology.

Polymorphic modeling as described in chapter 5 has been based on a combination of modularization and a single inheritance subtyping mechanism. It has appeared that the use of single inheritance gives problems during classification of more complex subsystems, as it is not expressive enough. However, multiple inheritance is not applicable; rather, the solution should be pursued in the direction of other subtyping mechanisms. Further research in this area is needed before definite answers can be given.

In relation to multiple model formulations, the major extension that is desirable is to incorporate a functional language (e.g. THESIS formalism, block diagrams or linear graphs) into the mechanism. This implies that the formalism in which the core model

is stored has to be adjusted and that additional conversion algorithms have to be specified. Furthermore, it would be interesting to make the transformation algorithms that were presented work in an incremental fashion. Also, visualization of model formulations can be improved. Finally, an investigation should be started into formalisms that support dynamic contraction and unfolding of the part-of hierarchy of a model, thereby adjusting the decomposition of the model.

Numerous directions can be imagined in which the work on MAX could continue. Of a high priority is the extension of the subsystem library so that it actually contains significant declarative design knowledge. This is currently being addressed in the OLMECO project. Other significant improvements that are desirable are to enable the use of multi-bonds throughout the system, to support the process of selecting optional solutions and deciding which one to utilize appropriately, and to enhance the maintenance and manipulation of alternative models.





## Simplification of bond graphs and iconic diagrams

In this appendix, rules are given for simplification of junction structures, both for bond graph models and iconic diagrams.

### B.1 Rule 1: Dangling junctions

A dangling junction is a junction that has no signal bond connection and one power bond connection. It can simply be removed from the graph. The power bond is removed too.

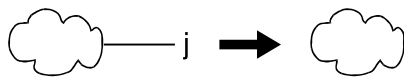


FIGURE B.1 *Removal of dangling junction*  
(*j can be a 0-junction, a 1-junction or a M-junction*)

### B.2 Rule 2: Elimination of junctions

Junctions can be eliminated from a graph if the energy flow is not branched at the junction.



FIGURE B.2 *Elimination of junctions*  
(*j can be a 0-junction, a 1-junction or a M-junction*)

This means that the junction is connected to two bonds having the same dimension as the junction itself. In a bond graph, one of the bonds should be directed to the junction, the other from the junction. There should be no signal bond connected to the junction. The simplification consists of removing the junction and one of the two bonds. This situation is shown in figure B.2.

### B.3 Rule 3: Joining of junctions

Two junctions of the same type can be joined if there is exactly one power bond between the junctions.

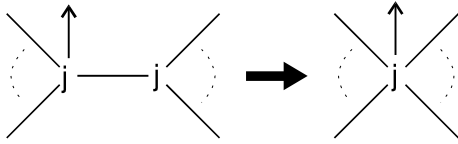


FIGURE B.3 *Joining of junctions*  
(*j* can be a 0-junction, a 1-junction or a  $\mathcal{M}$ -junction)

The simplification is carried out by removing the bond between the junctions and by transplanting all connections of one junction to the other junction. The first junction can then be removed. This situation is shown in figure B.3.

### B.4 Rule 4: Elimination of a Double Difference

This simplification rule is valid for domain attributed bond graph models only.

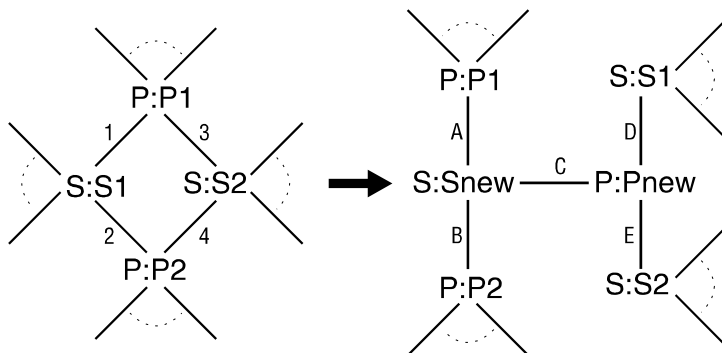


FIGURE B.4 *Double difference simplification*  
(only valid for domain attributed bond graph models)

A double difference is the situation that two S-type junctions (i.e., 1-junctions in non-mechanical bond graph fragments and 0-junctions in mechanical bond graph fragments) each are located between one pair of P-type junctions (the dual junction of the S-junction). Hence, a cycle consisting of four bonds is present. This situation can be simplified using the technique shown in figure B.4. Though the simplification actually increases the number of bond graph elements, the elimination of the loop is considered a profitable action, and it will often lead to more simplifications using rules 2 and 3. There is a number of possible orientations for the bonds involved in the

elimination of the double difference. The possible configurations and the consequences for the resulting graph are summarized in table B.1.

situation before simplification				situation after simplification				
bond 1	bond 2	bond 3	bond 4	bond A	bond B	bond C	bond D	bond E
P→S	P→S	P→S	P→S	P→S	P→S	S→P	P→S	P→S
P→S	P→S	P→S	S→P	illegal situation				
P→S	P→S	S→P	P→S	illegal situation				
P→S	P→S	S→P	S→P	P→S	P→S	S→P	P→S	S→P
P→S	S→P	P→S	P→S	illegal situation				
P→S	S→P	P→S	S→P	P→S	S→P	S→P	P→S	P→S
P→S	S→P	S→P	P→S	P→S	S→P	S→P	P→S	S→P
P→S	S→P	S→P	S→P	illegal situation				
S→P	P→S	P→S	P→S	illegal situation				
S→P	P→S	P→S	S→P	S→P	P→S	S→P	P→S	S→P
S→P	P→S	S→P	P→S	S→P	P→S	S→P	P→S	P→S
S→P	P→S	S→P	S→P	illegal situation				
S→P	S→P	P→S	P→S	S→P	S→P	S→P	P→S	S→P
S→P	S→P	P→S	S→P	illegal situation				
S→P	S→P	S→P	P→S	illegal situation				
S→P	S→P	S→P	S→P	S→P	S→P	S→P	P→S	P→S

TABLE B.1 Bond orientations in the double difference simplification





## The THESIS formalism

This appendix gives an overview of the THESIS formalism. It has been taken (with permission) from Wijbrans (1993).

A specification in THESIS consists of a *hierarchical* set of graphs and element descriptions, as shown in Figure C.1. The graphs, called *activity diagrams*, describe the structure of the system, whereas the element descriptions describe the behavior and the properties of the components. The edges in the graphs denote information flow in the system, the vertices denote processing, storage or mode-switching.

The controller can be described with three different kinds of descriptions, namely:

- The *context diagram*, a *graph* that describes the interaction with the environment (Section C.1).
- The *activity diagrams*, *graphs* that describe the structure of the controller (Section C.2).
- The *element descriptions*, that describe the elements that are not decomposed further. There are three kinds of element descriptions, that have a common format for the declaration of the interface to the activity diagram (Section C.3). These are:
  - 1 The *primitive process specification* (Pspec), the *textual* or *graphical* description of data processing (Section C.4).
  - 2 The *control process specification* (Cspec), the *textual* or *graphical* description of mode switching (Section C.5).
  - 3 The *store specification* (stores), the *textual* or *graphical* description of the buffer that stores information (Section C.6).

Only the vertices in the activity diagrams are described. The edges, called *flows*, are defined implicitly. They depend on the elements they are connected to. This deviation from the formalism of Yourdon (1989) enables reuse and information hiding.

## C.1 The context diagram

Each specification starts with a toplevel diagram, called the *context diagram*. It shows the interaction of the system with its environment. Figure C.2 shows the four different symbols that may be used on this diagram. These are:

- a The *terminator*, symbolizing an external entity.
- b The *data process* indicating the processing of information; on the context diagram it symbolizes the complete *controller under design*.
- c The *data flow*, symbolizing the flow of information for processing, in general, these are continuous or sampled flows.
- d The *control flow*, symbolizing the flow of information for mode-switching, in general these are events.

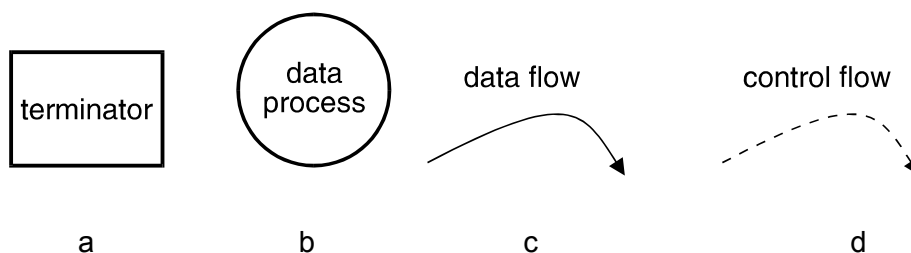


FIGURE C.2 *Symbols used on the context diagram*

The context diagram must contain one data process that symbolizes the whole controller under design. All entities in the environment that this controller communicates with must be shown as *terminators*. All communication flows, i.e., *data flows* and *control flows*, between the controller and its environment must be shown on the context diagram. No other flows are allowed on this diagram. No other information from the environment than that specified by flows may be used in the controller.

## C.2 The activity diagrams

An activity diagram describes the relationships between the components of the controller graphically. It is a graph, in which the edges show information flow and in which the vertices show processing or storage of information. Both data processing and mode switching are modeled in a single activity diagram. Separate symbols indicate the different activities, these are symbol b, c and d of figure C.2 and the following new symbols:

- e The *control process* or Cspec symbolizes the finite state machine for mode-switching.
- f The *store* shows storage of information for either data processing or mode-switching.
- g The *merge point* indicates that the contents of these flows are multiplexed.
- h The *split point* indicates that the information flowing on the input flow is copied to both output flows.
- i The *activation flow*, only allowed between a control process and a data process, indicates that the control process (de)activates the data process.

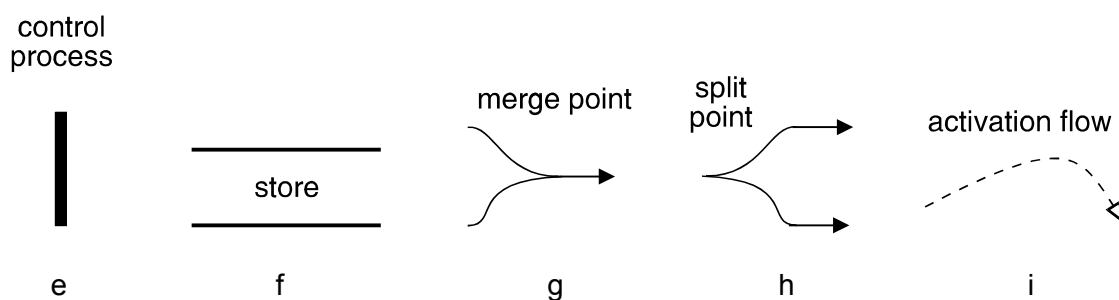


FIGURE C.3 Graphical elements on activity diagrams

### Elements

Data processing is modelled with *data processes* (figure C.2b), *data flows* (figure C.2c) and *data stores* (figure C.3f). As shown in figure C.1, a data process may either be specified by a lower level activity diagram or by a *primitive process specification*.

Mode switching is modelled with the *control flows* (figure C.2d) and the *control process* (figure C.3e). It contains a finite state machine that controls the activation and

deactivation of processes on the activity diagram. Only one control process is associated with each activity diagram. The *activation flow* (figure C.3i) shows the activation of processes by this control process.

The *merge point* (figure C.3g) consists of several flows combining into a single flow of the same data type. All data entering on the input flows is multiplexed on the single outgoing flow. In the continuous-time part only one of the input flows may be active. In the discrete-time part several of the input flows may be active concurrently. However, in general only one of the input flows will be active in a practical controller, because otherwise synchronization problems may result.

The *split point* (figure C.3h) consists of a single flow splitting into multiple flows. The data on the input flow is copied to all outgoing flows. Split points are used if the same data is needed in several processes.

### ***Structure and element attributes***

The activity diagram itself is an attributed, labeled and directed graph, i.e., attributes are associated with the edges and the vertices. These attributes define important properties of each. In the graph, vertices are defined as elements with ports. The different ports of a vertex may not have the same name.

Vertices in the graph have the following attributes:

- A *kind*, i.e., process, control specification or store.
- A *name*, that is used to identify their description.
- For processes a *number*.

The name of the vertex, and the diagram number of the graph it resides on together determine the name of its description. The description specifies the ports of that vertex.

The edges have three attributes: the *flow type*, and the names of the *source* port and *destination* port. These must be defined in the vertices that they connect. The directions of the ports have to match the direction of the edge. All other attributes of the ports, e.g., the arithmetic type and the sample frequency have to be the same.

*Ports* on an activity diagram are defined by edges (flows) that are connected at one side only. Depending on whether the input or the output is unconnected, the port is an input or an output port. These flows must have a name. This name is also the name of the port.

## **C.3 The interface declaration part**

Pspecs, Cspecs and stores have the interface declaration part in common. This part declares *types, ports and parameters*. It consists of the keyword *pspec, cspec or store*, followed by the actual port and parameter declarations. Listing C.1 gives the syntax

rules in Backus-Naur form for the interface declaration part, the ports and the parameters. In this notation, items between angle brackets ('<' and '>') denote nonterminals of the language. Items in **bold** denote reserved words and special characters. A list separated by bars ('|') denotes items of which one must be chosen. Brackets denote optional parts of the syntax, braces denote the repetition of parts. An index number with a brace denotes the minimum number of times the part within the braces must be used. Finally, parentheses are used to group symbols together.

```

<interfacepart> ::= <heading> [<typedecpart>] [<parmdecpart>] <portdecpart>
<heading>      ::= (pspec | cspec | store) <name>
<typedecpart> ::= type { <typedec> }1
<typedec>     ::= <typename> = ( <arraytype> | <recordtype> | <enumtype> | <typename> ) ;
<arraytype>   ::= array { [ <size> ] }1 <typename>
<recordtype>  ::= record { <variabledecl> }1 end
<enumtype>    ::= <name> { , <name> }0 ;
<parmdecpart> ::= parameters { <variabledecl> }1
<variabledecl> ::= <typename> <name> { , <name> }0 ;
<portdecpart> ::= interface { <portdecl> }1
<portdecl>    ::= (actout | conin | conout | datin | datout ) : { <variabledecl> }1

```

LISTING C.1 *Syntax of the element header*

The *arithmetic types* that can be used as typenames are the set of predefined basic types (table C.1) and the defined types. Ports have a direction, i.e., *input* or *output* and a *flow type*, i.e., *activation*, *data* or *control*. The direction and the flow type are combined to the keywords **actout**, **conin**, **conout**, **datin** and **datout**.

## C.4 The primitive process specification

The body of a primitive process specifies the algorithm that is performed by this process to transform the input data to the output data. In principle, assignment statements in any mathematical form are allowed in THESIS, as long as the following rules are obeyed:

- Only the input ports, parameters and local variables can be used in an expression. There are no global variables.
- Local variables must receive a value before they are used in an expression.
- All output ports must be assigned a value exactly once.

Examples of valid descriptions are mathematical equations, block diagrams and SHIDECS statements. In SHIDECS, a process is always described by a sequence of statements.

Name	Description
bool	Boolean value, only the values TRUE and FALSE are allowed.
int	Integer value. It represents the set of whole numbers.
real	The type real represents a continuous continuum of values.
BOOL	Boolean value implementation on a computer. Only the values TRUE and FALSE are allowed.
BYTE	Byte value on a computer. It is represented by an 8 bit number and commonly used for character representation.
INT16 INT32 INT64	Implementations of integer values on a computer. The range of these types is respectively from $-2^{16}$ to $2^{16} - 1$ for the INT16, $-2^{32}$ to $2^{32} - 1$ for the INT32 and $-2^{64}$ to $2^{64} - 1$ for the INT64.
REAL32 REAL64	implementations of real values on a computer. The REAL32 uses the 32 bit IEEE-754 standard format for its representation, the REAL64 uses 64 bit IEEE-754 standard format.

TABLE C.1 *Primitive types supported in THESIS*

## C.5 The control specification

The body of a Cspec specifies the behavior of the Finite State Machine (FSM) controlling the activation of processes. A control specification consists of four parts:

- A state variable containing the current state of the FSM.
- Conditions specifying when the transitions of the FSM may occur.
- Actions that are associated with the transitions (the Mealy part).
- Actions that are associated with the state (the Moore part).

The control specification may be specified as a set of boolean equations and statements, as a set of state-transition tables and action tables, or as a state-transition diagram. In SHIDECS, it is always described as two sequences of statements, i.e., one for the Mealy part and one for the Moore part.

## C.6 The store specification

The body of the store specification specifies its behavior. The contents of the store are defined implicitly by the arithmetic type of the incoming flow and of the outgoing flow. Currently, the following store types are available:

- The *integrator* that is a state-variable in the continuous-time system.
- The *Zero Order Hold* that forms the interface from the discrete-time part to the continuous-time part.
- The *sampler* that specifies the sampling of continuous-time signals in the discrete-time part.
- The *initial token* provider, this store is used when cyclic dependencies of processes occur. It contains a single token at startup, the initial value.
- The *subsampler*, used to get from a higher frequency to a lower frequency.
- The *oversampler*, used to go from a lower sampling frequency to a higher sampling frequency.
- The *overwriting* store. If a new token arrives before the old token is consumed, it will overwrite the existing token.
- The *regenerating* store. As the overwriting store, but it will deliver the token to its output as often as it is asked for by the receiving process, e.g., a variable.
- The *bounded queue*, this is a first-in first-out queue with a fixed number of buffer places.





## Models of the example system

In this appendix, different forms of models will be derived of the example system of chapter 5. Figure D.1 depicts the iconic diagram of the system.

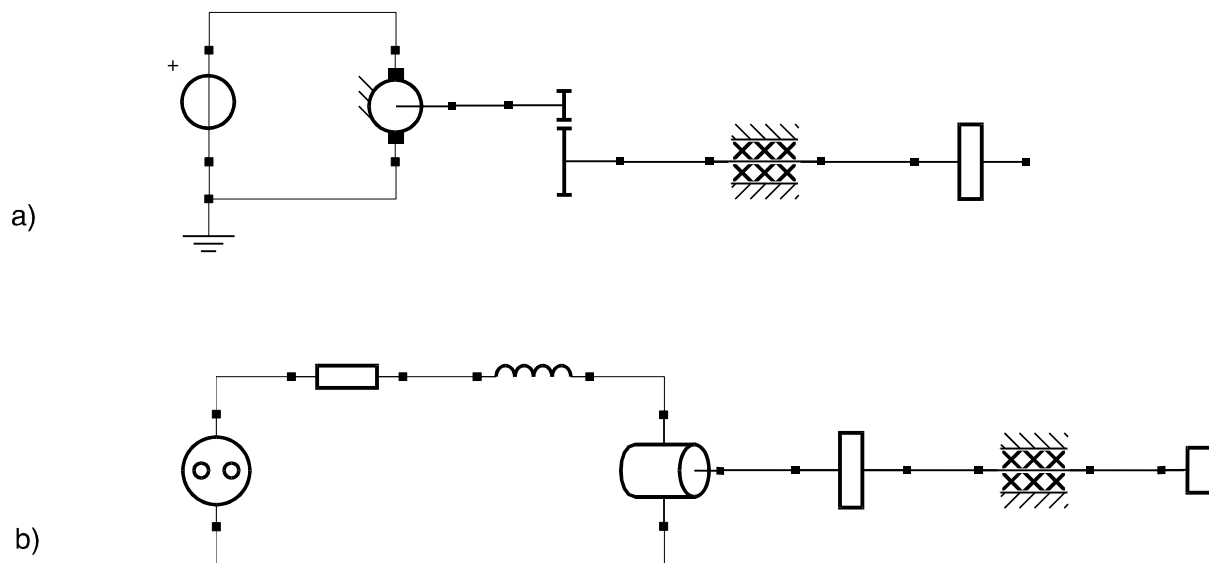


FIGURE D.1 *Iconic diagram of the example system*  
 a) *Top level*  
 b) *Internal model for the dc-motor*

In bond graph terms, this model is as shown in figure D.2. Note that by means of the indicated parameters, the bond graph makes clear that all elements (except the voltage source) are thought to be linear and time-invariant.

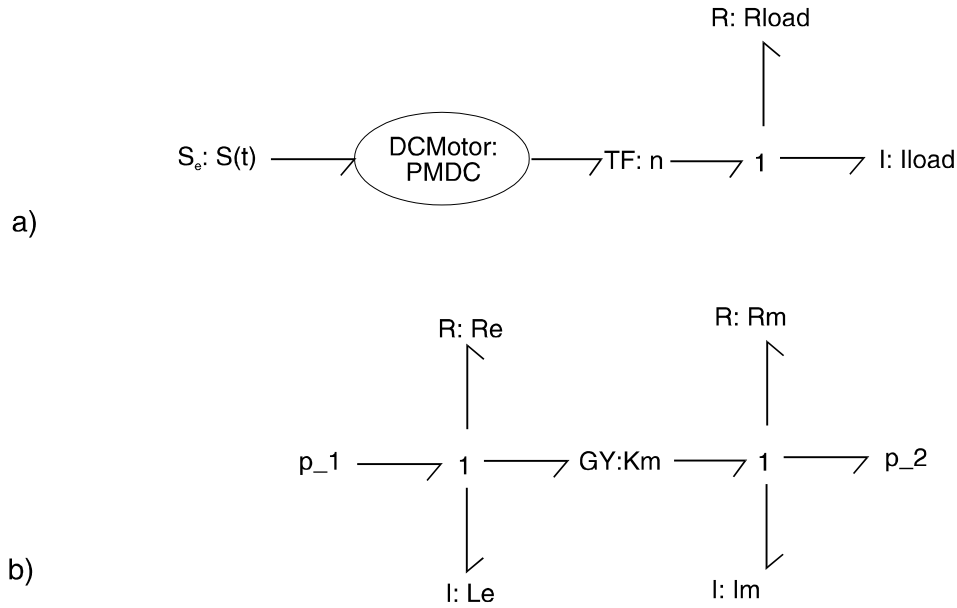


FIGURE D.2 Corresponding bond graph model of the example system  
a) Top level  
b) Internal model for the dc-motor

A state space formulation of this model is as follows:

$$\dot{\mathbf{x}} = \begin{bmatrix} -\left(\frac{R_m}{n^2} + R_{load} + I_m\right) \frac{1}{I_{load}} & \frac{1}{n} \frac{1}{L_e} K_m \\ -\frac{1}{n} \frac{1}{I_{load}} K_m & -\frac{1}{L_e} R_m \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ S(t) \end{bmatrix} \quad (\text{D.1})$$

with  $x_1$  equal to the state of  $I_{load}$  and  $x_2$  to the state of  $L_e$ .

By taking the parameters equal to the values specified in table D.1, the quantitative state space formulation becomes:

$$\dot{\mathbf{x}} = \begin{bmatrix} -0.003 & 2.78 \\ -0.027 & -33 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ S(t) \end{bmatrix} \quad (\text{D.2})$$

parameter	value
n	6
$R_{\text{load}}$	$5 \cdot 10^{-5}$ [Nms]
$I_{\text{load}}$	0.3 [kgm <sup>2</sup> ]
$R_e$	10 [ $\Omega$ ]
$L_e$	$3 \cdot 10^{-3}$ [H]
$K_m$	$50 \cdot 10^{-3}$ [NmA <sup>-1</sup> ]
$R_m$	$1 \cdot 10^{-5}$ [Nms]
$I_m$	$0.8 \cdot 10^{-3}$ [kgm <sup>2</sup> ]

TABLE D.1 *Parameter values*

---

## References

- Aalbers, R.G.D. (1993), Development of models of kinematic mechanisms (using MAX), internal report no 93R084, Control Laboratory, University of Twente, Enschede, Netherlands.
- Akman, V., and P.J.W. ten Hagen (1989), The Power of Physical Representations, *AI Magazine*, Fall, 49–65.
- Aksit, M., and L. Bergmans (1992), Obstacles in Object–Oriented Software Development, Proc. OOPSLA '92 (Vancouver, Canada), 341–358.
- Alberts, L.K. (1993), *YMIR: an Ontology for Engineering Design*, PhD thesis, University of Twente, Enschede, Netherlands.
- Andreasen, M.M. (1980), *Syntesemetoder på systemgrundlag* (Machine design methods based on a systematic approach – contribution to a design theory), in Danish, PhD thesis, Technical University of Lund, Lund, Sweden.
- Asimow, M. (1962), *Introduction to Design*, Prentice–Hall, Englewood Cliffs, N.J., U.S.A.
- Bolks, J. (1993), Mechatronisch ontwerp van een practicumopstelling voor Regeltechniek (Mechatronic design of an experimental setup for control engineering), in Dutch, internal report no 93R079, Control Laboratory, University of Twente, Enschede, Netherlands.
- Booch, G. (1991), *Object Oriented Design with applications*, Benjamin/Cummings, Redwood City, CA, U.S.A.
- Bosch, P.P.J. van den (1989), Linear System Analysis and Design with TRIP, BOZA automatisering B.V., Pijnacker, The Netherlands.
- Bradley, D.A., R.H. Bracewell and R.V. Chaplin (1993), Engineering Design and Mechatronics: The Schemebuilder Project, *Research in Engineering Design* **4**, 241–248.
- Bradley, D.A. and J. Buur (1993), The representation of mechatronic systems, *in* Roozenburg (1993) **1**, 37–44.
- Bradley, D.A., D. Dawson, N.C. Burd and A.J. Loader (1991), *Mechatronics – Electronics in Products and Processes*, Chapman and Hall, London, U.K.

- Breedveld, P.C. (1982), Proposition for an unambiguous vector bond graph notation, *J. Dynamic Systems, Measurement, and Control* **104**, 267–270.
- Breedveld, P.C. (1984), *Physical system theory in terms of bond graphs*, PhD thesis, University of Twente, Enschede, Netherlands.
- Breedveld, P.C. (1986), A systematic method to derive bond graph models, Proc. 2nd European Simulation Congress, G.C. Vansteenkiste, E.J.H. Kerckhoffs, I. Dekker and J.C. Zuidervaart (eds.), Antwerp, Belgium, 38–44.
- Breedveld, P.C., R.C. Rosenberg and T. Zhou (1991), Bibliography of bond graph theory and application, *J. Franklin Institute* **328** (5/6), 1067–1109.
- Breunese, A.P.J. (1992), Design and implementation of a mechatronic modelling environment using object oriented principles, MSc thesis no 92R071, Control Laboratory, University of Twente, Enschede, Netherlands.
- Breunese, A.P.J. (1993a), Preliminary specification of Sidops++, internal report no 93R199, Control Laboratory, University of Twente, Enschede, Netherlands.
- Breunese, A.P.J. (1993b), Design of the MAX Equation Editor, internal report no 93R200, Control Laboratory, University of Twente, Enschede, Netherlands.
- Broenink J.F (1986), SIDOPS, a bond graph based modelling language, Complex and distributed systems: analysis and control, IMACS trans. Scient. computation **4**, Tzafestas S, Borne P. (eds.), North Holland, Amsterdam, Netherlands, 81–86.
- Broenink, J.F. (1990) *Computer–Aided Physical–Systems modeling and simulation: a bond graph approach*, PhD thesis, University of Twente, Enschede, Netherlands.
- Broenink, J.F., J. Bekkink and P.C. Breedveld (1992), Multibond–graph version of the CAMAS modeling and simulation environment, *Bond graphs for engineers*, P.C. Breedveld and G. Dauphin–Tanguy (eds.), Elsevier, Amsterdam, Netherlands, 253–262.
- Buur, J. (1990), *A theoretical approach to mechatronics design*, PhD thesis, Institute for Engineering Design, Technical University of Denmark, Lyngby, Denmark.
- Buur, J., and M.M. Andreasen (1989), Design models in mechatronic product development, *Design Studies* **10**, 19–34.
- Cardelli, L., and P. Wegner. (1985), On understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys* **17** (4), 471–522.
- Clark, K., and T. Fujimoto (1991), *Product Development Performance*, Harvard Business Press, Boston, Mass., U.S.A.
- Cross, N. (1989), *Engineering Design Methods*, J. Wiley & Sons, Chichester, U.K.
- Dahl, O.G., and K. Nygaard (1966), Simula an Algol–based simulation language. *Communications of the ACM* **9** (9), 671–678.
- Dasgupta, S. (1991), *Design Theory and Computer Science*, Cambridge University Press, Cambridge, U.K.

- Darke, J. (1979), The primary generator and the design process, *Design Studies* **1**, 36–44.
- David, B.T. (1987), Multi-Expert Systems for CAD, Proc. 1st Eurographics Workshop (Noordwijkerhout, Netherlands), Springer Verlag, Berlin, Germany, 57–67.
- Dijk, J. van and P.C. Breedveld (1991), Automated mechatronic system modelling, Proc. 13th IMACS World Congress on Computation and Applied Mathematics (Dublin, Ireland), R. Vichnevetsky and J.J.H. Miller (eds), Criterion Press, Dublin, Ireland, 1088–1090.
- Dijk, J. van, T.J.A. de Vries, A.P.J. Breunese and P.C. Breedveld (1992), Automated mechatronic systems modelling using MAX, *Bond graphs for engineers*, Elsevier, Amsterdam, Netherlands, 269–280.
- Dijk, J. van (1994), *On the role of causality in models of mechatronic systems*, PhD thesis, University of Twente, Enschede, Netherlands.
- Dijkstra, E.W. (1976), *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., U.S.A.
- Dorf R.C. (1989), *Modern Control Systems*, 5th edition, Addison-Welsley, Reading, Mass., U.S.A.
- Eades, P. and R. Tamassia (1987), Algorithms for automatic graph drawing: an annotated bibliography, TR 82, Dep. of CS, University of Queensland, Australia.
- Eekels, J. (1973), *Industriële doelontwikkeling, een filosofisch-methodologische analyse* (Industrial new business development, a methodological study), in Dutch, Van Gorcum & Comp., Assen, Netherlands.
- Elmqvist, H. (1979), Dymola – a structured model language for large continuous systems, Proc. Summer Computer Simulation Conference (Toronto, Canada).
- Evers, H.G. (1993), Equation editor for MAX, internal report no 93R082, Control Laboratory, University of Twente, Enschede, Netherlands.
- Falkenhainer, B., and J.L. Stein (eds.) (1992), *Automated Modeling*, Proc. ASME Winter Annual Meeting (Anaheim, CA, U.S.A.), DSC-41, ASME, New York, NY, U.S.A.
- Finger, S., and J.R. Dixon (1989a), A review of research in Mechanical Engineering Design. Part I: descriptive, prescriptive, and computer-based models of design processes, *Research in Engineering Design* **1**, 51–67.
- Finger, S., and J.R. Dixon (1989a), A review of research in Mechanical Engineering Design. Part II: representations, analysis, and design for life cycle, *Research in Engineering Design* **1**, 121–137.
- Finger, S. and J.R. Rinderle (1989), A transformational approach to mechanical design using a bond graph grammar, Proc. ASME conf. on Design Theory and Methodology – DTM '89, DE-17, ASME, New York, U.S.A., 107–116.

- Firestone, F.A. (1933), A new analogy between mechanical and electrical systems, *J. Acous. Soc. Am.* **4**, 249–267.
- French, M.J. (1985), *Conceptual Design for Engineers*, 2nd edition, The Design Council, London, U.K. and Springer Verlag, Berlin, Germany.
- French, M.J. (1993), The Opportunistic Route and the Role of Design Principles, *Research in Engineering Design* **4**, 185–190.
- Goldberg, A. (1989), *Smalltalk–80: The interactive programming environment*, Addison–Wesley, New York, U.S.A.
- Goldberg A. and D. Robson (1989), *Smalltalk–80: The language*, Addison–Wesley, New York, U.S.A.
- Geertsema, P. (1993), MSc thesis no 93R222, Control Laboratory, University of Twente, Enschede, Netherlands.
- Grace, A., A.J. Laub, J.N. Little and C. Thompson (1990), Control system toolbox for use with MATLAB™ User's guide, The MathWorks, Inc., South Natick, Mass., U.S.A.
- Hogan, N. (1987), Modularity and Causality in Physical System Modeling, *J. Dynamic Systems, Measurement, and Control* **109**, 384–391.
- Hogan, N. and E.D. Fasse (1989), Conservation principles and bond–graph junction structures, Proc. ASME Winter Annual Meeting (Chicago, Il., U.S.A.), DSC–**8** (Automated Modeling for Design), R.C. Rosenberg and R. Redfield (eds.) ASME, New York, NY, U.S.A., 9–13.
- Hoover, S.P., J.R. Rinderle and S. Finger (1991), Models and abstractions in design, Proc. Int. Conf. on Engineering Design ICED '91 (Zürich, Switzerland).
- IDA (1988), Report R–338.
- IEEE (1987), *Electrical and electronics graphical symbols and reference designations*, 2nd ed., IEEE, New York, U.S.A., ISBN 471–63456–5.
- Interactive Development Environments (1991), *Software through pictures*, San Fransisco, CA, U.S.A.
- IRDAC, (1986) Opinion on R&D needs in the field of mechatronics, Industry R&D Advisory Comittee of the Comm of the EC, Bruxelles.
- Jain, P. and A.M. Agogino (1990), Theory of design: an optimization perspective, *Mechanism and Machine Theory* **25**, 287–303.
- John, P.A. (1988), The ergonomics of computer aided design within an advanced manufacturing technology, *Applied Ergonomics* **19**, March, 40–48.
- Jones, J.C. (1980), *Design methods – Seeds of human futures*, J. Wiley & Sons, London, U.K.
- Jong, A.J.M. de (1986), *Kennis en het oplossen van vakinhoudelijke problemen* (Knowledge and the solution of specific problems), in Dutch, PhD thesis, Technical University of Eindhoven, Eindhoven, Netherlands.

- Karnopp, D.C. and R.C. Rosenberg (1968). *Analysis and Simulation of Multiport Systems: The Bond Graph Approach to Physical System Dynamics*, MIT Press, Cambridge, Mass., U.S.A.
- Karnopp, D.C. and R.C. Rosenberg (1968), *System Dynamics: A Unified Approach*, J. Wiley & Sons, New York, U.S.A.
- Kleijn, C. (1993), Modelling kinematic mechanisms using iconic diagrams, internal report no 93R2??, Control Laboratory, University of Twente, Enschede, Netherlands.
- Konda, S., I. Monarch, P. Sargent and E. Subrahmaniam (1992), Shared Memory in Design: A Unifying Theme for Research and Practice, *Research in Engineering Design* **4**, 23–42.
- Koster, M.P. (1993), Constructieprincipes (Construction Principles), in Dutch, course material, University of Twente, Enschede, Netherlands.
- Koster, M.P. and W.T.C. van Luenen (1993), Inleiding mechatronica (Introduction to Mechatronics), in Dutch, course material, University of Twente, Enschede, Netherlands.
- Krasner, G.E. and S.T. Pope (1988), A cookbook for using the Model–View–Controller user interface paradigm in Smalltalk–80, *J. Object Oriented Programming*, August/September, 26–49.
- Libardi, E.C., J.R. Dixon and M.K. Simmons (1988), Computer environments for the design of mechanical assemblies: A research review, *Engineering with computers* **3**, 121–136.
- Luenen, W.T.C. van (1993), *Neural networks for control – on knowledge representation and learning*, PhD thesis, University of Twente, Enschede, Netherlands.
- Macfarlane, J.F. and M. Donath (1988), The automated symbolic derivation of state equations for dynamic systems, Proc. IEEE Conf. on Artificial Intelligence, San Diego, CA, U.S.A.
- Macfarlane, J.F. (1989), *Qualitative and symbolic analysis of dynamic physical systems*, PhD thesis, University of Minnesota, Minnesota, U.S.A.
- Malmqvist, J. (1993), *Towards Computational Design Methods for Conceptual and Parametric Design*, PhD thesis, Chalmers University of Technology, Göteborg, Sweden.
- McCall, R.J. (1986), Issue–serve systems: a descriptive theory of design, *Design Methods and Theories* **20** (3), 443–458.
- Meindertsma, P. (1992), Polymorphic modelling within the mechatronic modelling environment MAX II (using object oriented principles), MSc thesis no 92R220, Control Laboratory, University of Twente, Enschede, Netherlands.
- Miller, G.A. (1956), The Magical Number Seven, Plus or Minus Two, *Psychological Review* **63**, 81–97.



- Mitchell, E.E.L. and J.S. Gauthier (1976). Advanced Continuous Simulation Language (ACSL). *Simulation* **26** (5), 49–53.
- Mullins, S. and J.R. Rinderle (1991), Grammatical Approaches to Engineering Design, Part I: An Introduction and Commentary, *Research in Engineering Design* **2**, 121–135.
- Nevenzel, G.S.H. (1992), Hierarchical causality assignment in MAX, MSc thesis no 92R207, Control Laboratory, University of Twente, Enschede, Netherlands.
- Newell, A. and H.A. Simon (1972), *Human Problem Solving*, Prentice–Hall, Englewood Cliffs, N.J., U.S.A.
- NNi (1991), *NEN 5152: Technische tekeningen. Elektrotechnische symbolen* (Technical drawings. Graphical symbols for electro technology), in Dutch, NNi, Delft, Netherlands, ISBN 90–5254–060–8.
- Olmeco Consortium (1991), OLMECO: Open Library for models of MEchatronic COmponents, ESPRIT proposal EC 60521, Brussels, Belgium.
- Paynter, H.M. (1961), *Analysis and design of engineering systems*, MIT–press, Cambridge, Mass., U.S.A.
- Paynter, H.M. (1975), Resistive multiports, Lecture no. 5 of the Berkeley Lecture series.
- Perelson, A.S. (1975a), Bond Graph Sign Conventions, Trans. ASME *Journal of Dynamic Systems, Measurement and Control* **97**, 184–188.
- Perelson, A.S. (1975a), Bond Graph Junction Structures, Trans. ASME *Journal of Dynamic Systems, Measurement and Control* **97**, 189–195.
- Peterson, J.L. (1981), *Petri net theory and modeling of systems*, Prentice–Hall, Englewood Cliffs, N.J., U.S.A.
- Popper, K.R. (1972), *Objective knowledge, an evolutionary approach*, Oxford University Press, Oxford, U.K.
- Prins, J.A.A. and M.M. Olthoff (1993), Morphology as a documentation framework, *in Roozenburg* (1993) **1**, 158–164.
- Pugh, S. (1984), CAD/CAM – hindrance or help to design?, Conf. Conception et fabrication assistées par ordinateur, Bruxelles, Belgium.
- Redfield, R.C. and S. Krishnan (1992), Towards Automated Conceptual Design of Physical Dynamic Systems, *J. Engineering Design* **3** (3), 187–204.
- Rinderle, J.R., E.R. Colburn, S. P. Hoover, J.P. Paz–Soldan and J.D. Watson (1989), Form–Function Characteristics of Electro–Mechanical Designs, Design Theory '88, S.L. Newsome, W.R. Spillers and S. Finger (eds.), Springer Verlag, New York, U.S.A., 132–147.
- Rinderle, J.R. and B.L. Subramaniam (1991) Automated bond graph modeling and simplification to support design, *in Stein* (1991), 45–68.

- Rivero, V. (1977), *Une Contribution a la Conception Architecturale Assistée par Ordinateur* (Contribution to computer supported architectural design), in French, PhD thesis, Université Scientifique et Médicale de Grenoble, Grenoble, France.
- Roozenburg, N. and N. Cross (1991), Models of the Design Process – Integrating Across the Disciplines, Proc. Int. Conf. on Engineering Design ICED '91 (Zürich, Switzerland).
- Roozenburg, N.F.M. (ed.) (1993), *Proceedings of ICED '93 1–3*, 9th International Conference on Engineering Design (The Hague, Netherlands), WDK–22, Heurista, Zürich, Switzerland.
- Rosenberg, R.C. (1987), Exploiting bond graph causality in physical system models, *Trans. ASME J. Dyn. Sys. Meas. Control* **109**, 378–383.
- Rosenberg, R.C., J. Whitesell, and J.D. Reid (1992). Extendible simulation software for dynamic systems. *Simulation* **58** (3), 175–183.
- Rosencode Associates (1990), The ENPORT Reference Manual, Lansing, MI, U.S.A.
- Salomons, O.W., F.J.A.M. van Houten and H.J.J. Kals (1993), Review of Research in Feature–Based Design, *J. Manufacturing Systems* **12** (3), 113–132.
- Sharpe, J.E.E. and R.H. Bracewell (1993), Application of Bond Graph Methodology to Concurrent Conceptual Design of Interdisciplinary Systems, Int. Conf. on Systems, Man and Cybernetics **1** (Le Touquet, France), IEEE, Piscataway, NJ, U.S.A., 7–13.
- Shearer, J.L., A.T. Murphy and H.H. Richardson (1967), *Introduction to System Dynamics*, Addison–Wesley, Reading, Mass., U.S.A.
- Simon, H.A. (1981), *The Sciences of the Artificial*, MIT Press, Cambridge, Mass., U.S.A.
- Simon, H.A. (1973), The structure of ill–structured problems, *Artificial Intelligence* **4**, 181–201.
- Stein, J.L. (ed.) (1991), Automated Modeling 1991. Proc. ASME Winter Annual Meeting (Atlanta, GA, U.S.A.), DSC–34, ASME, New York, NY, U.S.A.
- Stet, G.L.T. (1993), Building Open Look™ applications in SMALLTALK-80, internal report no 93R176, Control Laboratory, University of Twente, Enschede, Netherlands.
- Suh, N.P. (1990), *The principles of design*, Oxford University Press, New York, U.S.A.
- Sun Microsystems, Inc. (1990), *Open Look™ Graphical User Interface – Application Style Guidelines*, Addison–Wesley, Reading, Mass., U.S.A.
- Takeuchi, H., and I. Nonaka (1986), The New Product Development Game, *Harvard Business Review*, 137–146.

- Thoma, J.U. (1975), *Introduction to Bond Graphs and Their Applications*, Pergamon Press, Oxford, U.K.
- Tomiyama, T. and H. Yoshikawa (1987), Extended General Design Theory, Proc. IFIP WG 5.2 Working Conference 1985, Tokyo, Japan, North-Holland, Amsterdam, Netherlands, 95–124.
- Tomiyama, T., T. Kiriya, H. Takeda, D. Xue and H. Yoshikawa (1989), Metamodel: A Key to Intelligent CAD Systems, *Research in Engineering Design* **1**, 19–34.
- Top, J.L. (1993), *Conceptual modelling of physical systems*. PhD thesis, University of Twente, Enschede, Netherlands.
- Ullman, D.G. (1992), *The mechanical design process*, McGraw-Hill, New York, U.S.A.
- Ullman, D.G. (1989), A Taxonomy for Mechanical Design, *Research in Engineering Design* **3**, 179–189.
- Ullman, D.G., T.G. Dietterich and L.A. Stauffer (1988), A model of the mechanical design process based on empirical data, *Artificial Intelligence in Engineering Design and Manufacturing* **2** (1), 33–52.
- Ulrich, K.T. (1988), *Computation and Pre-Parametric Design*, report no AI-TR-1043 (revised PhD thesis), Artificial Intelligence Laboratory, MIT, Cambridge, Mass., U.S.A.
- Ulrich, K.T. and W.P. Seering (1989), Synthesis of Schematic Descriptions in Mechanical Design, *Research in Engineering Design* **1**, 3–18.
- Universal Technical Systems Inc. (1988), *TK Solver manual*.
- Vries, T.J.A. de, J. van Amerongen and P.C. Breedveld (1993), A model the design process for development of CAE systems, in Roozenburg (1993) **3**, 1409–1417.
- Vries, T.J.A. de and P.C. Breedveld (1992), A model of the modelling process, *Bond graphs for engineers*, P.C. Breedveld and G. Dauphin-Tanguy (eds.), Elsevier, Amsterdam, Netherlands, 291–302.
- Vries, T.J.A. de, J. van Dijk, A.P.J. Breunese and P.C. Breedveld (1992), Multiple model representations to support concurrent engineering, Proc. Workshop ‘Concurrent Engineering: Requirements for Knowledge-Based Design Support’, 10th European Conf. on Artificial Intelligence, Vienna, Austria.
- Vries, T.J.A. de, P.C. Breedveld and J. van Amerongen (1991), A design theory for classification and development of CAE systems, in Stein (1991), 37–43.
- Vries, T.J.A. de, P.C. Breedveld and P. Meindertsma (1993), Polymorphic modelling of engineering systems, Proc. Int. Conf on Bond Graph Modeling and Simulation, Western Simulation MultiConference, SCS, San Diego, California, U.S.A., 17–22.

- Wijbrans, K.C.J. (1993), *Twente Hierarchical Embedded Systems Implementation by Simulation*. PhD thesis, University of Twente, Enschede, Netherlands.
- Ward, A.C. (1989), *A theory of quantitative inference for artefact sets, applied to a mechanical design compiler*, PhD thesis, Artificial Intelligence Laboratory, MIT, Cambridge, Mass., U.S.A.
- Welch, R.V. (1992), *Conceptual design of mechanical systems: a representation and computational model*, PhD thesis, University of Massachusetts, Amherst, Mass., U.S.A.
- Welch, R.V. and J.R. Dixon (1991), Conceptual design of mechanical systems, Proc. ASME Design Theory and Methodology, DE-31, 61–68.
- Wijsman, G. (1992), Automatic generation of schematic diagrams, MSc thesis no 92R221, Control Laboratory, University of Twente, Enschede, Netherlands.
- Wirth, N. (1982), *Programming in Modula-2*, Springer-Verlag, Berlin, Germany.
- Yourdon, E.N. (1989), *Modern Structured Analysis*, Prentice-Hall, Englewood Cliffs, N.J., U.S.A.