Constraint-Aided Conceptual Design

A thesis submitted to the National University of Ireland
in accordance with the requirements of the degree of
Doctor of Philosophy in the Faculty of Science.

Barry Alan O'Sullivan
Department of Computer Science
University College Cork

July 1999

Contents

List of Figures

List of Tables

Abstract

The product development process is concerned, not only with the design of products, but with how these products are manufactured, distributed and serviced. Engineering conceptual design can be defined as that phase of the product development process during which the designer takes a specification for a product to be designed and generates many broad solutions to it. Each of these broad solutions is, generally, referred to as a *scheme*. Each scheme should be sufficiently detailed that the means of performing each function in the design has been fixed, as have any critical spatial and structural relationships between the principal components. While it is generally accepted that conceptual design is one of the most critical phases of the product development process, few computer tools exist which provide support to designers working in this stage of product development.

The thesis defended in this dissertation is that constraint processing offers a rich basis for supporting the human designer during engineering conceptual design. A new perspective on the conceptual phase of engineering design is presented, upon which a constraint-based approach to supporting the human designer is developed. This approach is based upon an expressive and general technique for modelling: the design knowledge which a designer can exploit during a design project; the life-cycle environment which the final product faces; the design specification which defines the set of requirements that the product must satisfy; and the structure of the various schemes that are developed by the designer. A computational reasoning environment based on the notion of constraint filtering is proposed as the basis of an interactive design support tool to assist a human designer working in the conceptual phase of design. Using this interactive design support tool, the designer can be assisted in developing models of proposed schemes which satisfy the various constraints that are imposed on the design.

The primary contribution of this research is that it provides a novel approach to supporting the human designer during the conceptual phase of engineering design. The approach presented here not only addresses the issue of modelling and reasoning about the design of products from an abstract set of requirements, but it also demonstrates how life-cycle knowledge can be incorporated into the conceptual design of a product and how alternative schemes can be compared.

Declaration

This dissertation is submitted to University College Cork, in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Science. The research and thesis presented in this dissertation are entirely my own work and have not been submitted to any other university or higher education institution, or for any other academic award in this university. Where use has been made of other people's work, it has been fully acknowledged and referenced.

Barry Alan O'Sullivan
July 1999.

Dedication

This dissertation is dedicated to my mother, for always being there.

Acknowledgements

I would like to extend my warmest gratitude to my PhD supervisor, Professor James Bowen, for his help and encouragement while I carried out the research presented in this dissertation. In particular, I would like to thank him for the time he spent helping me to present my research and thesis in the form of this dissertation.

I would like to thank my partner in love and life, Linda, for always being there for me while I completed this research. Her love has been a constant source of inspiration to me.

I would like to thank Frank Boehme and Chris Dornan for the all the advice on LaTeX. Also, I would like to thank my fellow members of the Constraint Processing Group, Alex Ferguson, Peter MacHale, Marc van Dongen and Steve Prestwich, for the many interesting discussions on constraint processing, but in particular, for just being good friends.

I would like to thank Maurice O'Brien, Pat Doyle and Mel Conway, of Bourns Electronics Ireland, for their assistance with the industrial case-study that was carried out at their company.

I also wish to acknowledge the support of the European Commission and their funding of the "Concurrent Engineering Design Adviser System" (CEDAS) project (Esprit Project 20501) and the Integration in Manufacturing and Beyond (IiMB) Subgroup for Design Coordination (Esprit Working Group 21108). My involvement in these activities was a very valuable source of inspiration for the research that I present here.

Finally, I would like to extend my deepest thanks to my family: my mother, Joan; my father, Noel; my brother, Brian; my sister, Carmel; and my brother-in-law to be, Anthony. They are a constant reminder of what is truly important in life.

Chapter 1

Introduction

This chapter briefly introduces the research presented in this dissertation. The influence of conceptual design on the remainder of the product development process and on the final product is presented. The research objective and scope of the project is defined. A formal statement of the thesis is presented. The major contributions of this work are outlined in general terms. The chapter ends with a description of the architecture of the dissertation and a guide to reading it.

1.1 Engineering Design and Product Development

Engineering design[1] is concerned with the development of detailed specifications for products which provide a technical function. It is a demanding process which requires expertise in many different fields such as science, engineering and, often, art. Product development is concerned not only with the design of products, but with how these products are manufactured, assembled, distributed and so on. A typical product life-cycle is illustrated in Figure 1.1. In this figure it can be seen that design can be considered to be a phase of the life-cycle of a product.

Over the past few years, a good deal of research has been carried out in the fields of design and product development. This is due to the recognition that product development has become the battleground upon which industry can achieve competitive advantage [104]. Improving the product development process increases efficiencies in operational areas within an organisation, such as manufacturing and assembly. The benefits include shorter development and manufacturing lead-times, increased prod-

[1] In the remainder of this dissertation the term *design* will be used as a shorthand for the phrase *engineering design*.

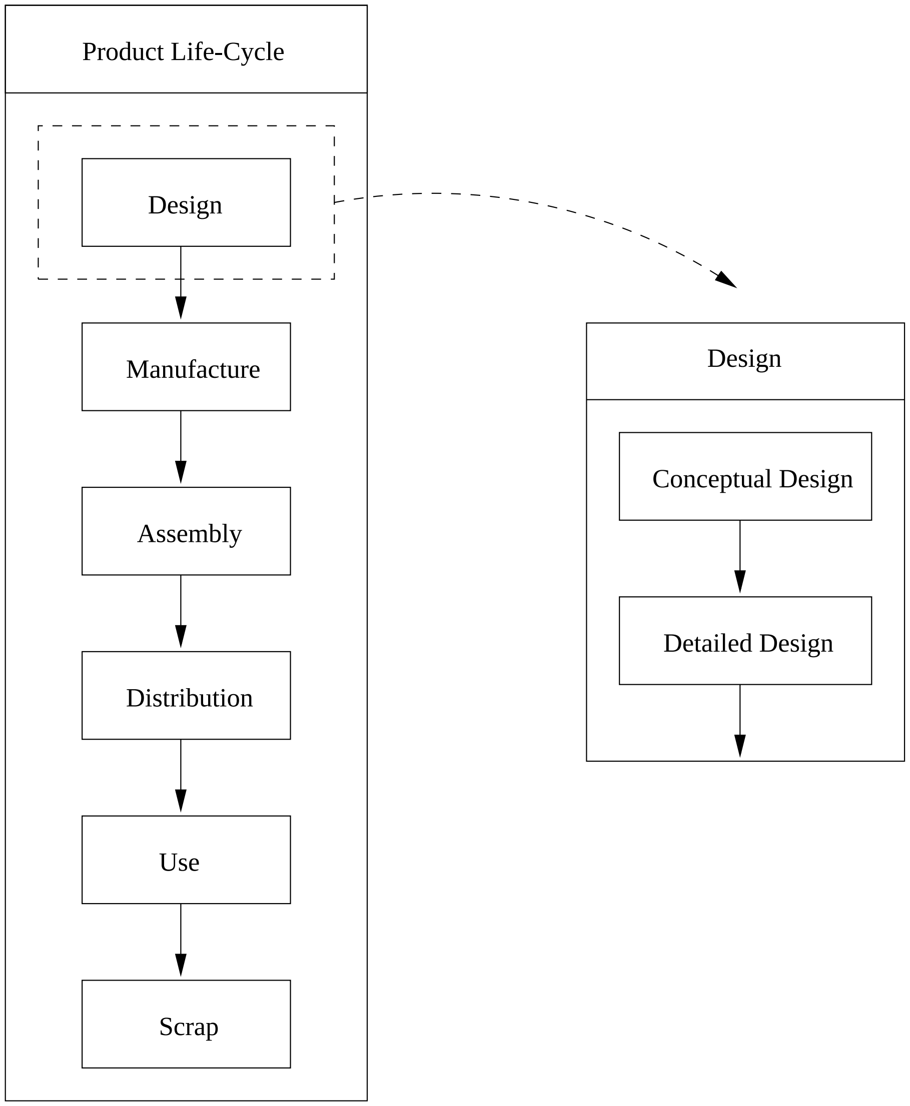


Figure 1.1: Engineering design and the product life-cycle

uct life-cycle revenue, reduced engineering change-order costs and less development waste.

In an effort to increase the effectiveness and productivity of design, several approaches to incorporating knowledge of the downstream phases of the product life-cycle have been developed. The most commonly used of these approaches is *Design for X (DFX)* [17, 85]. The "X" in DFX can be interpreted in a number of ways [150]. For example, "X" could be interpreted as either a *life-cycle process*, such as manufacturing or assembly, or a *design property*, such as quality or cost; it can also be interpreted as meaning a *set* of these processes or design parameters. The objective of DFX is the optimisation of the design of a product from the perspective of the life-cycle process(es) or design parameter(s) referred to by "X". DFX techniques have been successfully applied to the design of products in industry [17]. The improvement in the competitiveness of many companies has been due to the use of DFX during product development.

Due to the level of sophistication and customisation expected by customers in today's markets, the amount of knowledge required for bringing a product to market has become vast. Consequently, modern approaches to product development advocate integration amongst the various phases of the product development process. This model of product development has given rise to several approaches to supporting multidisciplinary decision making and design, such as Integrated Product Development [5], Concurrent Engineering [39] and Design Coordination [51].

1.2 The Topic of this Research

The thesis presented in this dissertation is concerned with the development of a constraint-based approach to supporting engineering conceptual design. Engineering conceptual design can be regarded as that phase of the design process in which the designer takes a specification for a product to be designed and generates many broad solutions to it. Each of these broad solutions is generally referred to as a *scheme* [65]. Each scheme should be sufficiently detailed that the means of performing each function in the design has been fixed, as have any critical spatial and structural properties of, and relationships between, the principal components.

It is generally accepted that conceptual design is one of the most critical phases of the product development process. It has been reported that up to 80% of a product's total cost is dictated by decisions made during the conceptual phase of design [101]. Furthermore, poor conceptual design can never be compensated for by good detailed design [82].

The cost of a product is a complex function of the elements from which it is configured and the various processes required to design, manufacture, assemble and service it. During design, the human designer has an ideal opportunity to tailor the design of a product to the particular life-cycle processes which face the products that she[2] develops. Therefore, major cost advantages can be gained through ensuring that, during conceptual design, the decisions which are made are informed by data relating

[2]Throughout this thesis the feminine pronoun *she* is used to refer to the human designer. This is a neater approach than continually writing *he or she*, or *(s)he*.

to the product being designed and the life-cycle processes required to produce and deliver the final product.

Although the many benefits of effective conceptual design have been recognised, very few computational tools exist which provide support to the human designer working in this stage of the design process. The majority of those tools which do exist are to be found mostly in research laboratories in universities around the world. Additionally, many of the tools which are available do not *assist* the designer in the task of design but *dominate* the entire design process. However, a designer may be inhibited by the design policies of her company or may be required to carry out design tasks in a particular order due to the design tools being used. While the design policies of the company should be supported, indeed enforced, by good design tools, the designer should be allowed to perform design tasks in any order she wishes. It is the primary objective of this research to address this need for interactive designer support during conceptual design. A more precise statement of the goals of this research are presented in Section 1.3.

1.3 The Goals of this Research

In the discussion of engineering design and product development presented so far, the importance of conceptual design has been highlighted. The benefits of providing support during the conceptual phase of design are well-known. Thus, there exists a real justification for developing approaches to supporting the designer during this phase of design.

The Galileo constraint programming language [22, 49, 116] has been widely used to develop design adviser systems for supporting integrated approaches to design, such as Concurrent Engineering [22] and post-design verification [49, 116]. It has been shown that Galileo offers an appropriately rich basis for developing design adviser systems for these aspects of the *detailed* phase of design. It is believed that Galileo can also provide an appropriately rich basis for supporting *conceptual* design.

Therefore, the objectives of the work presented in this dissertation are as follows:

1. To investigate whether constraint-based reasoning can be used as a basis for supporting conceptual design;

2. To determine if the expressiveness needs of conceptual design motivate the

introduction of new features into Galileo.

These goals have been used to form a central thesis which will be defended in this dissertation. This thesis is presented in Section 1.4.

1.4 A Statement of the Thesis

To achieve the goals presented in the previous section, the approach adopted in this research is to attempt to use constraint processing to provide a basis for the modelling and evaluation of a set of alternative schemes for a product. In this approach, constraint processing is used to ensure that all design concepts are consistent with respect to the various restrictions imposed by the design specification and by the product life-cycle.

Thus, the thesis which is presented and defended in this dissertation can be stated as follows:

> *"It is possible to develop a computational model of, and an interactive designer support environment for, the engineering conceptual design process. Using a constraint-based approach to supporting conceptual design, the important facets of this phase of design can be modelled and supported."*

1.5 Contributions of this Research

The research presented in this dissertation contributes to the state of knowledge in the fields of constraint processing and computer-aided engineering design. The research addressed many important issues in product development, such as computer support for a primarily abstract and creative phase of design, constraint processing in early-stage engineering design, computer-assisted evaluation of alternative schemes and the use of DFX concepts during conceptual design. The research provides a unique insight into how constraint processing techniques can be used to support conceptual design. The concept of the function-means tree [28] was extended and used as a vehicle for interactively developing alternative constraint-based models of a product having particular properties while providing a particular functionality. The use of Pareto optimality within the constraints paradigm to actively prune uninteresting parts of the design space during design is also novel. From a design perspective, this research represents a novel approach to interactively supporting designers during conceptual design. The interactive nature of the approach advocated here ensures that the utility of the designer's expertise, knowledge and creativity is never compromised.

1.6 Structure of this Dissertation

This dissertation comprises six chapters and four appendices. The remainder is structured as follows:

Chapter 2 reviews the background literature relevant to the thesis presented in this dissertation.

Chapter 3 presents the theory of conceptual design upon which the thesis presented in this dissertation is based.

Chapter 4 describes a constraint-based implementation of the conceptual design theory presented in Chapter 3.

Chapter 5 demonstrates the approach to supporting conceptual design that is presented in this dissertation, on two design problems: a toy design problem, related to the design of a transportation vehicle, and an industrial case-study.

Chapter 6 presents a number of conclusions and recommendations for further study.

Appendix A presents an overview of the Galileo language and its run-time environment.

Appendix B presents the Galileo implementation of the various generic design concepts which are used as a basis for supporting every conceptual design project.

Appendix C presents all the Galileo code used in the toy design problem related to the design of a transportation vehicle.

Appendix D presents all the Galileo code used in the industrial case-study carried out in association with Bourns Electronics Ireland.

Chapter 2

Background

This chapter reviews the background literature relevant to the thesis presented in this dissertation. The literature on conceptual design research is reviewed from three perspectives: firstly, with a focus on the approaches to modelling design problems, design knowledge and design solutions; then, with a focus on the various design reasoning techniques which have been advocated; thirdly, a number of trends in conceptual design research are identified and discussed. Constraint processing techniques have been applied to many aspects of the engineering design problem. This chapter also reviews the literature on the application of constraint processing techniques to engineering design. It will be shown that, while constraint processing has been applied to many aspects of design, there is considerable scope for research into using constraint processing techniques to support the conceptual phase of design. Finally, this chapter reviews the concept of Pareto optimality. The principle of Pareto optimality is used in this research as a basis for assisting the human designer compare alternative schemes that satisfy a design specification for a product.

2.1 A Review of Conceptual Design Research

Conceptual design has been defined as that phase of design which takes a statement of a design problem and generates broad solutions to it in the form of, what are generally referred to as, "schemes" [65]. A scheme should be sufficiently detailed that the means of performing each major function has been fixed, as have any spatial and structural relationships of the principal components.

While conceptual design is regarded as the most demanding phase of design on the designer [101], it also offers the greatest scope for improvements in the design of the product [65]. It is the phase of design where engineering science, practical knowledge, knowledge of production methods and commercial expertise must be brought together, and where the most important design decisions are made. It is widely acknowledged that up to 80% of a product's total cost is dictated by decisions made during the conceptual phase of design [101]. Furthermore, the effects of poor conceptual design can never be rectified by good detailed design [82]. Most errors in design, as opposed to those made during production, are due to the use of a flawed conceptual design [124].

In Chapter 1 the role of conceptual design in the product development process was discussed. While in many theories of design there is no explicit mention of a conceptual phase of design, all theories recognise the need to synthesise preliminary solutions to the design problem. It is these early synthesis activities that can be regarded as conceptual design.

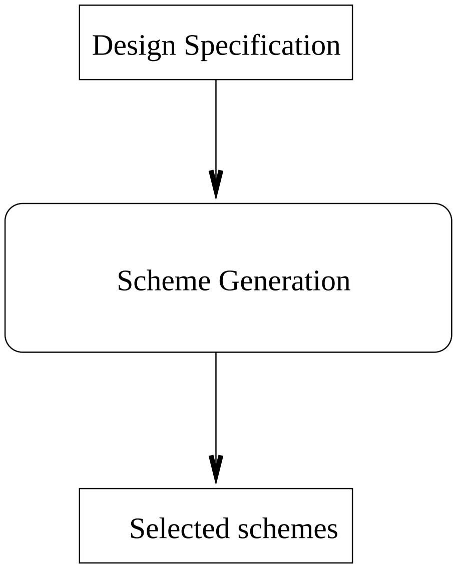


Figure 2.1: A simple perspective on the conceptual design process.

A simple perspective on the conceptual design process is illustrated in Figure 2.1. A design specification can be regarded as a set of requirements that the product must satisfy. The output of the conceptual design process is a set of schemes for products that have the potential to satisfy the requirements described in the design specification. This set of schemes has been selected from a larger set of alternatives which have been considered during conceptual design. These schemes will be further developed during later stages of product design. A more detailed view of the conceptual design process is presented in Figure 2.2.

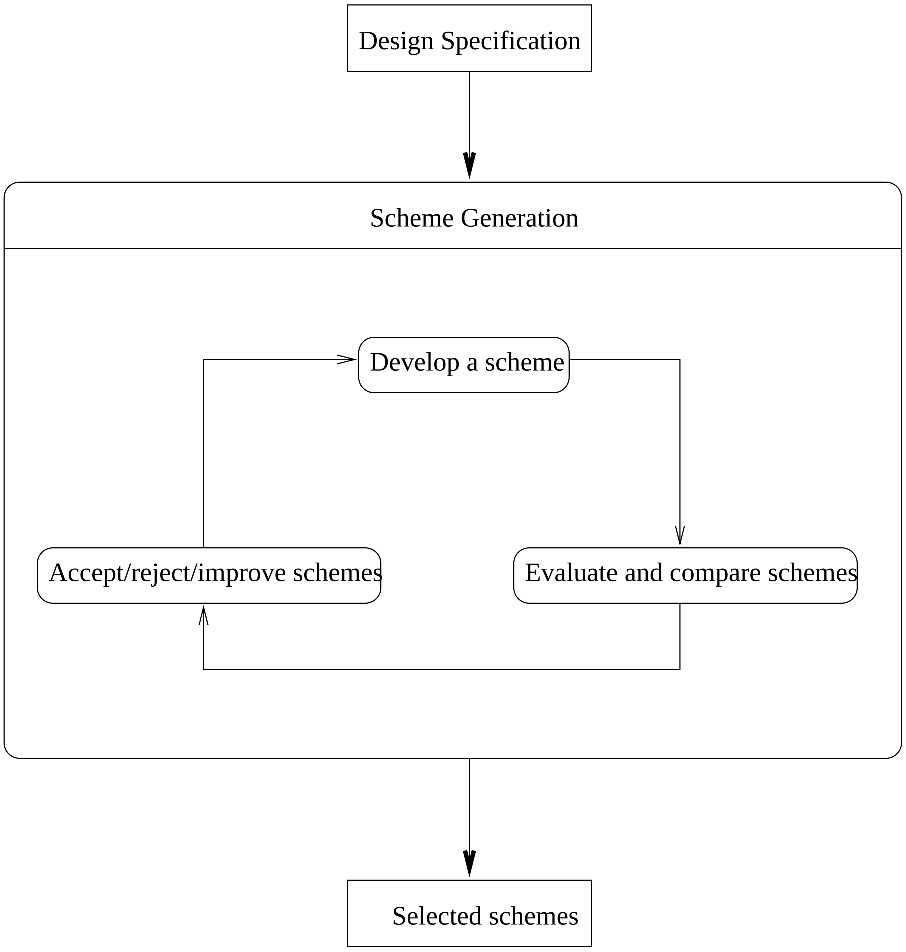


Figure 2.2: A more detailed view of the conceptual design process.

From Figure 2.2 it can be seen that conceptual design is an iterative process during which the designer generates a set of alternative schemes for a product based on a design specification. The design specification will contain a set of requirements that the product to be designed must satisfy. Based on these requirements the designer begins an iterative process of scheme generation. During this process the designer will develop a scheme that is intended to satisfy the design specification. The designer will then evaluate the scheme against the design specification and compare it to those schemes which have already been developed. As new schemes are generated, the designer may decide to accept, improve or reject from the existing pool of schemes. One of the primary objectives at this stage of design is to avoid prematurely selecting one scheme without considering as many other schemes as possible. This model of conceptual design is consistent with all of the “text-book” descriptions of the process that exist in the literature [86, 118, 153, 156].

2.1.1 Modelling for Conceptual Design

In the design theory literature it is recognised that modelling, both the product design and the design knowledge from which the design is developed, is one of the most difficult issues to address [29, 82]. A number of approaches have been proposed that attempt to address this issue. These approaches can be classified as:

- Function-based representations;
- Domain representations;
- Design grammars;
- Geometrical representations;
- Ontologies and Object-oriented techniques;
- Knowledge-based approaches.

These will now be reviewed.

Function-based Modelling

Since designs exist to satisfy some purpose or function [35], knowledge of functionality is essential in a wide variety of design-related activities, such as the specification, generation, modification, evaluation, selection, explanation and diagnosis of designs. Generally, function can be regarded as the abstraction of the intended behaviour of a product. A number of researchers have integrated the role of function into complete theories of design [4, 86, 99].

Formalising the representation of functional design requirements is essential for supporting conceptual design using a computer. Two approaches to representing function have been reported in the literature [155]. These approaches involve representing functions as either verb-noun pairs [106] or as input-output transformations where inputs and outputs can be energy, materials or information [118]. The former approach is often referred to as the *symbolic function* model while the latter is often referred to as the *I/O function* model [100].

It has been reported that while the I/O function model is more systematic than the symbolic function model, the symbolic function model is the more flexible of the two approaches [100]. The symbolic function model has been widely used [64, 106, 115, 127, 145, 154, 157]. However, the I/O based approach has also been widely advocated [36, 25, 121]. One of the major advantages of the symbolic function model is that designers can define a function at quite a high level of abstraction. Furthermore, by describing the intended functionality in this way the designer is not biased towards developing a product with a particular physical structure.

The term "synthesis" is used in design to refer to the process of developing a physical realisation for an abstract notion of a product. For example, developing a configuration of parts from a functional specification of a product can be regarded as an instance of synthesis. Many approaches to design synthesis based on function have been reported in the literature [28, 33, 36, 100, 127, 157].

Function-means modelling is based on the notion of a *function-means tree* [28]. A function-means tree describes alternative ways of providing a top-level (root) function through the use of means. A means is a known approach to providing functionality. Two types of means can be identified in a function-means tree: principles and entities. A principle is defined as a collection of functions which, collectively, provide a particular functionality; it carries no other information than the lower-level functions to be used in order to provide a higher-level function. An entity represents a part or sub-assembly. The function-means approach has been adopted by several researchers as a basis for supporting synthesis [28, 99]. Indeed, from the earliest stages of the development of the thesis presented in this dissertation a similar approach has been adopted [111, 115].

Another approach to design synthesis based on function is known as Function-Behaviour-State modelling [157]. Function-Behaviour-State modelling has been proposed as an approach to minimising the subjectivity of function in design [157]. This approach is based on a distinction between two types of relationships: *Function-Behaviour* relationships and *Behaviour-State* relationships. Function-Behaviour relationships are used to relate functions to behaviours. For example, the function "to make a sound" can be provided by behaviours such as "something striking a bell". In order to support synthesis, Behaviour-State relationships are used to describe all possible behaviours of an entity.

Some researchers have adopted a "Function-Behaviour-Structure" (FBS) approach to supporting synthesis based on function [127]. This approach uses the relationships between the physical structure, behaviour and functionality of existing designs to provide a basis upon which designs for new products can be developed using analogical reasoning. Integrating new production technologies has been addressed using a generalisation of the FBS approach called "Function-Behaviour-Process-Structure" modelling [33].

The interpretation of function as a transformation between inputs and outputs is the basis for systems such as the DICAD-Entwurf system [100] and the FuncSION system [36, 37]. In these, a functional model of a product is developed from a set of IO function units. These units can be configured in a particular way to develop a complete functional description of a product based on the transformation of flows through the design. Amongst the advantages of the IO approach is that it is a formal approach to conceptual design. However, the disadvantages are that it is not very flexible and is often limited to particular design domains such as the design of mechanical mechanisms.

Domain Representations

Many European design researchers regard the design of mechanical systems as a progression through a two dimensional space [4]. This space is illustrated in Figure 2.3. One dimension of this space relates to the designer's understanding of a solution to a design problem - defined as a progression from an abstract to a concrete understanding. The second dimension relates to the specification of the design solution - defined as a progression from a simple to a detailed specification of the solution.

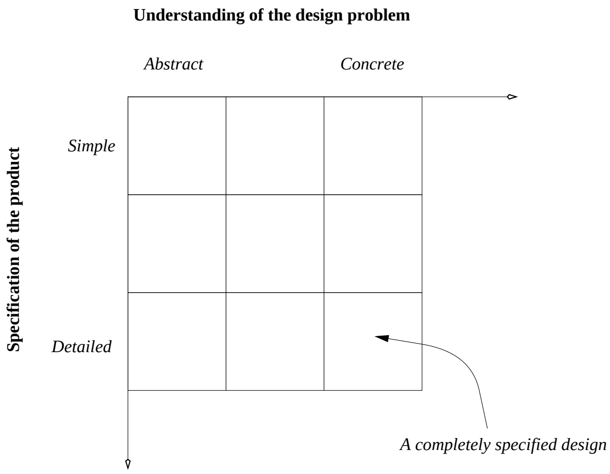


Figure 2.3: The progression of a designer's ideas through the design process.

When a designer is developing a design for a product there are several perspectives that she can have on the emerging design. For example, the "Theory of Domains" defines four perspectives, called domains, of a mechanical system design [4]. These four domains are the *process domain* which describes the transformation that takes place in the product, the *function domain* which describes the functions to be provided in the product and the relationships between them, the *organ domain* which describes the interfaces between the entities that provide the functions and the *part domain* which describes the parts and structure of the product.

Another approach to supporting design synthesis based on the maintenance of consistency between a number of views of a product model is the "Artifact Model" approach [132]. This approach is based on the premise that a sufficient level of support can be offered to a designer through the use of three product viewpoints: a function view, a solution view and a state view [132]. These views collectively provide a complete model of the state of the product being designed and the design activities that were performed by the designer.

Grammars and Ontologies

Grammatical approaches to design effectively define a design language [138]. There are two main categories of design grammar: *shape grammars* [143] and *graph grammars* [61]. Grammars have been developed as a mechanism for specifying a set of designs in terms of the transformations that can be used to generate that set [143]. For example, shape grammars are often used in CAD systems to ensure the validity of the geometric model of the product [61, 79].

The use of design grammars during conceptual design has been widely reported in the literature [3, 82]. Designs which have been generated from shape grammars may often need to be modified before they are considered in further detail. The refinement of designs generated from shape grammars has been reported in the literature [30, 138]. Some grammar-based approaches to conceptual design attempt to ensure the validity of designs in terms of the design specification, which may change during design, and in terms of the constraints inherent in the life-cycle of the product. For example, the issue of ensuring that the designs that are generated using grammars are manufacturable has been addressed [26]. In addition, predicting the variety of possible designs that can be generated using a given grammar has also been addressed [95].

Instead of developing ad-hoc design languages, many researchers have directed their efforts at developing a common design ontology. An ontology is a set of common terms and concepts that are general enough to describe different types of knowledge in different domains but specific enough for application to particular design problems [82]. The YMIR ontology has been proposed as a domain-independent ontology for the formal representation of engineering design knowledge [2]. Design ontologies have been used as the basis for many general-purpose approaches to addressing some critical aspects of product development, such as the evaluation of designs using design norms [44].

Geometry-based Methods

Most modern CAD systems are geometry-based. In other words, when using a conventional CAD system, the designer focuses on the dimensions and spatial relationships between design elements. However, during conceptual design the various alternatives that are created and compared by a designer are generally created from a non-spatial perspective and lack detailed geometric structure [9]. That is not to say that consideration of geometry is irrelevant at the conceptual design stage. In order to satisfactorily evaluate an engineering design concept, it is often important to consider all critical geometric and spatial relationships that are relevant.

The "minimum commitment principle" is well-known in engineering design. This principle states that no decision should be made beyond what is necessary to ensure the quality of the current solution [6, 53, 74]. This principle has been used as an approach to creating geometric models of design solutions at the conceptual stage of design in which only the most critical details of the product are modelled [74]. This ensures that the designer has maximum scope later in the design process to make decisions about the specifics of a particular design.

The use of features as carriers of function in design has been an important area of research for a number of years [55, 96, 126]. A feature is considered to be "a region of interest in a part model" which provides some function [159]. Since features provide function and are generally described geometrically, feature modelling for conceptual design is a specific application for geometry-based modelling. Many sophisticated approaches for supporting feature modelling for conceptual design in both two and three dimensions have been reported [55]. Representing features for use in collaborative product design environments has also be reported in the literature

[96]. In these approaches features are represented by different views - each view relating to a particular discipline represented in the product development team.

Some researchers have combined parametric geometry, features and variational modelling into integrated design representation schemes for conceptual design [117].

Knowledge-based Methods

During conceptual design a designer exploits a considerable variety of knowledge to effectively generate a set of good design concepts. A designer will typically exploit information relating to function, technologies for providing function, cost information and life-cycle knowledge. To represent the variety in the knowledge required by designers, flexible modelling paradigms are required. Knowledge-based methods have been developed by a number of researchers which are capable of modelling many aspects of the design problem that are relevant to the conceptual stage of design.

While there exist domain specific knowledge-bases for domains such as vehicle design [32, 40] and the preliminary design of tall buildings [135], more general approaches have also been reported. For example, a knowledge-base known as the "Design Model", which provides a basis for enhancing the design process by providing the designer with a means for communicating a broad range of design knowledge derived from the multiple disciplines involved in the design process, has been reported [94]. Two of the more well-known knowledge-based systems for conceptual design are the QPAS system [91] and the Scheme-Builder system [25, 125].

The use of databases of known design solutions has become very common in conceptual design [1, 48, 89, 90, 160]. These databases have the flexibility to store vast amounts of data relating to many different aspects of the design domain of interest to the designer.

Many other knowledge-based design approaches have been reported in the literature [14, 34, 37, 44, 47, 93, 108, 140, 146]. Amongst the approaches taken are the use of model-based representations, analogical knowledge-bases and concept libraries for modelling complex collections of design knowledge. These systems address different aspects of the knowledge requirements of conceptual design and general engineering design.

2.1.2 Reasoning Techniques for Conceptual Design

Based on the modelling techniques presented in Section 2.1.1, many approaches to supporting reasoning during conceptual design have been developed. Design reasoning techniques form the basis for computer-based design support systems. Computer-based design tools can provide the human designer with various levels of support, from fully automated design generators through to post-design verification environments [151].

A review of the literature on reasoning techniques for conceptual design is presented here. The review is structured according to the predominant reasoning mechanisms for conceptual design which have been reported in the literature; the following categories will be used here:

- Adaptive Methods;
- Case-based Reasoning;
- Other Knowledge-Based Approaches.

Adaptive Methods

Adaptive Search is a collective term which describes those search and optimisation techniques with operational characteristics that are heuristic in nature [119]. The term "adaptive search" relates to search techniques, such as genetic algorithms, evolutionary search, simulated annealing and hill-climbing. In general, adaptive search techniques have been used as a basis for automated design applications. The goal of these automated design applications has been improving the features of existing designs through the optimisation of variables defined within the system [31]. Of the many adaptive search methods that exist, Genetic Algorithms [72] have been most widely used to support reasoning during conceptual design; thus, the emphasis in this section will reflect that.

Genetic Algorithms (GAs) have been used to generate candidate solutions to particular types of design problems. The design of optical prisms and other solid objects has been reported in the literature [11, 12]. The use of GAs in general engineering is a considerably more difficult problem due to the possibility of the existence of constraints on what constitutes a valid design. The use of problem-specific knowledge to both enhance the search power and ensure that only valid designs are generated has been reported [88, 119]. The use of GAs in automatically generating design solutions is often used in conjunction with other techniques such as backtracking, constraint propagation and heuristic-based methods [18].

A number of other adaptive methods have been used to support reasoning in conceptual design. These include neural networks and machine learning [31, 50, 71, 75, 97].

Case-based Reasoning

Case-based Reasoning (CBR) is a relatively new approach to decision automation [89]. CBR is based on the premise that humans generally solve new problems by modifying solutions to previously solved problems. The solutions to previously solved problems are stored in a case-base. When a new problem is being addressed, the case-base is searched for solutions to similar problems that can be used as a basis for a solution to the current project. Failed searches form the basis for a new case. It is in this way that the case-base can be extended over time.

Many successful applications of CBR have been reported in the literature. DOMINIC is a multi-domain parametric design system in which general design goals are explicitly represented in term of design parameters [80, 45]. KRITIK is a case-based design system for the conceptual design of physical devices [14].

A number of CBR systems have been developed for the conceptual design of structures. Amongst these are the analogical reasoning system STRUPLE [163], and the architectural design systems CADSYN [103] and ARCHIE [120]. CADRE is an architectural design system that can support reasoning about topologies and spatial relationships [83].

The development of a distributed conceptual design system based on using CBR over the world-wide web has been reported [89, 90, 160]. This work is of considerable interest since an approach to supporting design through a combination of Internet, multimedia and CBR technologies is a very promising approach to supporting distributed concurrent engineering.

Other Knowledge-based Approaches

A number of knowledge-based reasoning systems for supporting conceptual design have been reported in the literature. These systems have been developed to support conceptual design in a number of specific engineering domains such as the preliminary design of buildings, creative elementary mechanism design and early-stage design of machine systems.

A knowledge-based expert system called TALLEX has been developed for supporting the preliminary design of buildings [135]. This system is based on the integration of symbolic and numerical processing. The knowledge-base for this system is based on a set of IF/THEN rules that provide the inference mechanism with the information needed to support the designer during the design process.

The creative design of elementary mechanisms has been addressed from a knowledge-based system perspective [146]. Explanation-based learning has also been used to support the acquisition of conceptual design knowledge through an analysis of the structural features of designed objects [93].

The HIDER methodology enables detailed simulation models to be used during the early stages of design [131]. HIDER uses a machine learning approach to develop abstractions from these detailed models. These abstractions are used as a basis for formulating a multiple objective optimisation problem which is used to generate a set of "optimal" conceptual designs for a product. The designer is supported in interactively exploring the design space by managing trade-offs between design objectives.

Constraint-based approaches to supporting engineering design have been reported in the literature. However, these will be reviewed separately in Section 2.2.

2.1.3 Trends in Conceptual Design Research

Over the past number of years industry has needed to become more effective in bringing new products to the market. There is an increasing demand from customers for customised products. In order to achieve manufacturing and field-maintenance economies in this sort of competitive environment, modern manufacturers must strive for minimum lead-time, a minimum of dedicated machine tools and considerable flexibility to react to changes in market demand for their products. In response to these market characteristics, manufacturers have invested heavily in maximising the degree of flexibility of their facilities while minimising lot sizes, minimising manufacturing and delivery lead-times, and maximising the quality and variety of products available to their customers. In terms of product design, the effect has been that manufacturers must have extremely efficient product development processes. Concurrent Engineering and Integrated Product Development have been applied in industry although they are macro-approaches to addressing the problem, the detail of whose implementation is largely unreported.

In the field of engineering design, and in particular conceptual design, there are a number of issues which have emerged as areas which can offer significant competitive advantage to companies. The following have attracted the most attention from industry and academia as critical issues to be addressed:

- Design reuse;
- The design of product families;
- Configurability and modularity in product design.

Each of these issues will be discussed in a little more detail here.

Design Reuse

One of the most difficult aspects of conceptual design is that it can be regarded as a problem solving process aimed at addressing an abstractly defined problem. Conceptual design is often performed in a free-form manner which can result in a designer developing a solution that comprises many novel elements. This can have a considerable effect on the manufacturabily of a product and can cause many difficulties in managing the re-design of the product, if this becomes necessary at a later date. The reuse of design knowledge in new design projects is an attempt at minimising the non-standard content in the designs for new products.

Many companies have tried to increase the degree of commonality between different products by re-using libraries of preferred parts and technologies [54]. It is now becoming commonplace for companies to attempt to structure their design knowledge in a form which can be readily used by designers who develop new designs to new design problems. The DEKLARE project (ESPRIT project 6522) developed a methodology for generating specialised computer-based re-design support systems by capturing the design knowledge used in a company [63]. This approach has also proved useful in supporting the design of new products since designers can use known technological approaches in their product designs.

A number of methodologies have been put in place to assist industry in building databases of design knowledge. The "PS Methodology" has been proposed as a practical approach to rationalising past designs for their effective reuse in future design projects [52].

Modularity and Configurability

Many industrial approaches to increasing the efficiency of product design focus on two issues: modularity and configurability. Modularity in product design relates to the use of modules in structuring a product. Configurability in product design relates to the composition of parts or modules to satisfy some set of design requirements. These two issues have been receiving considerable interest from industry and academia since they offer a methodology for managing the complexity of product design.

A module is a physical element of a technical system which has a clear and explicitly defined interface, is totally self-contained, provides particular known functionalities and exhibits a well-understood behaviour. A product can be regarded as modular if it is composed of a set of modules. Adopting a modular approach to product design can yield significant benefits for the design process, such as reduced design lead-time and limited product complexity [122]. Using a modular approach to product design facilitates an "engineer-to-order" approach to manufacturing since specific customer requirements can be realised by modifying and composing a set of predefined modules.

A number of surveys reporting the experiences of industrial companies who exploit modularity in product design have been carried out [58, 141]. Due to the importance of modularity in product design a new technique for supporting product structuring called "Modular Function Deployment" has been proposed [57]. This technique provides a basis for supporting the design of modular products from the gathering of customer requirements, through conceptual design, to the final design of a fully modular product.

Product configuration management has become a crucial product development success factor in a number of industries. Configuration design can be regarded as the activity of combining a set of predefined components and their relationships in a certain manner so that the resulting design satisfies a set of design requirements

[42, 107]. It has been reported that configuration is a critical factor in a number of industries [66]. A number of extensive reviews of the field of product configuration design exist in the literature [134]. An open question in the field of configuration relates to the problem of re-configuration [128, 144]. The need for approaches which can cope with the re-configuration of an existing configured design is becoming more critical.

Product Family Design

Modern manufacturing companies are facing an ever increasing demand for customised products to be manufactured and delivered within ever shortening lead-times. In many industries the design of families of products has been an attempt at managing market demands while maintaining competitiveness [54]. Competitiveness in many industries requires efficient design and delivery of large numbers of product variants [152]. Product variants can be regarded as members of a product family. During the research presented in this dissertation some work in the area of product family design has been carried out, although it is not presented in this dissertation. In that research a product family was regarded as a collection of products which are similar from some perspective [114].

The effective design of product families requires that designers can quickly develop product designs which are based on a common well-defined set of design elements. The use of modular design techniques is obviously relevant, as indeed is the use of configuration. The key to successful product family design is the use of a small set of well-understood modules whose interfaces are well-defined [56].

2.2 A Review of Constraint Processing for Design

Most decisions that are made in daily life involve considering some form of restriction on the choices that are available. For example, the destination to which someone travels has a direct impact on their choice of transport and route: some destinations may only be accessible by air, while others can be reached using any mode of transport. Formulating decision problems in terms of parameters and the restrictions that exist between them is an intuitive approach to stating these types of problems. These general restrictions can be referred to as “constraints”.

The fact that constraints are ubiquitous in many decision problems has given rise to the emergence of many popular problem-solving paradigms based on the notion of constraints. These techniques have been widely reported in the literature in such research fields as Operations Research (OR) and Artificial Intelligence (AI).

Some of the most popular approaches to solving problems comprising a set of constraints defined on a set of parameters stem from the constraint processing paradigm. Constraint processing is concerned with the development of techniques for solving the *Constraint Satisfaction Problem* (CSP) [102]. A large number of problems in AI, computer science, engineering and business can be formulated as CSPs. For example, many problems related to machine vision, scheduling, temporal reasoning, graph theory, design, design of experiments and financial portfolio management can be naturally modelled as CSPs.

One of the major advantages of the constraint processing approach to solving decision problems is that all that is required is an appropriate formulation of the CSP. There is no need to specify an approach to solving it since constraint processing techniques can be readily used to solve a problem formulated as a CSP.

2.2.1 Constraint Processing for the Phases of Design

In Chapter 1 the process of product design was discussed. In Figure 1.1 it was shown that this process could be divided into two phases: conceptual design and detailed design. Constraint processing techniques have been applied to a variety of aspects of the product design process in recent years. In the following, the literature reporting constraint-based research for phases of the product design process will be presented.

Constraint Processing and Conceptual Design

Research related to constraint-based approaches to supporting conceptual design has been on the increase. However, most of this research does not address the synthesis problem; the vast majority has focused on constraint propagation and consistency management relating to more numerical design decisions.

One of the earliest works in the field of constraint management for conceptual design was carried out in the MIT [137]. The research resulted in the development of a computer tool called "Concept Modeller". This system is based on a set of graph processing algorithms that use bipartite matching and strong component identification for solving systems of equations. The Concept Modeller system allows the designer to construct models of a product using iconic abstractions of machine elements. However, a number of issues are not addressed by this work. Amongst these issues is the dynamic nature of conceptual design. During conceptual design constraints may be added or deleted at any point. In addition, the system does not address the issue of design synthesis, nor does it address the comparison of alternative solutions to a design problem. However, Concept Modeller demonstrated that constraint processing did offer a useful basis for supporting designers working through particular aspects of the conceptual design problem.

Based on the earlier work on Concept Modeller, a system called "Design Sheet" has been developed [27, 130]. This system is essentially an environment for facilitating flexible trade-off studies during conceptual design. It integrates constraint management techniques, symbolic mathematics and robust equation solving capabilities with a flexible environment for developing models and specifying tradeoff studies. The Design Sheet system permits a designer to build a model of a design by entering a set of algebraic constraints. The designer can then use Design Sheet to change the set of independent variables in the algebraic model and perform trade-off studies, optimisation and sensitivity analysis.

Some researchers have used the Dynamic Constraint Satisfaction Problem as a basis for managing conflict during the preliminary phases of engineering design [69, 76]. Traditional conflict resolution techniques in constraint-based models of the design process use *backtracking* and *constraint relaxation*. Some researchers focus on differentiating between types of assumptions that are made by designers during design. Variations on this type of approach have also been proposed for managing conflict in collaborative design [7].

The use of autonomous agents to solve CSPs for conceptual design has been reported in the literature [73]. The motivation for the work is the support of spatial layout generation. The constraint specification used in the work facilitates a high-level representation and manipulation of qualitative geometric information. The search engine used in the proposed system is based on a genetic algorithm. The issue of constraint consistency is not addressed in the work. In addition, important design issues such as synthesis are not considered. However, it is realised that the primary focus of this work is the use of autonomous agents to solve CSPs.

The use of constraint logic programming for supporting reasoning about dynamic physical systems has been reported [60]. This work combines a constraint logic programming approach with bond graphs to assist in the development of a simulation model of a system in the form of a set of differential algebraic equations. The approach can be used for identifying causal problems of a bond graph model of a dynamic physical system.

Constraint Processing and Detailed Design

The later phases of design are concerned with developing a subset of the schemes generated during the conceptual design phase into fully detailed designs. In the design literature two later phases of design are generally identified: embodiment design and detailed design [118]. The embodiment phase of design is traditionally regarded as the phase during which an initial physical design is developed. This initial physical design requires the determination of component arrangements, initial forms and other part characteristics [150]. The detailed phase of design is traditionally regarded as the phase during which the final physical design is developed. This final physical design requires the specification of every detail of the product in the form of engineering drawings and production plans [150]. In reality the boundaries between the various phases of design are quite fuzzy. Therefore, the constraint processing literature for supporting the later phases of design will be reviewed as a whole. However, the constraint processing literature relating to configuration will be reviewed separately in Section 2.2.2. This is to reflect the fact that there exists a clearly identifiable body of literature on constraint processing and configuration design.

The CADET system has been developed at the Cambridge University Engineering Design Centre (EDC) as a computer tool for supporting embodiment design [148]. CADET is an acronym for **C**omputer-**A**ided **E**mbodiment **D**esign **T**ool. CADET is capable of assisting the designer to formulate and satisfy large sets of algebraic constraints. CADET comprises a generic database of components which can be used to develop a constraint-based model of the geometry of the product being designed. The CADET system uses simulated annealing as the basis for its constraint satisfaction algorithm for solving the constraint-based representation of the geometric product model [149]. Recent research at the Cambridge EDC has focused on the development of tools that can assist a constraint-driven design process based on the CADET system [161].

A constraint-based knowledge compiler for parametric design in the mechanical engineering domain called MECHANICOT has been proposed [109]. This system is based on the assumption that the design process can be modelled as a CSP. The purpose of the tool is to generate knowledge-based systems for parametric design of mechanical products by using knowledge compilation techniques. The MECHANICOT knowledge compiler is useful for supporting the reuse of design knowledge and can be used for producing design plans.

Many constraint-based systems reported in the literature have been developed for supporting reasoning about purely geometric aspects of design for use with CAD systems [13, 67, 68, 139]. However, these systems have been developed to address aspects of the design process which are too specific to geometric CAD to be reviewed in depth here.

2.2.2 Constraint Processing for Configuration

The use of constraint processing techniques for supporting configuration design has been widely reported in the literature. Configuration can be regarded as a special case of engineering design. The key feature of configuration is that the product being designed is assembled from a fixed set of predefined components that can only be connected in predefined ways [133]. The core of the configuration task is to select and arrange a collection of parts in order to satisfy a particular specification. The growing interest in configuration systems is reflected by the level of interest reported from industry [59]. The role of constraint-based configurators has been reported in a number of reviews [134].

The configuration problem can be naturally represented as a CSP. In general a configuration problem can be formulated as a CSP by regarding the design elements as variables, the sets of predefined components as domains for each of the design elements and the relationships that must exist between the design elements as constraints. Constraints can also be used to state the compatibility of particular arrangements of components and connections.

One of the earliest works in the field of constraint-based support for configuration was based on dynamic constraint satisfaction [107]. The key characteristic of Dynamic Constraint Satisfaction Problems is that not all variables have to be assigned a value to solve the problem. Depending on the value of particular variables, other variables and constraints may be introduced into the network. Inspired by this approach, the use of constraint processing for configuration problems in complex technical domains has been reported [62, 77].

A general constraint-based model of configuration tasks represented as a new class of non-standard constraint satisfaction problem, called the Composite CSP, has been proposed [133]. The Composite CSP unifies several CSP extensions to provide a more comprehensive and efficient basis for formulating and solving configuration problems. In the Composite CSP approach, variables in the problem can represent complete sub-problems in their own right. This approach provides a useful basis for supporting abstraction in configuration whereby the product can be viewed, recursively, as a larger component aggregated from a set of parts.

A number of variations on the standard configuration problem have been highlighted by industry with a number of constraint-based solutions being proposed in the literature. Amongst these are: reactive and interactive configuration [105]; reconfiguration [144]; and the management of configuration complexity through abstraction [158].

2.2.3 Constraint Processing for Integrated Design

Design is a complex activity which requires a variety of knowledge to enable a product to be successfully designed and developed. The phase model of design is often considered to imply that design is carried out as a sequential process. However, this is not generally how design is performed in industry. Modern approaches to product development, such as Concurrent Engineering [16], Integrated Product Development [5] and Design Coordination [51], attempt to maximise the degree to which design activities are performed in parallel. One of the most common techniques used during the design phase of product development is to use knowledge of the life-cycle of the product being designed to assist designers make well-informed decisions at as early a stage in product development as possible. This is generally known as Design For X (DFX) [150].

A number of researchers in the constraint processing community have developed constraint-based technologies that support integrated approaches to product development [22, 116, 151]. These technologies can be used to support collaborative design by facilitating the use of DFX knowledge during the design process.

Constraint-based approaches to supporting Concurrent Engineering have been very successful, although these systems generally focus on the design of a product in the detailed stages of design. The inter-dependencies between design and manufacturing have formed the basis for the development of design adviser systems which can evaluate the manufacturability of a design and generate re-design suggestions to alleviate any problems that may exist. The "Manufacturing Evaluation Agent" is a computer-assisted Concurrent Engineering technology which identifies cost-critical tolerances in the design and generates cost-reducing design suggestions [78]. The purpose of the system is to help focus the designer's attention on the specific aspects of the design that influence manufacturing cost.

A more general approach to supporting Concurrent Engineering using constraints has been adopted in developing the Galileo constraint programming language [22, 49, 116]. The Galileo language is based on a generalisation of the Predicate Calculus. A program in Galileo is a declarative specification of a constraint network. Constraints may be atomic, compound or quantified sentences in the Predicate Calculus. An interactive run-time environment has been developed for the Galileo language [22]. Using this system a Galileo program becomes fully interactive. At run-time users, who can be regarded as designers, can declare additional parameters and constraints. Alternatively, users may ask questions about the consequences of the existing constraints. A constraint network in Galileo can be divided into various, possibly overlapping, regions which correspond to the perspectives of the various members of a product development team. Each perspective can be presented in a variety of interface styles, including spreadsheets and simple feature-based CAD [19].

The use of Pareto optimality in managing conflict in collaborative design systems has been reported [7]. The same principle has also been reported as part of the research presented in this dissertation [112]. Indeed, there is also a recognised need for assisting the design team co-ordinate the design activities that must be carried out. The use of constraint processing for assisting in the co-ordination of design activities has also been reported [151].

Constraint-based approaches to supporting the use of DFX knowledge during design have been widely reported in the literature. Amongst these are techniques for supporting the evaluation of design norms during the design process [44]. A design norm can be regarded as a statement of good design practice stated in the form of a quality standard or a code of good design practice. The constraint logic programming language CLP($\Re$) has been used to support the Design For Test of active analog filters [110]. Models of active circuits are developed in CLP($\Re$) and these models are then used to support fault diagnosis of the designed circuit. The constraint programming language Galileo has been used to develop design adviser modules for use with electronics CAD systems which critique a design against a set of user specified life-cycle guidelines [49, 116].

2.2.4 Constraint Processing and Design Domains

Constraint processing techniques have been applied to the design of a wide variety of design domains, such as mechanical system design, electronics design and structural design.

In the field of mechanical design, constraint processing techniques have been applied to the validation of features [46] and the general design of mechanical parts [142]. The automated generation of constraint-based design expert systems for mechanical design has also been reported [109].

In the field of electronics design, constraint programming techniques have been widely used to support the verification of both expected and observed functionality of electronics systems. For example, constraint programming has been used to support model based diagnosis of analog circuits [15]. Many modern electronics CAD systems are marketed as having constraint-based editors built-in to allow designers to specify customised relationships that must be maintained during design. Amongst these CAD systems are the electronics CAD systems from such vendors as Zuken-Redac [129], Mentor Graphics [41] and Cadence Systems [147].

Structural and architectural design have also been reported as application domains for constraint processing. The design of bridges has been used as an application domain for constraint processing research into issues such as dynamic constraint satisfaction [84] and conflict management during preliminary phases of design [76]. Constraint-based approaches to floor-planning have also been reported [10]. The layout planning problem has been studied in depth by researchers in the constraint processing community [81].

2.3 Pareto Optimality

A design specification defines the various requirements that a product must satisfy. Many of these requirements are categorical in the sense that they must be satisfied by every scheme that the designer considers. Sometimes, however, a design requirement will merely specify some preference about some aspect of the design; for example, a requirement may state that the mass of the product should be as small as possible. These preferences play a critical role in the evaluation of schemes. There may be many preferences defined in the design specification, related to many aspects of the product and its life-cycle. These design preferences can be regarded as defining a constrained multiple objective optimisation problem.

Pareto optimality [123], which takes its name from the economist Vilfredo Pareto, is an economics technique for identifying solutions to a multiple objective optimisation problem; the principle has been in use since 1906. The principle of Pareto optimality can be used to assist the designer in making decisions about the details of a design which will result in a Pareto optimal concept being developed [113, 115].

In contrast to a single objective optimisation problem which produces a single optimum (or a set of equivalent optima), multiple objective optimisation produces a set of non-dominated (Pareto optimal) solutions [131]. A set of non-dominated solutions is characterised by the property that every solution in the set is either better than every other solution to the problem, with respect to at least one of the objectives, or is at least as good as every other solution on all objectives.

Interest in the principle of Pareto optimality has resulted in the development of a number of techniques for solving multiple objective optimisation problems. Amongst such techniques are "The Method of Weighted Convex Combinations" [162], goal programming [136], multilevel programming [38] and Normal-Boundary Intersection [43]. Recently, a number of researchers have begun to apply the principle of Pareto optimality to a wide variety of problems in design [31, 70, 112, 115, 123, 131]. Indeed, in this dissertation, the principle of Pareto optimality will be used to assist in the evaluation and comparison of alternative schemes that are intended to satisfy a design specification.

The principle of Pareto optimality is illustrated in Figure 2.4 (adapted from [131]). In this figure the the points X, Y and Z represent the *Pareto optimal set* of solutions to a multiple objective optimisation problem involving two conflicting

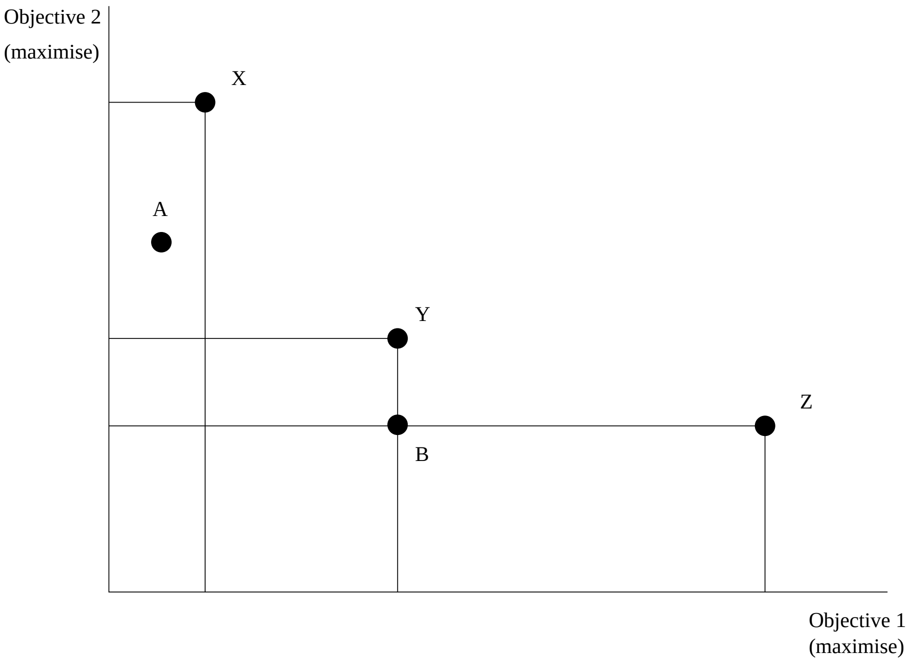


Figure 2.4: The Pareto optimal set of solutions to a multiple objective optimisation problem.

objective functions, both of which are to be maximised. The point X is not dominated by any other solution since there is no solution which is better than it in terms of *Objective 2*. The point Y is not dominated by any other solution since there is no solution which is is better than it in terms of both *Objective 1* and *Objective 2*, simultaneously. The point Z is not dominated by any other solution since there is no solution which is is better than it in terms of *Objective 1*. Two dominated solutions are shown in Figure 2.4; these are points A and B. Point A is dominated by point X because point X performs better than point A on both *Objective 1* and *Objective 2*. Point B is dominated by both points Y and Z; it is dominated by Y because Y performs better than B on *Objective 2* while it does just as well on *Objective 1*; point B is dominated by Z because Z performs better than B on *Objective 1* while it does just as well on *Objective 2*.

Formally, given the set, S, of candidate solutions to a multi-objective optimisation problem the Pareto optimal set, P_s, can be defined as

$$P_s = \{s_i | s_i \in S, not\_dominated(s_i, S)\}.$$

The predicate $not\_dominated(s_i, S)$ means that a candidate solution, s_i, is not dominated by any other candidate solution in S. Thus,

$$not\_dominated(s_i, S) \Leftrightarrow \forall s_j \in S, \neg dominates(s_j, s_i).$$

A solution, s_1, *dominates* another solution, s_2, if s_1 *improves_on* s_2 with respect to any objective and s_2 does not improve on s_1 with respect to any objective. Thus,

$$dominates(s_1, s_2) \Leftrightarrow improves\_on(s_1, s_2) \wedge \neg improves\_on(s_2, s_1).$$

The predicate $improves\_on(s_1, s_2)$ can be defined as follows,

$$improves\_on(s_1, s_2) \Leftrightarrow \exists \langle F, I \rangle \in O, better\_than(F(s_1), F(s_2), I).$$

. where O is the set of objective functions, each objective function in O being a pair, $\langle F, I \rangle$, where F is a function and $I \in \{minimal, maximal\}$; the predicate *better_than* is defined as

$$better\_than(x, y, maximal) \Leftrightarrow x > y$$

$$better\_than(x, y, minimal) \Leftrightarrow x < y.$$

Later in this dissertation it will be shown how the *dominates* relation is used to compare schemes that are being developed by the designer. A scheme which is dominated by another can be regarded as being Pareto sub-optimal and must be either improved or discarded.

2.4 Summary

This chapter reviewed the relevant background literature for the thesis presented in this dissertation. The literature on conceptual design research was reviewed from three perspectives. The literature was firstly reviewed with a focus on the approaches to modelling design problems, design knowledge and the design solutions. The literature was then reviewed with a focus on the various design reasoning techniques which have been advocated. Thirdly, a number of trends in conceptual design research were identified and discussed.

This chapter also reviewed the literature relating to the application of constraint processing techniques to engineering design. It can be seen from the literature review presented in this chapter that there have been a wide variety of approaches adopted by the constraint processing community in addressing the problem of support for the human designer. However, it should be noted that much of this research addresses either well-structured aspects of the design problem or more parametric phases of design. There is a particular lack of constraint processing research in the area of designer support during conceptual design which addresses the critical aspects of the process such as design synthesis and the evaluation and comparison of alternative schemes for a product. As noted in the previous chapter, successful conceptual design adds significantly to the potential for overall successful product design. It is this need which is a primary motivation for the research presented in this dissertation.

Finally, this chapter reviewed the principle of Pareto optimality; this will be used in this research as a basis for assisting the human designer compare alternative schemes that satisfy a design specification for a product.

Chapter 3

Theory

Conceptual design is an extremely demanding phase of the design process. Designers must often combine imagination and technical expertise to satisfactorily solve a design problem; thus, in order to assist a designer during conceptual design, a computer needs be to capable of supporting both the technical and non-technical aspects of the process. Conceptual design can be regarded as a process which translates a specification for a product into a set of broad product concepts, known as "schemes". During conceptual design, the human designer develops a set of alternative schemes for the product by applying design knowledge which is known to her. In this chapter a theory of conceptual design is presented. This theory was developed as a result of conversations with design researchers and a review of the design research literature. The theory was then used as the basis for the computer-based implementation presented later in this dissertation.

3.1 A Perspective on Conceptual Design

Engineering conceptual design can be regarded as that phase of the design process during which the designer takes a specification for a product to be designed and generates many broad solutions to it. These solutions are generally referred to as "schemes" [65]. Each scheme should be sufficiently detailed that the means of performing each function in the design has been fixed, as have any critical spatial and structural relationships between the principal components [65]. The task of developing even a single scheme for a design specification requires the application of expertise from a wide variety of technical and non-technical disciplines. Also, designers are often required to use imagination and creative flair in order to develop satisfactory schemes.

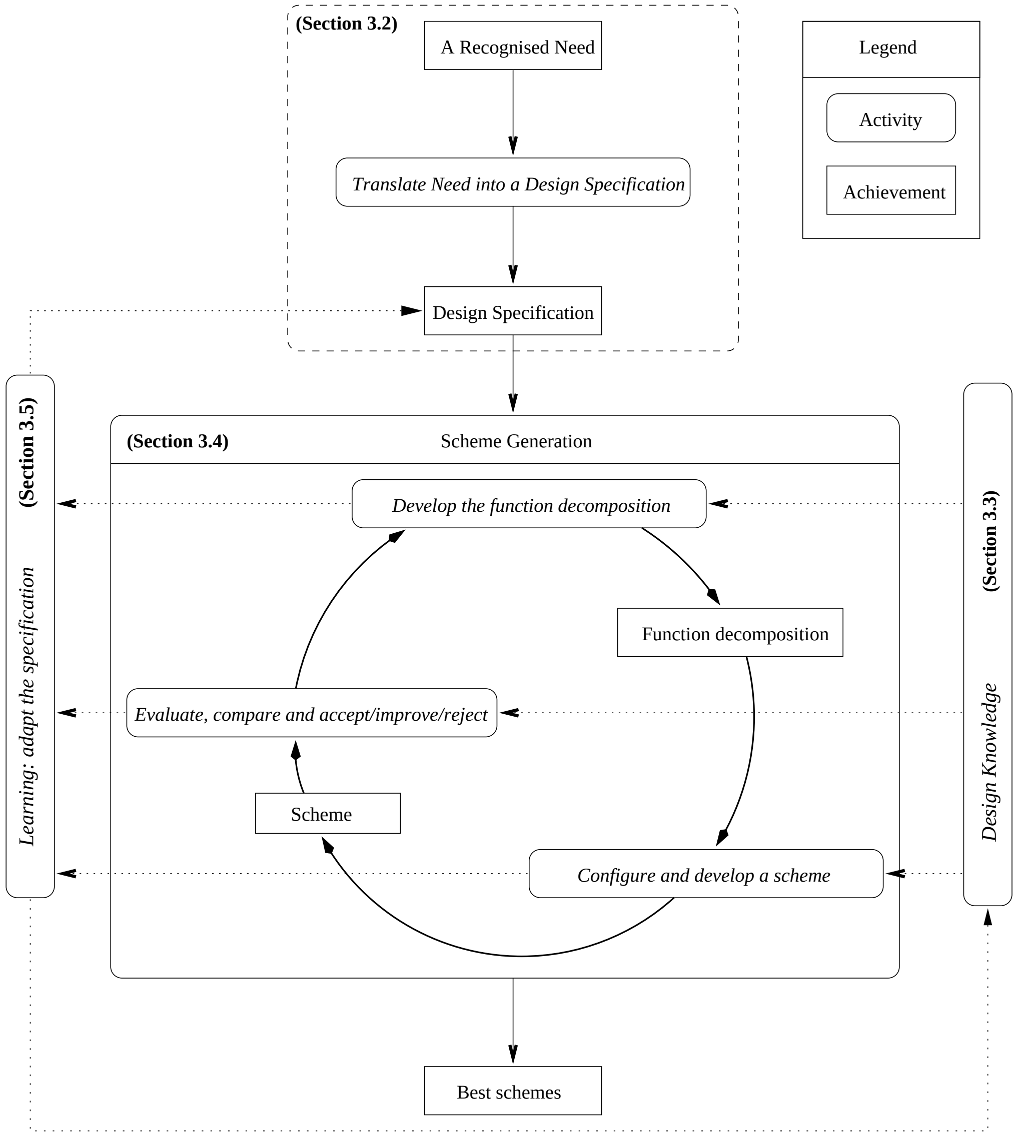


Figure 3.1: The model of the conceptual design process adopted in this research.

The model of conceptual design adopted in this research is based on the fact that during the conceptual design process a designer works from an informal statement, comprising an abstract functional requirement and a set of physical design requirements, and generates alternative configurations of parts which satisfy these requirements. Central to this exercise is an understanding of function and how it can be provided. Figure 3.1 illustrates this model of conceptual design graphically. While this model is based on a well-known approach to conceptual design [118], the detail is novel. In particular, the approach to developing schemes is based on an extension of the function-means tree; this extension, called the function-means map, has been developed as part of the research presented in this dissertation [113]. Various parts of the model presented in Figure 3.1 are annotated with section references. Each section reference points to the section of this chapter which offers a detailed discussion of that part of the conceptual design process which has been marked. One section reference pertains to a number of aspects of the model of the conceptual process illustrated in the figure. The scope of this section reference is denoted by a dashed box.

From Figure 3.1 it can be seen that the conceptual design process can be regarded as a series of activities and achievements. The achievements and activities relate to the development of the design specification and an iterative process of scheme generation. The process of scheme generation involves the development of a function decomposition which provides the basis for a configuration of parts that form a scheme. This scheme is then evaluated and compared against any schemes that have already been developed. Based on this comparison, the designer will choose to accept, improve or reject particular schemes. The process of scheme generation will be repeated many times in order to ensure that a sufficiently large number of schemes have been considered. The role of design knowledge and learning are also illustrated in Figure 3.1, using dotted lines. Design knowledge is used during the process of scheme generation. During scheme generation the designer may develop a greater understanding of the design problem being addressed; this learning may affect the design specification for the product or the design knowledge used to generate schemes.

In the remainder of this chapter a detailed discussion of the theory of conceptual design used in this research will be presented with reference to Figure 3.1.

3.2 The Design Specification

From Figure 3.1 it can be seen that the conceptual design process is initiated by the recognition of a need or customer requirement. This need is analysed and translated into a statement which defines the function that the product should provide (referred to as a functional requirement) and the physical requirements that the product must satisfy. This statement is known as a *design specification.* A number of techniques exist for generating a design specification from a perceived need. The most well known of these techniques is Quality Function Deployment (QFD) [92]. The formulation of the various requirements which comprise the design specification can be regarded as the first major achievement of the conceptual design process.

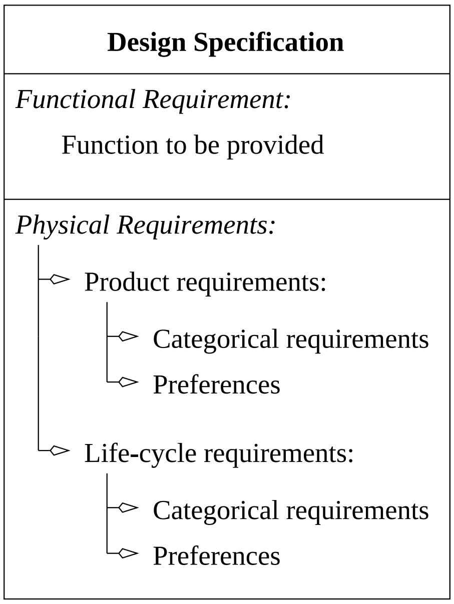


Figure 3.2: The contents of a design specification.

The contents of a design specification are illustrated in Figure 3.2. The design specification contains a set of requirements that the product must satisfy. Two categories of design requirement can be identified: *functional* requirements and *physical* requirements. A design specification will always contain a single functional requirement; it may also contain a set of physical requirements. The nature of each of these requirements will be discussed in further detail in the following sections with the aid of the example design specification for a transportation vehicle presented in Figure 3.3.

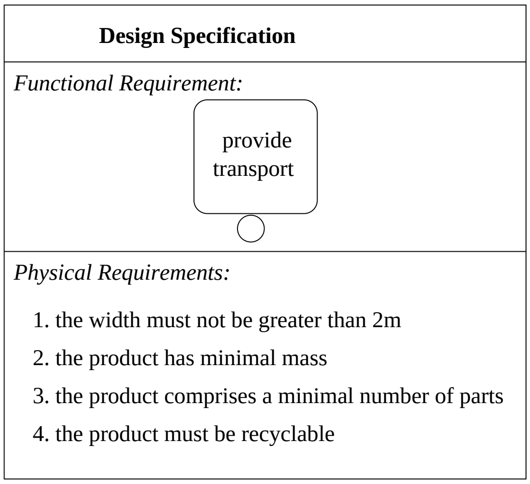


Figure 3.3: An example design specification for a transportation vehicle.

3.2.1 The Functional Requirement

The functional requirement represents an abstraction of the intended behaviour of the product. For example, in the design specification presented in Figure 3.3 the functional requirement is that the product can fulfill the function "*provide transport*". At this stage, there is no association made between the function that is to be provided and the physical mechanism that provides it. The designer's task is to find a physical configuration of parts that satisfies the functional requirement. A design specification should never be more specific about the intended behaviour of a product, for example, by specifying several functions – that would be far from ideal, since it would indicate a decision, made unwisely early, about the means for providing the required functionality. In this research the design specification will always contain only a single functional requirement. This gives the designer greatest scope to search for schemes for the required product.

In some cases the product specification resulting from an analysis of a market need may be couched in terms of a *set* of functions. However, before design commences, this set of functions should be re-stated in terms of a single, more abstract, function from which the designer can begin to develop schemes. This re-statement of a set of functions in terms of a more abstract parent function can be treated as the basis for developing a new design principle which is then used to structure the scheme [1]. the scheme.

3.2.2 Physical Requirements

As stated earlier, the designer's task is to develop a complete physical configuration of parts which satisfies the functional requirement specified in the design specification. However, since the design specification also contains a set of physical design requirements, the physical solutions which the designer develops must not only provide the required functionality but must also satisfy these physical requirements. In Figure 3.2 two classes of physical requirement can be identified: product requirements and life-cycle requirements. A product requirement can be either a categorical requirement that defines a relationship between attributes of the product or it can be a preference related to some subset of these attributes. A life-cycle requirement can be either a categorical requirement that defines a relationship between attributes

[1]Design principles will be presented in Section 3.3.3.

of the product and its life-cycle, or it can be a preference related to some subset of these attributes. Each of these classes of physical requirement are explained further here.

Categorical Physical Requirements

These describe relationships between some subset of the attributes that define a product (a product requirement), or between some subset of the attributes that define a product and its life-cycle (a life-cycle requirement). For example, in the electrical domain the current, resistance and voltage related to a resistor would be regarded as attributes of the resistor while Ohm's Law defines a particular relationship that a resistor must satisfy, namely that the voltage is equal to the product of the current and resistance. This is an example of a product requirement. On the other hand, a life-cycle requirement might forbid the use of particular materials in the design of a product because of an environmental issue or might specify that products to be manufactured using a particular process must have a particular geometry.

The design specification illustrated in Figure 3.3 contains a categorical product requirement that the width of the transportation vehicle be no greater that 2 metres. This implies that it must be possible to compute the width of the product in order to check that this requirement is satisfied. There may be many ways of computing the width of a product depending on the components involved and the manner in which they are configured. The functions used to compute the width of a product would be part of the knowledge of the particular company designing the product.

The design specification illustrated in Figure 3.3 contains one categorical life-cycle requirement, namely that the product be fully recyclable. The meaning of this life-cycle requirement is again part of the design knowledge of the organisation designing the product.

The example categorical physical requirements just discussed are quite simple. However, in general, these requirements can, of course, be arbitrarily complex.

Design Preferences

Often it may not be possible or appropriate to define a categorical relationship over some subset of attributes of the design or its life-cycle. For example, it may not be appropriate to stipulate that the mass of a product be less than a particular number of kilograms, but rather that the mass should be as small as possible. Therefore, a design specification will often contain one or more *design preferences*.

A design preference is a statement about the designers intent regarding some aspect of a product or its life-cycle. For example, a design preference may relate to the value associated with a particular attribute of a product or its life-cycle. A preference for an attribute may be that its value be maximal or minimal. However, design preferences may be also stated about any other aspect of a scheme, such as preferences on the use of particular components or relations over several product or life-cycle attributes.

The design specification illustrated in Figure 3.3 contains two design preferences, namely that the product have minimal mass and comprise a minimal number of parts. The functions used to calculate the values of these design attributes are, generally, part of the design knowledge of the organisation designing the product.

3.3 Conceptual Design Knowledge

During conceptual design, the designer must reconcile the functional and physical requirements for a product into a single model of the product. This means that the designer must synthesise a configuration of parts which satisfies each of the functional and physical requirements in the design specification. To do so, the designer needs considerable knowledge of how function can be provided by physical means. Often, this knowledge exists in a variety of forms. For example, a designer may not only know of particular components and technologies that can provide particular functionality, but may be aware of abstract concepts which could also be used. For example, a designer may know that an electric light-bulb can generate heat or, alternatively, that heat can be generated by rubbing two surfaces together. The latter concept is more abstract that the former. In order to effectively support the human designer during conceptual design, these alternative types of design knowledge need to be defined and modelled in a formal way. In Figure 3.1 it can be seen that design knowledge is used: to develop the function decomposition and the configuration of parts for a product; to form a scheme; and to evaluate and improve the schemes that are generated. The various aspects of the design knowledge used during conceptual design will be discussed in the remaining parts of Section 3.3.

3.3.1 The Function-Means Map

The notion of the *function-means tree* has been proposed by researchers from the design science community as an approach to cataloguing how function can be provided by means [4]. The use of function-means trees in supporting conceptual design has attracted considerable attention from a number of researchers [25, 99]. In general, the level of interest in the use of functional representations in conceptual design has increased in recent times [35], showing growing confidence in the potential of approaches incorporating such techniques.

In this dissertation a generalisation of the function-means tree called a *function-means map* is used to model functional design knowledge. The early developments of the concept of the function-means map has been reported in literature available on the research presented in this dissertation [111, 115]. A function-means map can be used to reconcile functions with means for providing them. In a function-means map two different types of means can be identified: a means can either be a *design principle* or a *design entity.*

A design principle is a means which is defined in terms of functions that must be embodied in a design in order to provide some higher-level functionality. The functions that are required by a particular design principle collectively replace the function being embodied by the principle. The functions which define a design principle will, generally, have a number of *context relations* defined between them. These context relations describe how the parts in the scheme which provide these functions should be configured so that the design principle is used in a valid way. Design principles are discussed in greater detail in Section 3.3.3.

A design entity is a physical, tangible means for providing function. A design entity is defined by a set of parameters and the relationships that exist between these parameters. Design entities are discussed in greater detail in Section 3.3.4.

Before a more detailed discussion of design principles and design entities is presented, the notion of function embodiment will be discussed.

3.3.2 Embodiment of Function

As the designer develops a scheme every function in the scheme is embodied by a means. In this section the icons used to describe the embodiment of functions will be presented.

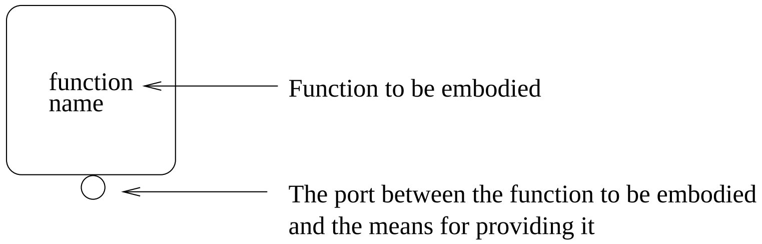


Figure 3.4: The icon used to represent a function to be embodied.

In Figure 3.4 it can be seen that an embodiment icon is defined by a function that is to be embodied and a port which will be connected to a means that is to be used to embody the function. This port is referred to as the "means port" of the embodiment icon. In Figure 3.4 no means has yet been selected for this embodiment since the port is an empty circle.

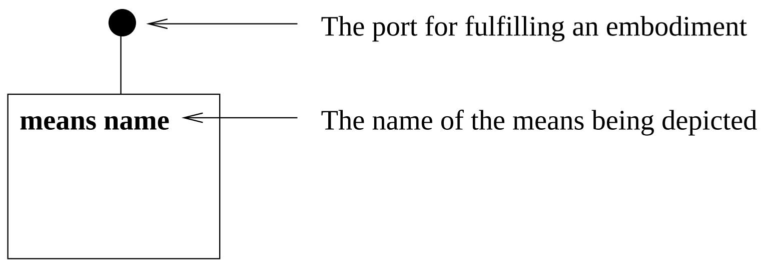


Figure 3.5: The icon used to represent a means.

In Figure 3.5 it can be seen that a means icon is defined by the name of the means being represented and a port which can be connected to the means port of an embodiment icon to indicate that the means is being used to provide the function being embodied. Two types of means are available: a design principle and a design entity. The detailed representation of these different means will be presented later in this chapter.

In Figure 3.6 an example of embodying a function with a means is illustrated. In this example a function *provide light* is embodied using a means called *bulb*. This is indicated by "plugging" the port of the means icon into the port of the embodiment icon for the function. While, in this example, there is a one-to-one mapping between a function and a means, this may not always be the case. A means will often be able to provide more than one function in a design.

Each means that is available to the designer has an associated *set of behaviours*.

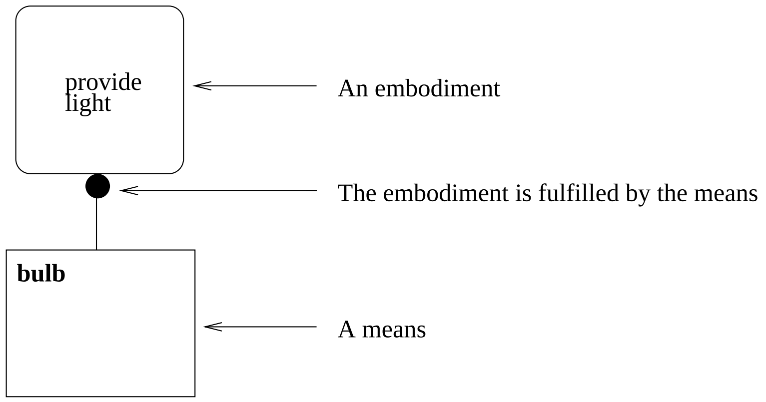


Figure 3.6: A means fulfilling an embodiment.

Each *behaviour* is defined as a set of functions that the means can be used to provide simultaneously. Each behaviour associated with a design principle will contain only one function to reflect the fact that it is used to decompose a single function. However, a behaviour associated with a design entity may contain many functions to reflect the fact the there are many combinations of functions that the entity can provide at the same time. For example, the bulb design entity mentioned above may be able to fulfill the functions *provide light* and *generate heat* simultaneously. However, when a design entity is incorporated into a scheme (for the purpose of supporting functionality provided by one of its behaviours), it is not necessary that every function in this behaviour be used in the scheme.

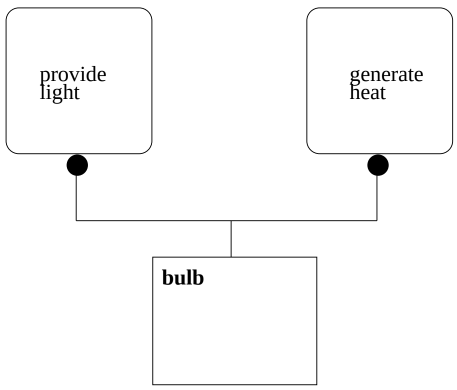


Figure 3.7: An example of entity sharing.

In Figure 3.7 a single bulb design entity is used to embody the functions *provide light* and *generate heat*. In other words these functions share the same entity. This is valid as long as the set of functions that share an entity are a valid behaviour of the entity.

Knowing the various possible behaviours of an entity is useful when the designer is embodying functions. Knowledge of behaviour enables a designer to identify when a particular design entity can be used to provide several functions simultaneously. In this research, the mapping of several functions onto a single design entity is known as *entity sharing*. Entity sharing is a critical issue in design because, without the ability to reason about entity sharing, a designer has no way of removing redundant parts from a design. For example, the body of a car is used to provide many functions because designers know that a single body can provide all of these functions. However, if a designer could not identify the fact that a car body can simultaneously provide many functions, the designer would introduce a different car body for each function to be provided; this would result in a designer attempting to incorporate several car bodies into her design, obviously not a desirable situation.

From a design perspective one of the novel aspects of the work presented in this dissertation is the manner in which design principles and design entities are defined and used. Using design principles to define abstract design concepts in terms of functions and relationships between them is novel. The ability to represent means at both an abstract functional level as well as a physical level provides a basis for a designer to combine and explore new approaches to providing the functionality defined in the design specification. In the remaining parts of Section 3.3 a detailed discussion of the different types of means will be presented.

3.3.3 Design Principles: Supporting Abstraction

A design principle is a statement which declares that a particular function, called the *parent function*, can be provided by embodying a set of *child functions*, provided the embodiments of these child functions satisfy certain *context relations*. A design principle is, therefore, a known approach, defined in abstract terms, to providing functionality. When a designer uses a design principle in a design to embody a particular function, she must incorporate each child function into the function decomposition of the product being designed. The child functions of a design principle replace the parent function at the point in the functional decomposition where the parent function was required.

Context relations define how the design entities which are, ultimately, used to embody the functions in the scheme must relate to each other. Thus, when each function in a function decomposition is mapped onto a design entity for providing it, context relations will define how these entities must be configured or interfaced.

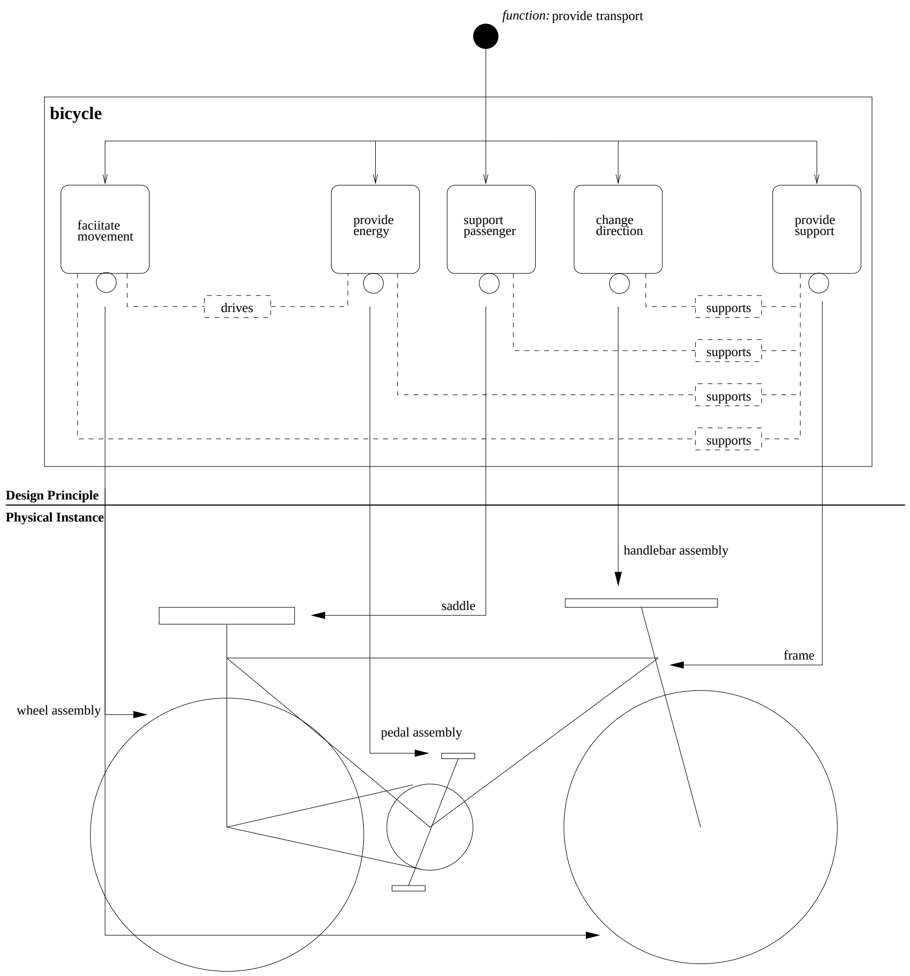


Figure 3.8: Abstracting an example design principle from a bicycle.

Typically, design principles are abstracted from previously successful designs. An example of a design principle, abstracted from a bicycle, is illustrated in Figure 3.8. The parent function in this example is *provide transport*. The child functions are *facilitate movement*, *provide energy*, *support passenger*, *change direction* and *provide support*. These functions are abstracted from the wheels, pedal assembly, saddle, handlebar assembly and frame parts of a bicycle, respectively.

The icon which represents a design principle is the same shape as the icon for a means except that it contains within it a number of icons representing the embodiments that must be fulfilled in order to properly use the principle. The icon for the bicycle design principle is illustrated in Figure 3.9.

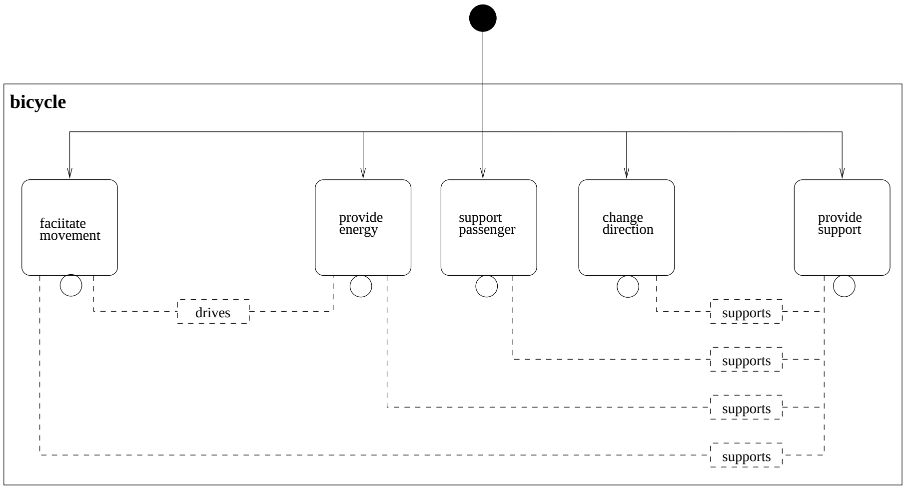
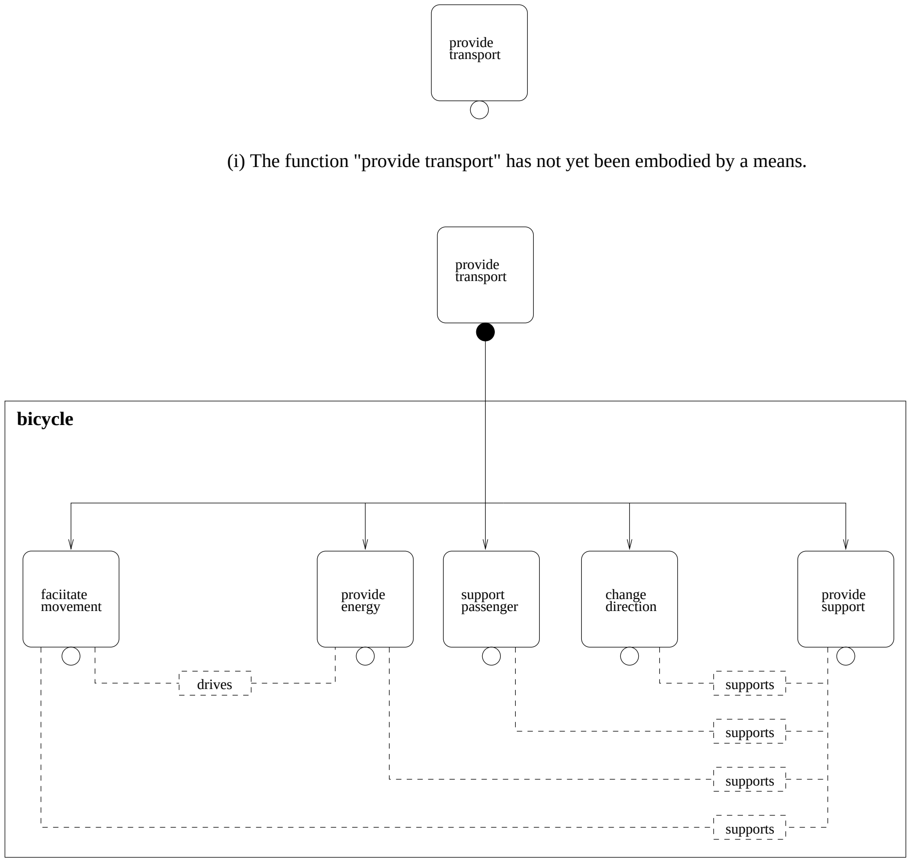


Figure 3.9: The icon used to represent the bicycle design principle.

The major parts of a bicycle from which these functions are abstracted have a context in the bicycle. This context is represented in the design principle as context relations. In the iconic representation of a design principle, context relations are represented as dashed boxes; they are connected to the embodiment icons of the functions to which they relate, using dashed lines. Two types of context relation between the child functions of the bicycle design principle are shown in Figure 3.9. These are the *drives* relation and the *supports* relation. A *drives* relationship must exist between the design entities that are used to provide the function *provide energy* and the function *facilitate movement*. In addition, a *supports* relationship must exist between the the design entities that are used to provide the function *provide support* and each of the functions *facilitate movement*, *provide energy*, *support passenger*, and *change direction*. We will see this principle being implemented in Chapter 4 (Figure 4.31) and being used in Chapter 5 (Figure 5.5).

Ultimately, every function in a scheme is provided by a design entity. The precise meaning of a context relationship depends on the design entities which are finally used to embody the functions between which the context relationship is defined. The context relation will define various relationships which must be considered in order to develop a valid configuration of design entities; as well as providing the required functionality these entities must respect the context relations due to any design principles used. For example, the *drives* context relation which is defined between the function *provide energy* and the function *facilitate movement* implies that those entities which provide the function *provide energy* must be capable of driving the entities which provide the function *facilitate movement*. This may mean that, if the function *facilitate movement* is embodied by a *wheel assembly* design entity and the function *provide energy* is embodied by a *pedal assembly* design entity, there must exist a *chain* design entity between the *wheel assembly* and the *pedal assembly*. Later in this chapter, once design entities have been discussed, the meaning of context relations will be revisited.



(i) The function "provide transport" has not yet been embodied by a means.

(ii) The function "provide transport" is embodied by the design principle "bicycle".

Figure 3.10: Using a design principle to embody a function.

In Figure 3.10 the embodiment of a function using the bicycle design principle is illustrated. In Figure 3.10 (i) it can be seen that there is a function *provide transport* whose embodiment is not complete, since the means port for the embodiment icon is empty. In Figure 3.10 (ii) the designer has embodied the function using the bicycle design principle. The effect of this is that the embodiment of the function *provide transport* is complete, but five further embodiments must now be considered. These five embodiments have been introduced due to the five child functions of the bicycle design principle. These embodiments have empty circles as their ports, to show that they have not yet been fulfilled. The designer must fulfill each of these embodiments by selecting means for them.

Design principles are used by a designer to develop the function decomposition upon which a scheme will be configured. At the functional level, what is of concern to the designer is the development of a function decomposition of a product from the functional requirements specified in the design specification. The designer is generally not concerned with how this detailed function decomposition should be embodied in a physical sense. The use of a design principle in a scheme increases the level of detail of the function decomposition, by increasing the number of functions and context relations to be considered by the designer. In this way, the use of a design principle increases the complexity of the function decomposition. If the number of functions in the function decomposition increases, the complexity of the final scheme may also be greater, since there is the possibility that a designer will employ a larger number of design entities. Since there may be a choice of more than one design principle to provide a given function, there may be many alternative function decompositions which provide same the required functionality. Designers should investigate as many alternative function decompositions as possible.

3.3.4 Design Entities: Design Building Blocks

A design entity defines a class of physical parts or sub-assemblies which can provide specific functionalities. In order to give physical form to a particular function decomposition, functions are embodied by design entities. Entities are defined in terms of a set of *attributes*. Each attribute defines a parameter associated with a design entity and its domain of possible values. For example, an attribute may be used to define the mass of a design entity which may take its value from the set of real positive numbers. Alternatively, an attribute may be used to define the colour of a design entity which may take its value from a set of colours. There may also be a number of constraints on these attributes which describe the relationships between them. For example, if the notion of a resistor is modelled as a design entity, it may be defined in terms of a set of attributes representing the voltage, resistance and current associated with it. In addition, there will be a constraint which will define the relationship between these attributes due to Ohm's Law. Since schemes are configured from design entities, a number of functions must be pre-defined over design entities, and over configurations of them, in order to facilitate the evaluation of schemes against the various criteria defined in the design specification. For example, in order to compute the power output from an electrical circuit, functions which compute the power output of each design entity used in the scheme must be available. In the case of the resistor, such a function could be that the power output of a resistor is the product of the voltage and current flowing through it.

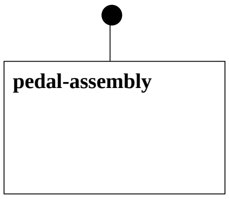


Figure 3.11: The icon used to represent a pedal assembly.

The icon for an example design entity, a pedal-assembly, is illustrated in Figure 3.11. This design entity could be part of a design knowledge-base used to generate schemes for the vehicle design specification presented Figure 3.3.

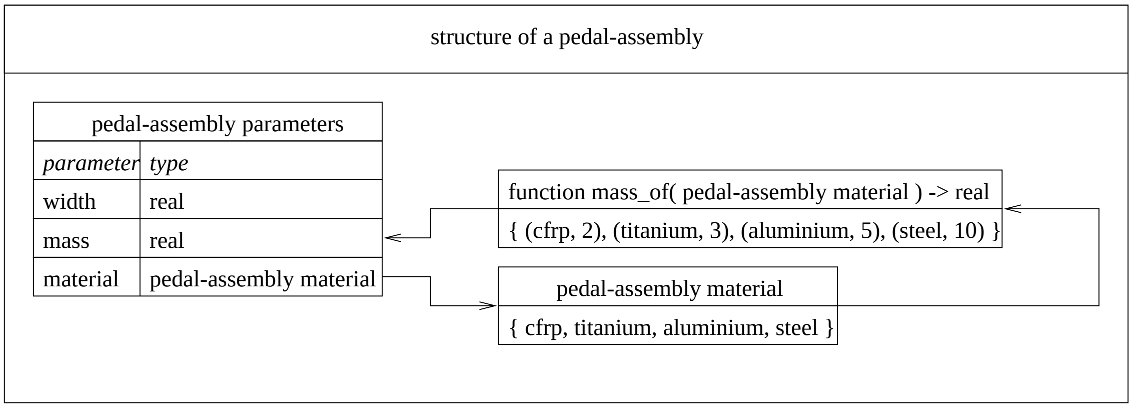


Figure 3.12: A graphical representation of an example design entity.

There are many possible representations for this design entity, one of which is illustrated in Figure 3.12. It can be seen that the pedal-assembly design entity is defined by three attributes: *width*, *mass* and *material*. The *width* and *mass* of the pedal-assembly are real numbers, while the *material* of the pedal-assembly takes its value from a set of materials, called pedal-assembly materials. The materials that are available are *cfrp*, *titanium*, *aluminium* and *steel*. There is a function available called *mass_of* which can estimate the mass of a pedal-assembly from its *material*.

For example, if the *material* of a pedal-assembly is *cfrp*, the estimated mass of the pedal-assembly is 2. The functions defined over design entities can be arbitrarily complex.

3.3.5 Scheme Configuration using Interfaces

When a designer begins to develop a scheme for a product, the process is initiated by the need to provide some functionality. The designer begins to develop a scheme to provide this functionality by considering the various means that she has available in her design knowledge-base. Generally, the first means that a designer will select will be a design principle. This design principle will substitute the required (parent) functionality with a set of child functions.

As the designer develops a scheme and produces a function decomposition tree she will ultimately embody all leaf-node functions in the scheme with design entities. During this embodiment process the context relations, from the design principles used in the scheme, will be used as a basis for defining the interfaces between the design entities used in the scheme. The precise nature of these interfaces cannot be known with certainty until the designer embodies functions with design entities; this is because the link between functions and design entities is generally not known with certainty during the development of the function decomposition for the scheme.

The types of interfaces that may be used to configure a collection of design entities will be specific to the engineering domain within which the designer is working. For example, the set of interfaces that a designer working in the mechanical domain would typically use are different to those used by a designer in the electrical domain. Indeed, these interfaces may also be specific to the particular company to which the designer belongs.

In Section 3.4 an example of scheme generation will be presented. As part of this presentation a detailed discussion of how context relations drive the configuration of a scheme will be presented.

3.4 Scheme Generation

In Figure 3.1 it can be seen that scheme generation is an iterative process comprising a number of activities. These activities relate to the development of a function decomposition for the scheme, the development of a configuration of design entities based on this function decomposition and an evaluation and comparison of the newly developed scheme with those schemes which have already been developed. As new schemes are developed, the designer will constantly consider which schemes to accept, reject or improve.

Once the design specification has been formulated the designer must attempt to search for as many schemes as possible which have the potential to satisfy the requirements in the design specification. The first step in developing a scheme is to explore the functional requirement and develop a function decomposition which can be used to configure a set of design entities to form a scheme. Developing a function decomposition involves substituting higher level functions with sets of equivalent functions which collectively provide the required functionality. To assist the designer in developing this decomposition, design principles are used.

There are generally many function decompositions possible from the same functional requirement. Generating alternative function decompositions can be regarded as a way of systematically exploring the design space for a product. This search is executed by a designer by considering the functional requirement for the product and exploring alternative ways of providing the functionality by using design principles to decompose the functional requirement into less abstract functions. In this way the designer can reduce the functional requirement into one which allows the use of standard technologies, represented as design entities, to satisfy it.

Using a function decomposition the designer can begin to develop a configuration of design entities. Each leaf-node function in a particular function decomposition must be provided by a design entity. The context relations inherited from the branch-nodes in the function decomposition, due to the use of particular design principles, are used to define the manner in which the design entities used in the scheme should be configured or interfaced. Some context relations will define constraints on the spatial relationships between design entities, while others may define particular types of interface which may be required between the entities. As the designer incorporates design entities into a scheme the various physical requirements which are described in the design specification can be brought to bear on it.

Before discussing how a designer can evaluate and compare schemes, an example of scheme development will be presented in Section 3.4.1.

3.4.1 An Example of Scheme Generation

In this section an example of how schemes are developed using the design theory described here, is presented. A number of schemes will be developed based on the vehicle design specification presented in Section 3.2. The process of scheme development presented here will be illustrated, in an example designer-computer interaction, in Chapter 5.

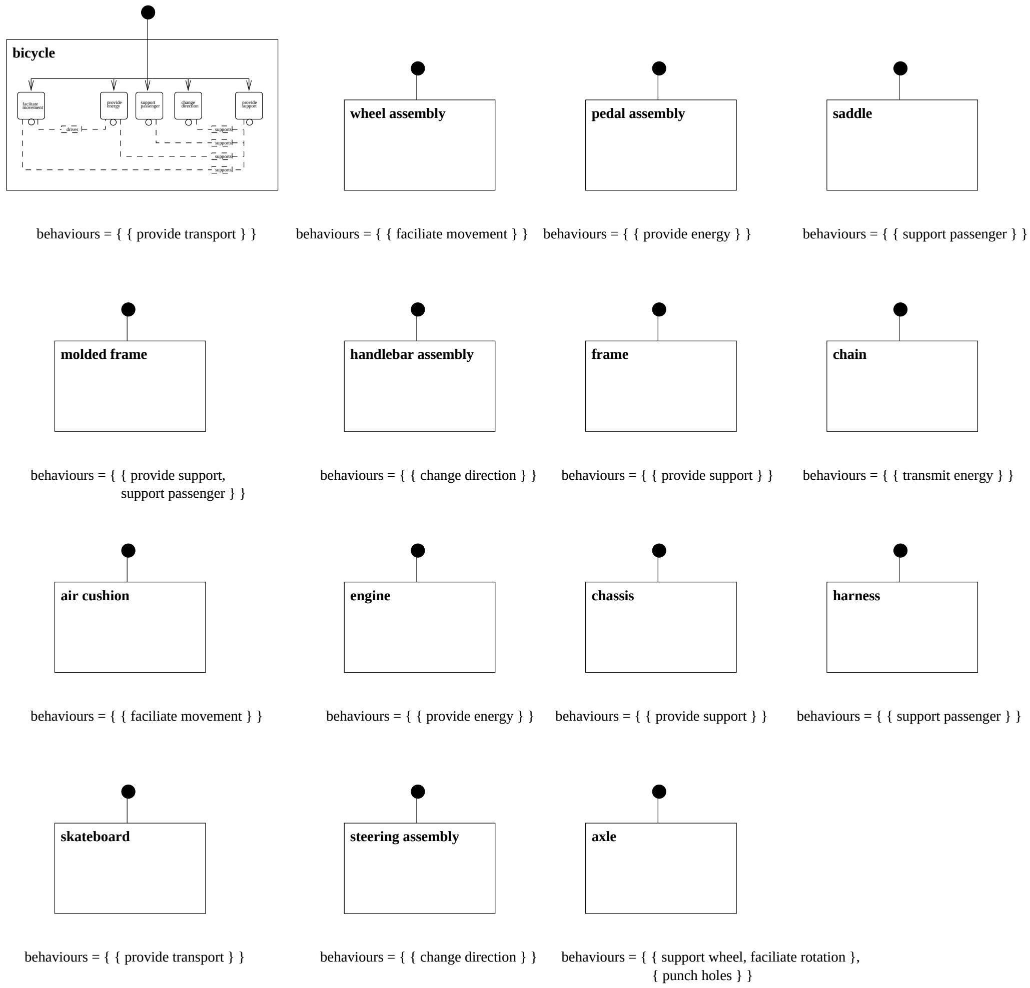


Figure 3.13: The means contained in an example design knowledge-base and their possible functionalities.

In Figure 3.13 an illustration of the means contained in an example design knowledge-base is presented. This knowledge-base comprises one design principle, called *bicycle*, and a number of design entities, such as a *wheel assembly* and a *saddle*. The set of behaviours for each means in the knowledge-base are presented under the icon representing the means. Remember that behaviour is a *set* of functions that the means can provide simultaneously (Section 3.3.2). Thus the behaviours for a means is a set of sets. Most of the means in this example knowledge-base have only one behaviour; that is, the set of behaviours for each means contains only one set of functions. Furthermore, most of the behaviours of the means in this knowledge-base can provide only one function at a time. However, the *molded frame* and *axle* design entities have more complex behaviours. The *molded frame* entity can provide two functions simultaneously: *provide support* and *support passenger*. The *axle* entity has two behaviours: it can provide the two functions *support wheel* and *facilitate rotation* simultaneously, but can also be used to provide the single function *punch holes*.

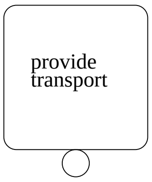


Figure 3.14: The functional requirement for a product.

The functional requirement from the vehicle design specification in Figure 3.3 is illustrated in Figure 3.14. It can be seen from this figure that the functional requirement is that the scheme must provide the function *provide transport*. To simplify this discussion of scheme development, the various physical requirements will not be considered until the scheme has been configured. However, in general a designer may be able to consider some of these requirements throughout the scheme development process.

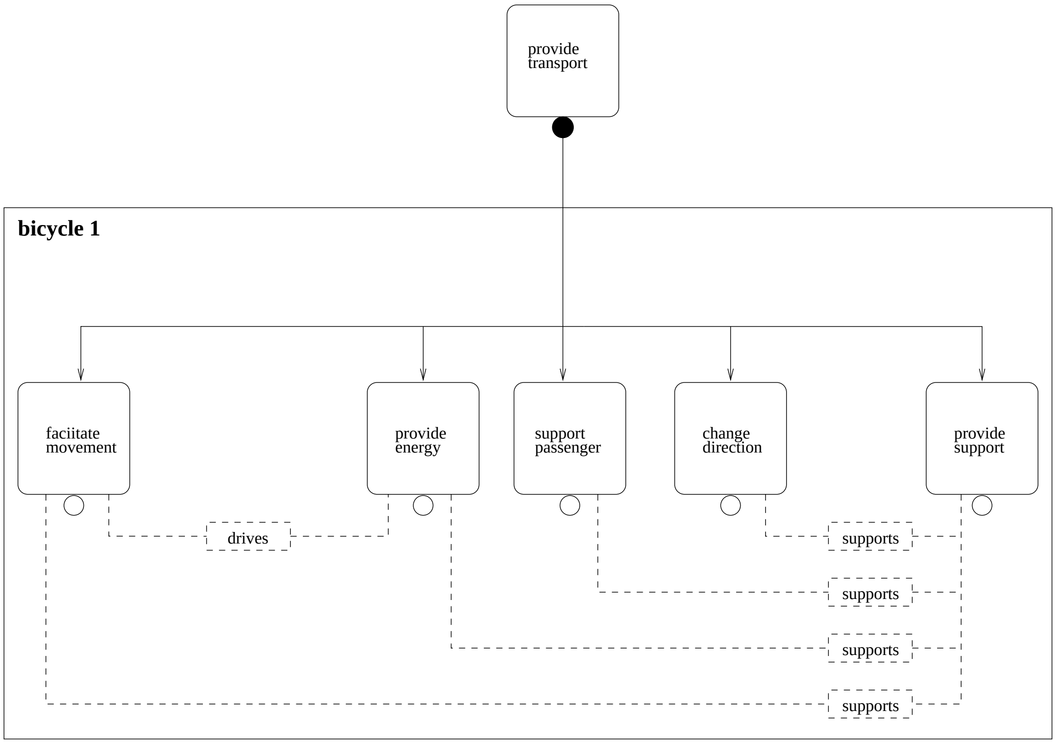


Figure 3.15: Using a design principle to embody the functional requirement.

In Figure 3.15 an instance of the design principle *bicycle*, called *bicycle 1*[2] , has been used to embody the function *provide transport*. This design principle introduces the need for five more functions to be embodied. In particular, the bicycle design principle introduces the need to embody the functions *facilitate movement*, *provide energy*, *support passenger*, *change direction* and *provide support*. The designer must now select means for embodying each of these functions.

[2]When a designer selects a means to embody a function, an instance of the means in incorporated into the scheme. Thus *bicycle 1* is the first instance of the *bicycle* design principle to be used by the designer.

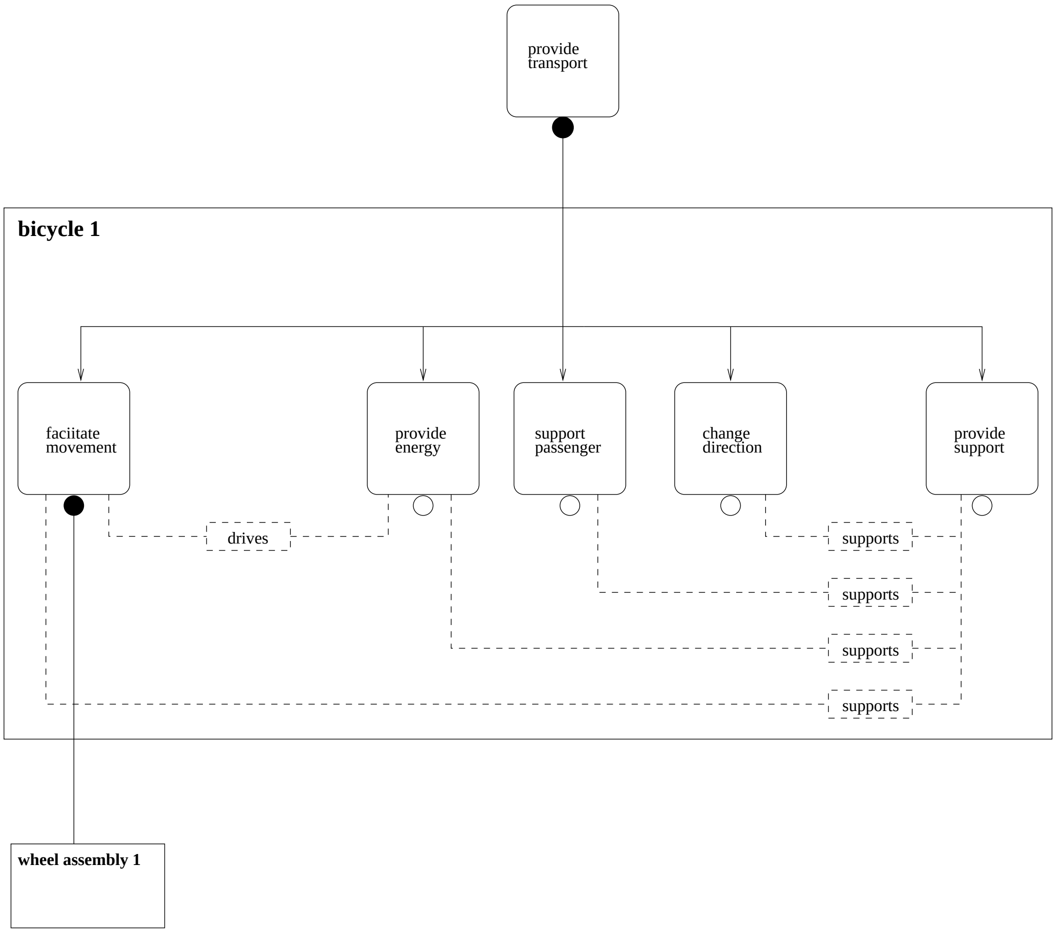


Figure 3.16: Using a design entity to embody a function in the scheme.

In Figure 3.16 the designer selects the *wheel assembly* design entity to embody the function *facilitate movement*. This introduces an instance of this means, called *wheel assembly 1*, into the scheme. As the designer introduces design entities into the scheme the context relations that exist between the function embodiments must be considered. However, since there is only one design entity in the scheme presented in Figure 3.16 no context relations are considered at this point in the scheme's development.

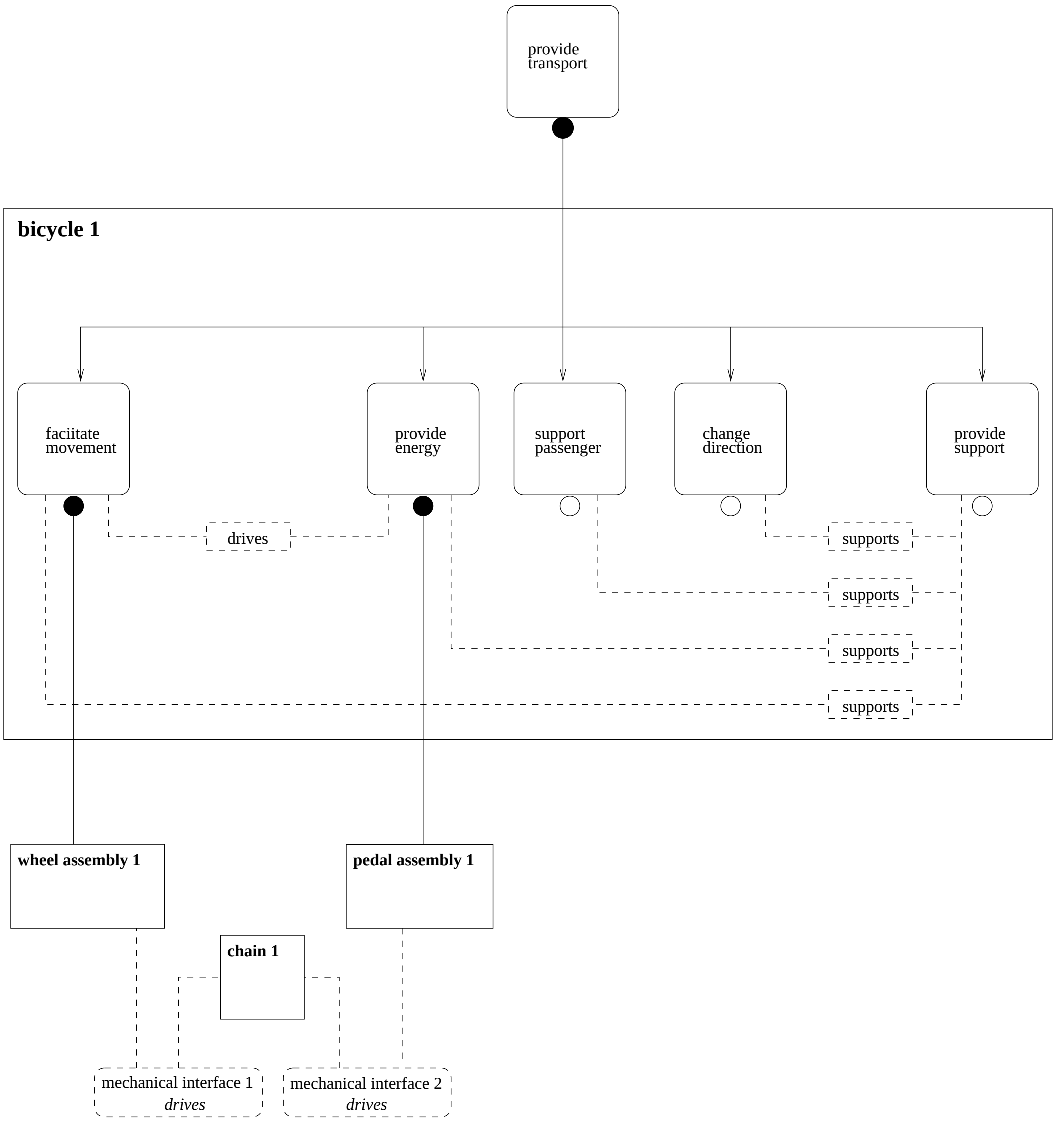


Figure 3.17: The effect of a context relation on the configuration of design entities.

In Figure 3.17 the designer has chosen to embody the function *provide energy* with the *pedal assembly* design entity. This introduces an instance of this means, called *pedal assembly 1*, into the scheme. Since the *drives* context relation must exist between the embodiments of the functions *facilitate movement* and *provide energy*, the designer must consider this context relation between *wheel assembly 1* and *pedal assembly 1*. Before the meaning of this context relation is discussed, the approach to reconciling context relations with the entities to which they relate will be presented.

The meaning of a context relation is part of the company-specific design knowledge that is used to develop schemes. Context relations are implemented as interfaces between design entities. An interface is used to define some relationship that should exist between design entities. The design knowledge-base of a company will contain a repertoire of interfaces that are used in the conceptual design of products in the company. For example, a company may have interfaces for handling various types of mechanical, spatial or electrical relationships between design entities. The meaning associated with a context relation may be that an interface of a particular type should exist between the design entities, the precise nature of this interface being specified during a later phase of design. Alternatively, a context relation may introduce new design entities into a scheme with interfaces between them and the design entities between which the context relation should be satisfied. The precise meaning of a context relation may depend on the type of design entities between which the context relation must be considered.

In order to be able to identify the design entities between which a context relation should be considered it is necessary to be able to identify which design entities *derive from* which functions. Each design entity used in a scheme derives from at least one function. Essentially, a context relation must be considered between design entities that derive from the embodiments between which a context relation is defined.

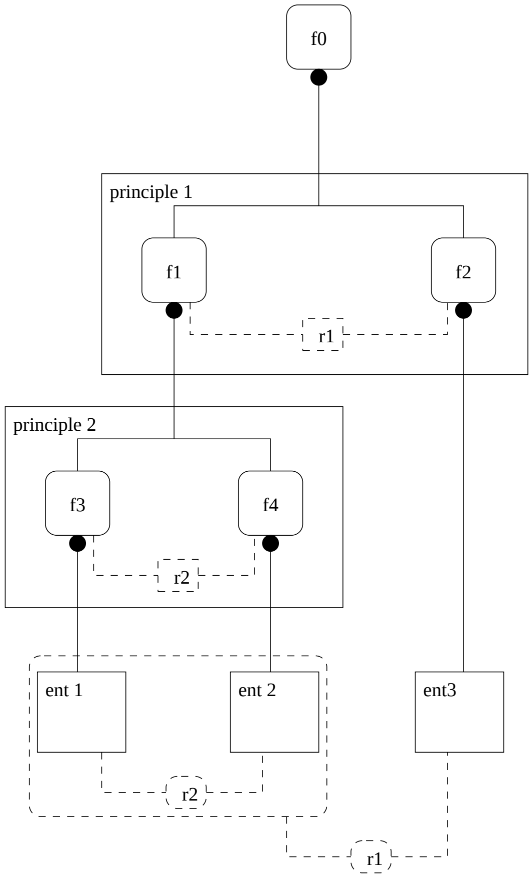


Figure 3.18: The structure of a scheme illustrating the use of context relations in configuring a scheme.

In Figure 3.18 an example scheme structure is illustrated. The top-level function in this scheme is *f0*. This function is embodied using a design principle called *principle 1*. This design principle introduces two functions, *f1* and *f2* to replace the function *f0*. A context relation, *r1*, is specified between these functions. The function *f1* is embodied by a design principle, *principle 2*, which introduces two further functions, *f3* and *f4* into the scheme. A context relation, *r2*, is specified between these functions. The function *f3* is embodied with the design entity *ent 1*, the function *f4* is embodied with the design entity *ent 2* and the function *f2* is embodied with the design entity *ent 3*. However, between which design entities should the context relations *r1* and *r2* be considered?

The context relation *r2* must exist between the entities that derive from the functions *f3* and *f4*. The design entities *ent 1* and *ent 2* are used to embody these functions. Thus, the context relation *r2* must be considered between these entities. Since, the design entities *ent 1* and *ent 2* are the means used to provide the functions *f3* and *f4*, these entities can be regarded as being *directly* used to provide these functions.

The context relation *r1* is a little more complex. This context relation must exist between the entities that derive from the functions *f1* and *f2*. The design entity *ent 3* is used to embody the function *f2*. Thus this entity can be regarded as being *directly* used for the function *f2*. The function *f1* is provided by the design principle *principle 2*, whose child functions are in turn embodied by the design entities *ent 1* and *ent 2*. Thus, these design entities can be regarded as being *indirectly* used to provide the function *f1*. Therefore, the context relation *r1* must be considered between the combination of design entities *ent 1* and *ent 2* on the one hand, and *ent 3* on the other hand.

In Figure 3.17, the *drives* context relation that exists between the embodiments of the functions *facilitate movement* and *provide energy* must be considered between the design entities *wheel assembly 1* and *pedal assembly 1*. In this figure it can be seen that this caused, in addition to the existence of the design entities *wheel assembly 1* and *pedal assembly 1*, the introduction of an instance of the *chain* design entity, called *chain 1*. Furthermore, it caused a *mechanical interface* between *chain 1* and *wheel assembly 1* and another between *chain 1* and *pedal assembly 1*. Both of these interfaces are used, along with *chain 1*, to embody the *drives* relation that should exist between *wheel assembly 1* and *pedal assembly 1*.

In the company design knowledge-base that is used here, one type of interface that is available is a *mechanical interface*. A *mechanical interface* can be used to define many relationships between design entities. One of these relationships is a form of *drives* relationship. During detailed design a more precise specification for this mechanical interface would be given.

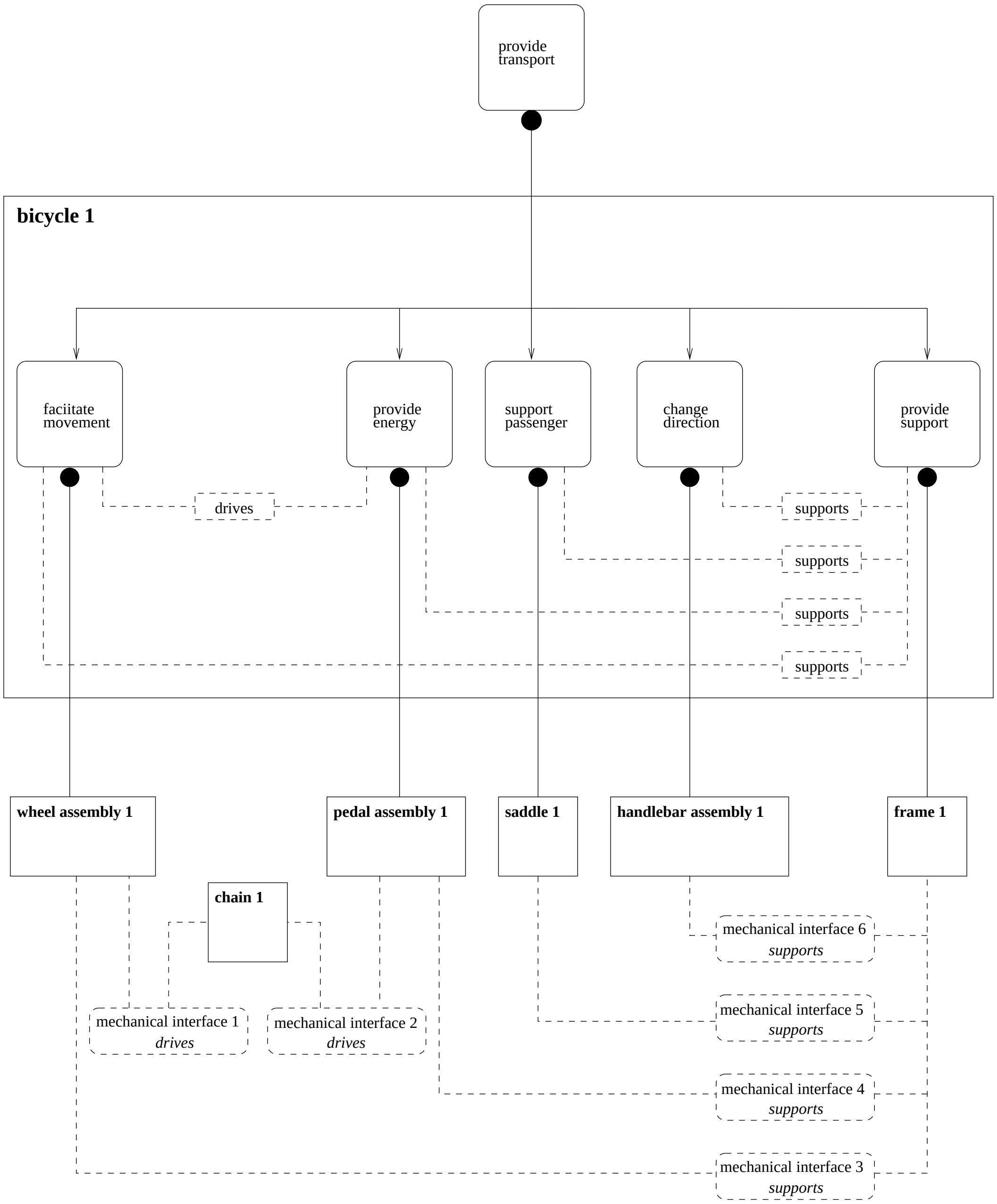


Figure 3.19: An example scheme configuration.

Figure 3.19 shows the state of the scheme after the designer has chosen to embody the function *support passenger* with the design entity *saddle*, the function *change direction* with the design entity *handlebar assembly* and the function *provide support* with the design entity *frame*. Due to the *bicycle* design principle, a context relation called *supports* must exist between the embodiment of the function *provide support* and the embodiments of each of the functions *facilitate movement*, *provide energy*, *support passenger* and *change direction*. Each of these context relations is embodied by a *mechanical interface* that defines a *supports* relationship. The details of these mechanical interfaces that define a *supports* relationship will be specified during detailed design.

Since all the functions have been embodied in the scheme presented in Figure 3.19, the designer must now begin to select values for the attributes associated with each design entity in the scheme. In making these decisions the designer must ensure that the various constraints that are imposed on her due to the design specification or the design knowledge-base must be satisfied. In addition, this scheme must be compared with any other alternative scheme for this product that are developed. However, before the process of evaluating and comparing schemes is discussed, the structure of a second scheme will be presented.

A second scheme is presented in Figure 3.20. This scheme is also based on the *bicycle* design principle. In this scheme the function *facilitate movement* is provided by an instance of the design entity *wheel assembly*, called *wheel assembly 2*, while the function *provide energy* is provided by an instance of the design entity *engine*, called *engine 1*. The *drives* relation that must exist between *wheel assembly 2* and *engine 1* is embodied by a *mechanical interface* between *wheel assembly 2* and an instance of the *chain* design entity, called *chain 2*, and a *mechanical interface* between *engine 1* and *chain 2*.

The function *support passenger* is provided by an instance of the design entity *molded frame*, called *molded frame 1*, while the function *change direction* is provided by an instance of the design entity *handlebar assembly*, called *handlebar assembly 2*. The function *provide support* is also provided by the design entity *molded frame*. In Figure 3.13 it was shown that one of the behaviours of a *molded frame* is that it can provide the functions *provide support* and *support passenger* simultaneously. Thus, since an instance of the *molded frame* design entity already exists in the scheme, there is no need for a second one to be introduced. Thus, the functions *provide support* and *support passenger* share the instance of the design entity *molded frame* called *molded frame 1*.

In order to embody the *supports* context relation between the embodiment of

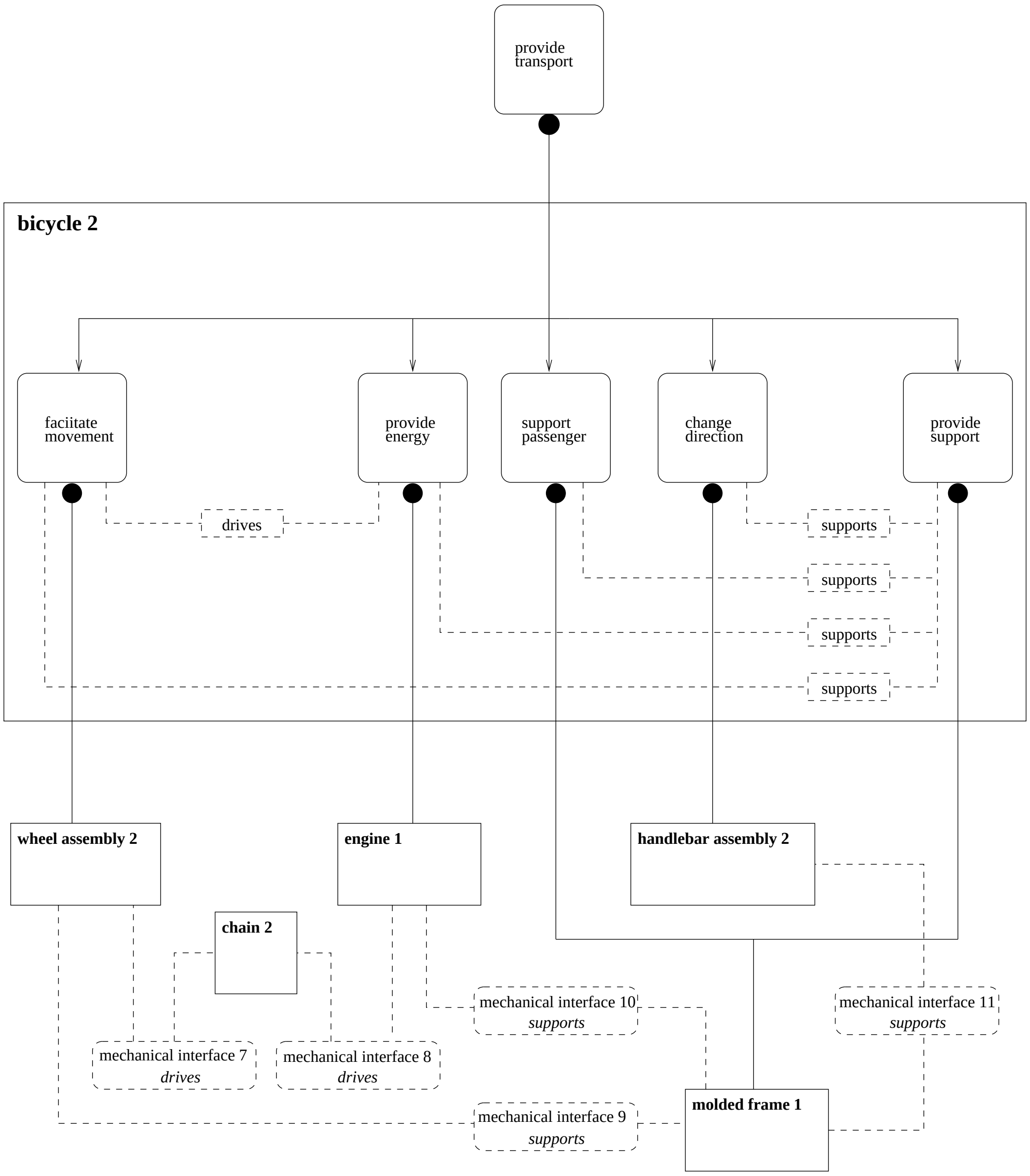


Figure 3.20: A second example scheme configuration.

the function *provide support* and the embodiments of each of the functions *facilitate movement*, *provide energy*, *support passenger* and *change direction*, a number of *mechanical interfaces* are used. These interfaces define a *supports* relationship between the appropriate design entities.

3.4.2 Evaluation and Comparison of Schemes

The designer's primary objective during conceptual design is to develop a set of alternative schemes which satisfy both the functional and physical requirements defined in the design specification. This set of schemes will be further refined during subsequent stages of the product development process until a small number of fully specified designs, possibly just one, will be selected for commercialisation.

In developing a particular scheme a designer must ensure that she does not violate any of the physical requirements defined in the design specification. In Section 3.2.2 two types of physical requirements were presented: product requirements and life-cycle requirements. Where possible during the development of a scheme, the designer should consider these requirements and ensure that the scheme being developed has the potential to produce a "good" product.

Every scheme must satisfy the categorical product and life-cycle requirements that are defined in the design specification. If a scheme violates one of these the designer must either reject the scheme which has been developed or modify it in order to satisfy them. For example, in the vehicle design specification presented in Figure 3.2 the categorical product requirement that the width of the vehicle be no more than two metres must be satisfied by every scheme. For the two schemes that have been developed here, the designer will, at some stage in the design process, need to define a spatial configuration of the design entities used in each scheme in order to estimate the width for each scheme. Likewise, the categorical life-cycle requirement that the scheme by recyclable must also be satisfied by every scheme. This requirement may mean that every design entity in the scheme be manufactured from a recyclable material. Thus, when the designer selects materials for each of the design entities in the scheme this life-cycle requirement can be checked.

While every scheme may satisfy the categorical physical requirements, not every one of these schemes will be selected for development in subsequent phases of design. This is due to particular schemes not "satisfying" the preferences that were defined in the design specification. For example, in the vehicle design specification presented in Figure 3.3, two preferences were defined: the product should have minimal mass and comprise a minimal number of parts. As the designer develops a scheme an estimate of its mass can be made. This estimate may be based on the materials used for each design entity and, possibly, the geometry of the design entity in the scheme. The number of parts can be regarded being the same as the number of design entities in the scheme.

In Chapter 2 the principle of Pareto optimality was introduced as a way of determining the best solutions to a problem involving multiple objective functions. Each design preference can be regarded as an objective function. Thus, the best schemes that are developed are those that are not dominated, in the Pareto optimal sense, by any other scheme. Obviously, in order to compare two schemes they must be based on the same design specification.

Property	*Preference*	*First Scheme*	*Second Scheme*
number of parts	minimal	6	5
mass	minimal	$< x$	x

Table 3.1: Using the principle of Pareto optimality to compare two schemes.

Table 3.1 presents a comparison of the two schemes shown in Figure 3.19 and Figure 3.20. The first scheme comprised six parts while the second scheme comprised five. Thus, on the preference for a scheme comprising a minimal number of parts, the second scheme is better than the first. At this point it is not possible to determine if the first scheme is dominated. However, if the first scheme is not to be dominated by the second scheme it will have to have a smaller mass than the second scheme. In this way, each scheme would be better than the other on one design preference.

Using the principle of Pareto optimality provides a useful basis for comparing alternative schemes. It can be used to identify schemes that are completely dominated by schemes which have already been developed; this should motivate a designer to modify a scheme so that it is no longer dominated. Using the principle of Pareto optimality a designer can compare schemes that have been developed in order to select those that will be developed further and in order to identify those schemes that should be either improved or discarded. However, a designer should not be *forced* to discard dominated schemes. The objective is to motivate the designer to think about ways of improving them.

3.5 Learning during Conceptual Design

During the conceptual phase of design a designer is faced with a number of problems, such as interpreting a design specification and developing a set of alternative schemes for it. While designing the designer may discover something new about the design problem that she is addressing or may discover that an issue which should have been considered earlier was not. Thus, as designers design they often develop a better understanding of the particular problem they are working on. This learning process may often result in new requirements being incorporated into the design specification or may cause the introduction of new means into the design knowledge-base being used. This is illustrated in Figure 3.1. Therefore, the designer will often need to return to work which has already been done in order to ensure that new design requirements do not invalidate any schemes that have been developed. In this way the conceptual design process is very dynamic.

3.6 Summary

This chapter presented the approach to conceptual design upon which the research presented in this thesis is based. It was noted that conceptual design is an extremely demanding phase of the product development process. In order to assist a designer during conceptual design a computer needs be to capable of reasoning about the technical and non-technical aspects of the process. The input to the conceptual phase of design is the specification of the product which is to be designed. During conceptual design the human designer develops a number of alternative schemes which have the potential to satisfy the design specification.

The chapter presented a novel perspective on design knowledge that can be used during conceptual design. This perspective involves formulating design knowledge into a mapping between functionality and means for providing functionality. It was shown how means could be described either at a parametric level using design entities or an abstract functional level using design principles. The various issues associated with developing schemes that satisfy a set of requirements, using this knowledge, were also discussed.

Finally, this chapter presented an approach for evaluating and comparing alternative schemes using the principle of Pareto optimality.

Chapter 4

Implementation Strategy

One of the most difficult aspects of providing computer-based support for conceptual engineering design is the modelling of products at various levels of abstraction. This chapter presents an approach to using constraints as a modelling paradigm for engineering design. In particular, this chapter describes a constraint-based implementation of the conceptual design theory presented in Chapter 3. The constraint programming language Galileo is used as the modelling language. This chapter demonstrates how the design knowledge that is required during the conceptual phase of design can be modelled in Galileo; it also shows how constraint-based modelling can be used as a basis for developing and evaluating schemes for a product described in a design specification. This chapter uses material from Appendix B and Appendix C. Appendix B provides full Galileo implementations of the generic conceptual design concepts presented in this chapter. Appendix C provides full Galileo implementations of a company-specific design knowledge base and a project-specific design specification. The modelling techniques that are presented in this chapter will be used in Chapter 5 where an interaction scenario between a designer and a conceptual design advice system will be presented.

4.1 Modelling for Conceptual Design

Conceptual design involves considering a specification for a product and developing a set of alternative schemes for it. This transformation, from an abstract specification to a set of alternative schemes, requires a range of different types of knowledge which, if a computer is to assist in the conceptual design process, must be represented in the machine. First, in order for the machine to understand the design problem that is being addressed by the designer, the *design specification* for the product which is to be developed must be modelled in some way. Second, if the machine is to assist the human designer, the *design knowledge* possessed by the company for which the designer is working must be modelled. Third, in order to evaluate each alternative scheme that the designer attempts to develop, the various *evaluation criteria* that the designer uses must also be modelled.

4.2 The Implementation Language Used: Galileo

The implementation language used in this research was Galileo [22, 49], a frame-based constraint programming language derived from the First-Order Predicate Calculus (FOPC). It is assumed in the remainder of this dissertation that the reader is familiar with Galileo. For those readers who wish to familiarise themselves with the language, an overview is presented in Appendix A. Section A.3 of this appendix presents an explanation of constraint filtering, the form of constraint processing which is provided by Galileo and which is used as the basis for supporting designers in this dissertation.

There were several reasons for using Galileo. First, Galileo is an extremely expressive language which has been developed for building systems to support engineering design. Second, since FOPC is widely used to represent the semantics of natural language, a programming language derived from it, such as Galileo, seems to offer a good basis for re-stating in a computer-understandable formalism any body of knowledge for which only a natural language formulation exists, such as a theory of conceptual design. Third, a more pragmatic reason, the centre in which this research was carried out is developing the Galileo language and its associated tools to suit various aspects of the design process and, since all previous developments were motivated by applications in *detailed* design, there was interest in seeing how well the language would support applications in *conceptual* design.

Two questions were to be addressed by the research presented in this dissertation. First: could constraint-based reasoning be used as the basis for supporting conceptual design? Second: would the expressiveness needs of conceptual design motivate the introduction of new features into Galileo?

The existence of this dissertation indicates that the first question was answered in the positive. Regarding the second question: although the pre-existing version of Galileo *could* have been used to support conceptual design, it was found that, by adding a few features to the language, program length could be reduced and the user-interface could be improved. Before the implementation of the conceptual design theory presented in Chapter 3 is discussed, these extensions to Galileo will be introduced.

4.2.1 An Enhancement to the Semantics of Quantification

In the current standard version of Galileo, quantification over structured domains only applies to top-level parameters. For example, in Figure 4.1[1], the universally quantified constraint over the structured domain `room` in Lines 9–10 would only apply to top-level parameters which are instances of this structured domain; it would not apply to fields of other parameters, even if those fields were also instances of the same structured domain.

```
 1 domain house
 2     =::= ( bedroom : room
 3            price   : real ).

 4 domain room
 5     =::= ( length : real,
 6            width  : real ).

 7 h1: house.
 8 r1: room.

 9 all room(R):
10     R.length >= R.width.
```

Figure 4.1: Example usage of quantification in Galileo

Thus, in standard Galileo, while the constraint in Lines 9–10 would apply to room `r1` (the top-level parameter declared in Line 8), it would *not* apply to `h1.bedroom` (a field of the top-level parameter declared in Line 7), even though `h1.bedroom` is also of type `room` (Line 2).

In this dissertation, it is proposed that all quantified constraints apply to *all* instances of the domain over which they are quantified, regardless of whether these

[1]In this dissertation, examples of Galileo code are presented annotated with line numbers on the left. Please note that these line numbers are *not* part of the Galileo program, but are introduced in this dissertation to facilitate easy reference to particular lines in a Galileo program.

instances are top-level parameters or merely fields of parameters. For example, in Figure 4.1 the universally quantified constraint in Lines 9–10 would apply to all instances of structured domain `room`. Thus, the universally quantified constraint in Lines 9–10 would also apply to the `bedroom` field of all instances of the structured domain `house`; in other words, it would apply to `h1.bedroom` as well as to `r1`.

4.2.2 Applying the "`!`" operator to the `exists` Quantifier

In this dissertation it is proposed that the `!` operator [23] be applicable to the "`exists`" quantifier. Before explaining what this means, the semantics of the "`exists`" quantifier in Galileo will be discussed.

```
1 domain store_room
2     =::= ( length  : real,
3            breadth : real,
4            number_of_contents : nonnegative integer ).

5 number_of_boxes : nonnegative integer.

6 number_of_boxes > 0 implies
7     exists store_room(S):
8         S.number_of_contents = number_of_boxes.

9 number_of_boxes = 10.
```

Figure 4.2: Example usage of the "`exists`" quantifier.

If the program in Figure 4.2 were submitted to a Galileo interpreter, a constraint violation message would be produced. The parameter `number_of_boxes` has the value 10 (Line 9). The constraint in Lines 6–8 deduces from this fact that there ought to exist a parameter of type `store_room` (defined on Lines 1–4) whose `number_of_contents` field should also have the value 10. However, no such parameter exists, so the constraint in Lines 6–8 is violated.

Now consider a modified version of this program, as presented in Figure 4.3. The only difference between the two programs is that, in Figure 4.3, the `!` operator is applied to the `exists` quantifier in Line 7. The `!` operator [23] tells the Galileo interpreter that, if a constraint in which the operator is used is violated, the interpreter system should perform some operation (in effect, make a non-monotonic assumption) which eliminates the violation. The `!` operator is only allowed in cer-

```
1 domain store_room
2     =::= ( length  : real,
3            breadth : real,
4            number_of_contents : nonnegative integer ).

5 number_of_boxes : nonnegative integer.

6 number_of_boxes > 0 implies
7     !exists store_room(S):
8        S.number_of_contents = number_of_boxes.

9 number_of_boxes = 10.
```

Figure 4.3: Example usage of the "!" operator applied to the `exists` quantifier.

tain situations and, in the current standard version of Galileo, it cannot be applied to the `exists` quantifier. The operational semantics of its application, as proposed here, is that, if the requisite kind of parameter does not exist, the interpreter should automatically introduce a new parameter which satisfies the constraint. Thus, in the case of Figure 4.3, the interpreter will introduce a new parameter, `store_room_1`, to satisfy the constraint in Lines 6–8 and will make its `number_of_contents` field have the value 10.

Those who are familiar with the Free Logic underpinnings of Galileo will remember that the `exists` predicate is used to require the presence of a specific parameter in a constraint network or, if it is absent, to enforce its introduction into the network. Applying the `!` operator to the `exists` *quantifier* is subtly different. It is similar in that it *does* require the presence of a parameter or force its introduction. However, it does not require the presence of a *specific* parameter; instead, it requires the presence of a parameter of particular type which must also have certain characteristics. Consider, for example, the program presented in Figure 4.4.

```
1 domain colour =::= { red, green, blue }.

2 exists( my_colour : colour ) and my_colour = green.

3 !exists colour(C):
4     C = blue.
```

Figure 4.4: Existance: predication versus quantification.

In Line 2 the existance of a specific parameter, called `my_colour`, is enforced;

it is specified that this parameter must be of type `colour` and must have the value `green`. In Lines 3–4 the existance of *some* parameter is required; it is specified that this parameter must also be of type `colour` and that it must have the value `blue`. This second constraint would cause the introduction of a new parameter into the network. However, the name of the parameter would be chosen by the Galileo run-time system.

Contrast the above program with that presented in Figure 4.5. Line 2 will behave exactly the same as Line 2 in Figure 4.4: it will cause the introduction of a `colour` parameter called `my_colour` and will assign it the value `green`. However, Lines 3–4 will not cause the introduction of any parameter: the requirements specified by this constraint are satisfied by the parameter introduced by Line 2.

```
1 domain colour =::= { red, green, blue }.

2 exists( my_colour : colour ) and my_colour = green.

3 !exists colour(C):
4     C = green.
```

Figure 4.5: Existance: predication versus quantification.

Care must be exercised when applying the `!` operator to the `exists` quantifier. However, it offers considerable user-friendliness in terms of the resolution of constraint violations, since it can automatically introduce missing parameters into a constraint network. The introduction of this usage of the `!` operator has been discussed with the developer of Galileo and its contribution to the language has been recognised [20]. While the inspiration for this new application of the `!` operator has come from the conceptual design domain, further study on its utility is required in order to ensure that it is suitable for inclusion in the standard version of the Galileo language. As will be seen later, the research reported in this dissertation provides significant motivation for the introduction of this new usage of the operator.

4.2.3 Other Extensions

As well as the extensions to quantification, a number of other minor extensions to the standard version of Galileo are assumed in this dissertation. The first is the introduction of a new keyword, "`hidden`". The second is the introduction of a new meta-level function called #parent_of.

```
domain box
    =::= ( length : positive real,
           width  : positive real,
           area   : positive real,
           hidden name : string ).
```

Figure 4.6: Example usage of the keyword "hidden".

The new keyword hidden is used to indicate which fields of a structured domain are visible on the user interface of the constraint filtering system. An example of its use is illustrated in Figure 4.6, where a structured domain called box is defined. This structured domain comprises a number of fields: length, width, area and name. The name field is annotated with the keyword hidden which indicates that this field will never be seen on the filtering system interface. This is a useful way of avoiding clutter on the filtering system interface by ensuring that uninteresting fields are not seen by the user. All fields not declared as hidden are, of course, visible to the user.

```
domain room
    =::= ( length  : positive real,
           breadth : positive real ).

domain house
    =::= ( owner   : string,
           kitchen : room,
           bedroom : room ).

relation has_the_same_owner_as( room, room )
    =::= { (R1,R2): #parent_of(R1).owner
                          = #parent_of(R2).owner }.
```

Figure 4.7: Program fragment showing a usage of the "#parent_of" function.

The standard version of Galileo already includes some meta-level relations and functions [21]. However, the meta-level aspects of the language are still under development and the range of relations and functions that currently exist is regarded as merely a first step in developing this aspect of the language. The research reported in this dissertation motivated the need for a new meta-level function, called #parent_of. This function can be used to access any field of a structured parameter from any one of its other fields. An example usage of this function is presented in Figure 4.7.

The three definitions presented in Figure 4.7 would, to have any effect, have to be part of some larger program. The relation `has_the_same_owner_as` (defined on Lines 8–10) establishes whether two rooms belong to the same person, by checking whether the rooms belong to house(s) which have the same owner. The `#parent_of` function is used to access the `house` of which a `room` is a part; once this has been accessed, the `owner` field of the `house` can be reached in the usual way.

4.3 Generic versus Application-Specific Concepts

It is believed that the approach to supporting conceptual design which is presented in this dissertation has wide applicability; it seems suitable for use in a wide variety of design domains. Consequently, its Galileo implementation should distinguish between those features which are generic to all appropriate applications and those features which are specific to individual design domains.

The generic aspects of the implementation are in Appendix B. Several applications of the approach have been investigated during the research. The implementations of two of them are presented in this dissertation: an implementation of an advice system for the conceptual design of "leisure vehicles" (bicycles and skateboards) is given in Appendix C; an implementation of a system for advising designers of electrical connectors is provided in Appendix D. Both of these application-specific implementations build on the implementation of the generic concepts.

The implementation of the generic concepts is discussed in Section 4.4. The way in which application-specific advice systems can be built on top of these generic concepts is illustrated in Section 4.5 by considering the implementation of an advice system for designing leisure vehicles.

4.4 Implementing Generic Concepts

As seen in Chapter 3, the development and comparative evaluation of schemes is the core task during the conceptual phase of design. Thus, in considering how to implement the generic concepts needed to support conceptual design, we will start by considering how to implement the generic notion of a scheme and will continue in a top-down fashion by considering how to implement the various concepts that are needed to support the notion of a scheme. Unless otherwise specified, all fragments of Galileo code which appear in the following discussion are borrowed from the Galileo module `generic_concepts.gal` which is presented in Section B.1 of Appendix B; the line numbers quoted in these fragments are those which the quoted lines have in the appendix.

4.4.1 A Generic Scheme Structure

The underlying premise of the approach to conceptual design presented in Chapter 3 is that every product exists to perform a particular function. The designer's task is to develop a scheme which embodies this required functionality. Since the Galileo language contained no model of a scheme, part of the research task was to design a suitable model of a scheme and any other concepts required to implement the design theory presented in Chapter 3.

```
46 domain scheme
47     =::= ( scheme_name : string,
48            structure   : embodiment ).
```

Figure 4.8: The representation of a generic scheme.

In Figure 4.8 the Galileo model of a generic scheme is presented. This shows that the concept of a scheme is implemented as a Galileo structured domain called `scheme` which has two fields, called `scheme_name` and `structure`, respectively.

The `scheme_name` is of type string and is used to uniquely identify a scheme. Thus, no two schemes should have the same name, a requirement which is specified by the constraint shown in Figure 4.9.

```
51 alldif scheme(S1), scheme(S2):
52     not ( S1.scheme_name = S2.scheme_name ).
```

Figure 4.9: All scheme names must be unique.

Since a scheme exists solely to provide the functionality required in the design specification, its structure should be the embodiment of that functionality. This is reflected by Line 48 in Figure 4.8, where the `structure` field of a `scheme` is declared to be of type `embodiment`.

4.4.2 A Generic Model of Function Embodiment

As seen in Chapter 3, a large part of the designer's task in conceptual design consists of identifying, from among the technologies allowed by the company's design policy, ways of providing required functionality; this required functionality may be either the "top-level" functionality specified in the design specification or lower-level functionalities which are identified as necessary during scheme development. In other words, the designer is mostly concerned with producing *embodiments* for *intended functions* by *choosing*, from among the *known means*, those which will provide the required functionality. This is reflected in Figure 4.10 which presents the Galileo implementation of an `embodiment` as a structured domain which has four fields: `scheme_name`, `intended_function`, `chosen_means` and `reasons`.

```
2 domain embodiment
3     =::= ( hidden scheme_name : string,
4            intended_function  : func,
5            chosen_means       : known_means,
6            reasons            : set of func_id ).
```

Figure 4.10: Modelling the embodiment of a function.

The `scheme_name` field, of type `string`, cross-references an embodiment to the scheme to which it belongs. At first glance, this form of cross-reference may seem unnecessary, since the only `embodiment` we have encountered so far in this discussion of implementation is the `structure` field of a `scheme`. However, as seen in Chapter 3, when principles are used to provide functionality they cause the introduction of further embodiments. These "derived" embodiments also need to be cross-referenced to their parent schemes and the method chosen to do this was to make each embodiment quote the name of its parent scheme. Since this type of cross-reference is of no interest to the user (although an essential housekeeping task for the computer), we do not want it to appear on the user interface; thus, it is labelled, in Line 3, as `hidden`.

The field `intended_function` represents the function which is to be provided by the embodiment; in Line 4 it is declared to be of type `func`. We will now consider the definition of type `func` in some detail, because when we have done so it will be easier to explain the rest of the `embodiment` definition.

First, it should be noted that the same type of functionality is frequently needed in different parts of a scheme; that is, the function to be provided by one embodiment may be the same type of function as that to be provided by a different embodiment in the same scheme (or, indeed, by an embodiment in a different scheme). Thus, a `func` must represent, not a function, but an *instance* of a function. Furthermore, of course, one function instance must be distinguishable from a different instance of the same function. Having noted this, consider the definition of a `func` provided in Figure 4.11.

```
17 domain func
18     =::= ( verb : string,
19            noun : string,
20            id   : func_id ).

21 all func(F):
22     has_a_unique_id(F).

23 domain func_id
24     =::= { I: nonnegative integer(I) }.
```

Figure 4.11: Modelling a function instance in Galileo.

In the conceptual design theory presented in Chapter 3, the approach to representing functionality is a symbolic one, consisting of representing a function by a verb-noun pair. As can be seen in Figure 4.11, this approach is implemented in the first two fields of the structured domain used to represent a `func`. Since a `func` is a function *instance*, it must contain some field which distinguishes it from other instances of the same function. On Line 20 in Figure 4.11, it can be see that the approach used was to give each `func` an `id` field, of type `func_id`; from Lines 23–24 it can be seen that a `func_id` is simply a synonym for a `nonnegative integer`.

The requirement that each function instance be distinguishable from all other function instances, including all other instances of the same function, is implemented by the constraint in Lines 21–22 of Figure 4.11. This constraint uses a relation called `has_a_unique_id` whose implementation is provided in Figure 4.12.

```
relation has_a_unique_id( func )
    =::= { F: exists nonnegative integer(I): I = F.id and
         contains_only_one_func_with_the_id( #parent_of(F).scheme_name, I) }.
```

Figure 4.12: Uniqueness for identifiers of function instances.

Examining Figure 4.12, we see that, for a `func`, `F`, to have a unique (and well-defined) `id`, the `id` must satisfy two conditions. First, the `id` of `F` must be a non-negative integer. Second, `F` must be the only `func` within its parent scheme which has this non-negative integer as its `id`. This second condition is implemented by using a relation called `contains_only_one_func_with_the_id`; the details of this relation are beyond the scope of this discussion but can, of course, be traced by the determined reader – it is defined in Lines 66–70 of Section B.1 in Appendix B.

For the purposes of this chapter, it is sufficient to note that a `func` in one scheme may use the same integer for its `id` as a `func` in a second scheme; this is because a `func` is uniquely identified by a combination of two things: its own `id` and the `scheme_name` field of the `embodiment` in which the `func` is used. This explains Line 79 in Figure 4.12: if a `func`, `F`, is to have a unique `id`, the scheme whose name is in the `scheme_name` field of the data object which is the `#parent_of F` (this parent will be an `embodiment`) must contain only one `func` which uses as its `id` the integer that is used by `F` as its `id`. The reader may wonder why this approach was used: why permit a function instance in one scheme to have the same identity number as a function instance in a different scheme? The decision to allow this was motivated by concern about the user interface. If every function instance were to have an identity number which is unique across all schemes, the designer of the first scheme would not be affected. However, the designer of every subsequently conceived scheme would find that the first function instance in her scheme would be allocated as its `id` some integer whose value depended on how many function instances were used in all previously conceived schemes – she might learn to accept this but would probably find it somewhat confusing. Instead, as we will see later, the first function instance in each scheme will have the `id` 0, the second 1, and so on.

It is worth noting one more point about `func`s before we return to complete our consideration of `embodiment`s. In Figure 4.13 a relation called `provides_the_function` is defined. This relation, which maps between a `func` and the function of which it is an instance, takes three arguments. The first argument is of type `func` while the others are of type `string`; the relation is true if the `func` in the first argument is an instance of the function which is described symbolically by the `verb-noun` pair in the second and third arguments. This relation is introduced here because it is a generic concept, although we will not see it being used until we see specific functions being introduced in Section 4.5.

```
relation provides_the_function( func, string, string )
    =::= { (F,V,N): V = F.verb and N = F.noun }.
```

Figure 4.13: Relation between a verb-noun pair and an instance of the function which they describe.

After this digression into the implementation of a `func`, let us return to the remaining part of the definition of an `embodiment`, which was presented in Figure 4.10. The third field in this structured domain is `chosen_means`. This represents the approach chosen by the designer to provide the `intended_function` for the `embodiment`. The approach chosen by the designer must conform to the technology policy adopted by the company for which she works: in other words, the *means* she chooses must be *known* and approved. This is reflected by the fact that in Line 5 of Figure 4.10 the field `chosen_means` must be of type `known_means`.

The repertoire of technologies known to, and approved by, a company varies, of course, from one company to another. Thus, the definition of `known_means` is not generic; it depends on the design domain to which the conceptual design advice system is being applied. Although we cannot, therefore, consider it further in this section, we will see an example definition of `known_means` in Section 4.5.

Before proceeding to discuss the final field in an `embodiment`, consider the constraint shown in Figure 4.14. This specifies that whatever `known_means` is chosen for an embodiment must, in fact, be capable of providing the function intended for the embodiment.

```
all embodiment(E):
    can_be_used_to_provide(E.chosen_means, E.intended_function).
```

Figure 4.14: The means chosen for an embodiment must be serviceable

The relation `can_be_used_to_provide` used Figure 4.14 is defined in Figure 4.15. As will be seen when we consider the definition of company-specific knowledge in Section 4.5, a `known_means` may be able to provide several functions simultaneously. Furthermore, a `known_means` may, if used in different ways, be able to provide different sets of functions simultaneously. The definition of the relation `can_be_used_to_provide` states that a `known_means` can provide a function if that function appears in some set of functions that the means can simultaneously provide. The relation `can_simultaneously_provide` used in this definition cannot be discussed here: it is company-specific knowledge and the way in which it is defined will be discussed in Section 4.5.

```
relation can_be_used_to_provide( known_means, func)
    =::= { (K,F): can_simultaneously_provide(K,Fs) and F in Fs}.
```

Figure 4.15: Definition of `can_be_used_to_provide`.

The final field in the definition of an `embodiment` (Figure 4.10) is called `reasons`. As discussed in Chapter 3, an embodiment may be introduced into a scheme because of different factors: it could be introduced to provide the top level functionality required in the design specification; alternatively, it could be introduced to provide some functionality whose necessity was recognised when some design principle was used during the development of the scheme. The `reasons` field in an `embodiment` records the motivation for introducing the `embodiment`. It does so by recording the identity numbers of the function instances whose provision required the introduction of the embodiment. This is reflected by the fact that, in Line 6 in Figure 4.10, the `reasons` field has the type `set of func_id`. The `reasons` field of an `embodiment` provides the basis for identifying those design entities between which context relations must be considered. The process of defining context relations between design entities will be discussed later in Section 4.5.4.

```
all scheme( S ):
    is_the_first_embodiment_in( S.structure, S.scheme_name ).
```

Figure 4.16: A constraint on the structure of the generic scheme representation.

Remember that, as we saw earlier, the `structure` field of a `scheme` is an `embodiment`. Since the `structure` of a `scheme` is intended to satisfy the top-level functionality requirement in the design specification, it will be *the first embodiment in* the `scheme`. This is reflected by the constraint shown in Figure 4.16.

```
relation is_the_first_embodiment_in( embodiment, string )
    =::= { (E,Sn): is_in_the_scheme( E, Sn ) and
                   E.intended_function.id = 0 and
                   E.reasons = {} }.
```

Figure 4.17: The meaning of the relation `is_the_first_embodiment_in`.

The definition of the relation `is_the_first_embodiment_in` that is used in this constraint is shown in Figure 4.17. The details of this definition are beyond the scope of this discussion. It is enough to note two points. First, as shown in Line 118 of Figure 4.17, the `id` of the `intended_function` for the first embodiment in a scheme must be zero; this reflects the fact that the functionality to be satisfied by the first embodiment in a scheme is the top-level functionality required in the design specification. Second, as shown on Line 119, the set of `reasons` for the first embodiment in a scheme is empty; this reflects the fact that, apart from the `intended_function` of the embodiment (which, of course, is the top-level functionality required in the design specification), no other function instances in the scheme motivated the introduction of this first embodiment.

4.4.3 A Generic Model of Means

As was mentioned above, the repertoire of technologies known to, and approved by, a company varies from one company to another. Thus, the issue of defining the specific means available to a designer is not something that will be covered in this section, which is concerned only with the definition of generic concepts. However, when specific means are defined (the method of doing so will be considered in Section 4.5), they will be defined in terms of a generic notion of means, whose definition we will consider now.

Figure 4.18 illustrates how the generic notion of a means can be modelled as a structured domain in Galileo.

```
35 domain means
36     =::= ( hidden scheme_name : string,
37            type               : means_type,
38            funcs_provided     : set of func_id ).

39 all means(M):
40     is_a_possible_behaviour_of( M.funcs_provided, M ).

41 domain means_type
42     =::= { a_principle, an_entity }.
```

Figure 4.18: Modelling a design means in Galileo

As shown in Lines 35–38, a means is implemented as a Galileo structured domain called `means`. This has three fields: `scheme_name`, `type` and `funcs_provided`. As was the case with the definition of an embodiment, the `scheme_name` field is used to cross-reference a `means` with the scheme in which the `means` is being used; similarly, for the same reasons as the `scheme_name` field was declared hidden in the definition of an embodiment, it is declared as `hidden` here.

It will be remembered from Chapter 3 that there are two kinds of means: principles and entities. This is reflected by the fact that, as shown in Figure 4.18, the domain from which the `type` field of a `means` takes its value contains only two possible values, `a_principle` and `an_entity` (see Lines 41–42).

The final field in the definition of the generic notion of a `means` is called `funcs_provided` and is of type `set of func_id`. It is used to remember which function instances within a scheme the `means` is being used to provide. Of course, a `means` should be used to provide only those function instances which it is capable of providing; this requirement is captured in the constraint in Lines 39–40. The definition of the relation `is_a_possible_behaviour_of` which is used in this constraint will not be considered here. It is presented in Lines 82–85 of `generic_concepts.gal` in Section B.1 but the details of its semantics are beyond the scope of this discussion; they can, of course, be traced by the determined reader. Essentially, however, the provision of a set of function instances can be regarded as a possible behaviour of a means if and only if the means can *simultaneously* provide the variety of function types as well as the quantity of instances of each type of function that is implied by the set of function instances.

4.4.4 A Generic Model of Design Principles

As discussed in Chapter 3, a design principle is an abstraction of a known approach to satisfying a functional requirement. A design principle describes how a particular type of functionality can be provided by embodying a set of lower-level functions. In addition, the principle may specify certain relationships, called *context relationships*, between the embodiments of these lower-level functions.

Having noted this, it can be seen that a specific design principle is a piece of intellectual property which, if not the patented property of a company, is central to the company's way of doing conceptual design. Thus, consideration of how to define specific principles will be delayed until Section 4.5. However, we will see in Section 4.5 that specific principles are implemented in terms of a generic notion of a principle, a concept which we will consider here.

```
domain principle
    =::= { P: means(P) and
              P.type = a_principle }.
```

Figure 4.19: A generic design principle model.

The generic notion of a principle is defined in Figure 4.19 as a specialisation of the generic notion of a `means`. Similarly, when we consider the definition of specific principles, in Section 4.5, we will see that these are defined as specialisations of this generic notion of a principle.

4.4.5 A Generic Model of Design Entities

A design entity represents a class of part, instances of which can be incorporated into a scheme to realize one or more of the function instances identified during the conceptual design of a product. As with design principles, specific design entities are part of a company's body of design expertise. As such, their representation will not be considered here, being delayed until Section 4.5. However, as with design principles, specific design entities are defined as specialisations of a generic notion of a design entity, a concept whose definition we will consider here.

```
domain entity
    =::= { E: means(E) and
              E.type = an_entity and
              exists ( E.id : entity_id ) }.

all entity(E):
    has_a_unique_id(E).

domain entity_id
    =::= { I: positive integer(I) }.
```

Figure 4.20: The model of a generic design entity.

In Figure 4.20 the generic notion of a design `entity` is defined, like the generic notion of a design principle, as a specialisation of the concept of a `means`. A strict approach to implementing computer-based support for conceptual design might distinguish between design entities and *design entity instances*. However, for our purposes here, we can avoid making this distinction. Consequently, the generic notion called an `entity` in Figure 4.20 actually represents a design entity *instance*. As such, it requires an identity field, which is provided by specifying (in Line 12) that it contains a field additional to those present in the generic notion of a `means`. This additional field, the `id` field, is of type `entity_id` which, as can be seen in Lines 15–16, is merely a synonym for a positive integer[2].

Each design entity instance should have a unique `id`, a requirement which is specified by the constraint in Lines 13–14. The meaning of the relation `has_a_unique_id` which is used in this constraint is given in Figure 4.21. It will not be explained here, since `id` uniqueness with respect to design entity instances is similar to `id` uniqueness with respect to function instances – in both cases, `id` uniqueness is local to the parent scheme; the explanation of `func_id` uniqueness in Figure 4.12 can be applied, *mutatis mutandis*, to Figure 4.21.

```
relation has_a_unique_id( entity )
    =::= { E: exists positive integer(I): I = E.id and
               contains_only_one_entity_with_the_id( E.scheme_name, I) }.
```

Figure 4.21: Uniqueness for entity identifiers.

4.4.6 Context Relationships and Entity Interfaces

As seen earlier, a design principle introduces a set of embodiments and a set of context relationships between them. Eventually, of course, each embodiment is realized by the introduction of design entity instances. This means that context relationships between embodiments will have to be realized by interfacing appropriately the entity instances which realize the embodiments.

In conceptual design, the minute detail of how a pair of components in a product are interfaced or connected is generally not of interest. However, there are certain minimum details which must be known in order to evaluate the quality of the scheme being developed. We will see in Section 4.5.4 how these details are represented. They will be represented as specialisations of a generic concept called an `interface`, whose definition we will consider here; it is provided in Figure 4.22.

[2]It will be remembered that a `func_id` was defined to be a non-negative integer. This is because the top-level functionality requirement in a scheme, which comes from the design specification, is given the special `func_id` of 0. Since no entity instance is treated as special, there is no need to allow 0 as a `entity_id`; thus, an `entity_id` can only be positive integer.

```
25 domain interface
26     =::= ( hidden scheme_name : string,
27            entity_1 : entity_id,
28            entity_2 : entity_id ).

29 all interface(I):
30     exists entity(E1), entity(E2):
31         I.entity_1 = E1.id and
32         I.entity_2 = E2.id and
33         is_in_the_same_scheme_as( I, E1 ) and
34         is_in_the_same_scheme_as( I, E2 ).
```

Figure 4.22: Modelling generic interfaces between design entities.

As can be seen in Lines 25–28, an `interface` is represented as a Galileo structured domain with three fields. The first field, `scheme_name`, is `hidden` from the user because it is present merely to facilitate system housekeeping; it cross-references the `interface` to the `scheme` in which it is used. The remaining two fields, `entity_1` and `entity_2`, contain the identity numbers of the two design entity instances which are being interfaced.

The constraint defined in Lines 29–34 ensures that, for every `interface` which is defined, both of the entity instances to which it relates exist in the same scheme as the interface. An `entity` instance is, as we have already seen, merely a specialisation of a `means` so the relation `is_in_the_same_scheme_as` used in the above constraint is the one defined in Figure 4.23, which is defined over an `interface` and a `means`. From this definition, we can see that an `interface` and a `means` are in the same scheme if the contents of their `scheme_name` fields are the same. Examination of Section B.1 in Appendix B will reveal that, although there are several different relations called `is_in_the_same_scheme_as`, defined between different pairs of concepts, they all rely on equality between `scheme_name` fields; we will not consider them further here.

```
relation is_in_the_same_scheme_as( interface, means )
    =::= { (I,M) : I.scheme_name = M.scheme_name }.
```

Figure 4.23: One of the several versions of the relation `is_in_the_same_scheme_as`.

A similar relation, which will be considered a little further here because it will be referenced later, is the relation `is_a_part_of` between a `means` (that is, either a principle or an entity) and a `scheme`. This relation's definition is shown in Figure 4.24

where it can be seen that a `means` is part of a `scheme` if the `scheme_name` field of the means contains the name of the scheme.

```
relation is_a_part_of( means, scheme )
    =::= { (M,S) : M.scheme_name = S.scheme_name }.
```

Figure 4.24: The relation `is_a_part_of`.

4.4.7 Generic Concepts for Comparing Schemes

Every scheme for a product must, of course, provide the functionality required by the design specification. However, as we have seen in Chapter 3, a design specification may, in addition to stipulating functionality, describe certain desirable, although not essential, properties of a design. These desirable properties are called *preferences*.

A design specification may include several preferences; the approach to be used for comparing two schemes when more than one preference is to be considered is described in Chapter 3. The implementation of that approach, or at least its generic aspects, can be found in Section B.2 of Appendix B, where a Galileo module called `comparison.gal` is presented. In what follows, unless otherwise noted, all Galileo code fragments come from this module.

```
domain intention
    =::= { minimal, maximal }.

domain preference
    =::= ( value  : real,
           intent : intention ).
```

Figure 4.25: Modelling a design preference.

The basic notion in the approach for comparing schemes is the *preference*. It is defined in Figure 4.25 as a structured domain containing two fields: the `value` field, which contains the value of whatever scheme property is the subject of the preference, and the `intent` field, which indicates whether it is preferred that this scheme property be minimised or maximised.

When two schemes are being compared, this will involve comparing how well they do in respect of each preference given in the design specification. A relation called `better_than` is used for comparing the instantiation of a preference in one scheme

```
relation better_than( preference, preference )
    =::= { (P1,P2): P1.intent = minimal and
                    P2.intent = minimal and P1.value < P2.value,
           (P1,P2): P1.intent = maximal and
                    P2.intent = maximal and P1.value > P2.value }.
```

Figure 4.26: Comparing two design preferences to determine which is better.

with the instantiation of the same preference in the other scheme. The definition of this relation is given in Figure 4.26. We can see that, if a preference involves minimising some property, the better instantiation of the preference is the one with the smaller value; similarly, if a preference involves maximising some property, the better instantiation of the preference is the one with the larger value. This notion of `better_than` was introduced in Section 2.3 on Page 31.

As explained in Chapter 3, no scheme for a product should dominate (in the sense of Pareto optimality) another scheme. This requirement is implemented as the constraint in Figure 4.27. If a designer develops a scheme which is dominated by a scheme that she (or one of her colleagues) has previously developed, this constraint will be violated and she will be given a message to that effect. Similarly, if she develops a scheme which dominates a previously developed scheme the constraint will be violated. In either case, it is intended that, as a result of the violation message, (one of) the designer(s) will be motivated to improve the inferior scheme or else to discard it.

The constraint in Figure 4.27 is defined in terms of a relation called `dominates` which is also defined in Figure 4.27. We can see that one scheme dominates another scheme if the first scheme `improves_on` the latter (in respect of some preference) while at the same time it is not true that the latter scheme `improves_on` the first in respect of any preference; this implements the ideas introduced in Section 2.3 on Page 31.

```
alldif scheme(S1), scheme(S2):
  not dominates( S1, S2 ).

relation dominates( scheme, scheme)
    =::= { (S1,S2): improves_on( S1, S2) and
                    not improves_on( S2, S1 ) }.
```

Figure 4.27: Comparing two schemes.

The relation `improves_on`, between two schemes, is defined in terms of the relation `better_than`, between instantiations of preferences. However, this definition is not generic; it will vary from one design specification to another and, thus, will not be discussed here; it will be considered later, in Section 4.6.3, when the representation of design specifications is discussed.

4.5 Implementing Application-Specific Concepts

The generic concepts introduced in Section 4.4 merely serve as the basis for defining the company- and project-specific concepts that are needed to support conceptual design. The definition of these application-specific concepts in terms of the generic concepts will now be considered.

To illustrate how this is done, we will use extracts from a design knowledge-base for a company called "Raleigh Leisure Vehicles Limited". This knowledge base, which is in a module called `raleigh_knowledge.gal`, is concerned with the design of products such as bicycles and skateboards; the module is presented in Section C.1 of Appendix C. In what follows, unless otherwise noted, each Galileo fragment quoted comes from this module.

4.5.1 Defining Known Means

As mentioned earlier, an important part of the company-specific knowledge that must be defined is the set of technologies that are available for use by the company's designer(s).

To define this knowledge we must list the `known_means`, that is the design principles and design entities that are approved for use by designers working for the company. We must specify what functionality is offered by these `known_means`. We must also describe each of these principles and entities in detail.

Listing the `known_means` is simply done. It merely involves defining a Galileo scalar domain which contains one symbol for each of the available principles or entities. In Figure 4.28 the collection of means which are available to designers working for Raleigh Leisure Vehicles Limited is presented.

As can be seen, the means available include `an_axle`, `a_bicycle` and `a_skateboard` and so on; as we shall see later, some of these refer to design principles while others

```
domain known_means
    =::= { an_axle, a_bicycle, a_skateboard, a_wheel_assembly,
           a_saddle, a_chain, a_chassis, a_steering_assembly,
           an_engine, a_harness, an_air_cushion, a_frame,
           a_handlebar_assembly, a_molded_frame,
           a_pedal_assembly, a_chain }.
```

Figure 4.28: A domain of known means available to designers.

refer to design entities.

As well as listing the `known_means` that are available to designers working for a company, the company knowledge-base must specify the functions that each `known_means` can provide. This is done by declaring the relation called `can_simultaneously_provide`. This company-specific relation was invoked by the generic definition which we saw earlier in Figure 4.15 but could not be considered at that stage, because we were considering only generic concepts.

Figure 4.29 provides the definition of this relation specific to Raleigh Leisure Vehicles Limited. It provides the information on means functionality that was shown in Figure 3.13. The relation shows the functionality that can be simultaneously provided by the various `known_means` available to engineers working for this company. Each pair in this binary relation associates a `known_means` with a set of functions. In many cases, this set of functions is simply a singleton set; however, the notion of a set is used in the definition of this relation because, in general, a `known_means` could provide multiple function instances at the same time. In Lines 133–134, it can be seen that `an_air_cushion` provides only one function, namely `facilitate movement`. In Lines 135–139 it can be seen that `an_axle` can provide the two functions *support wheel* and *facilitate rotation* simultaneously, but can also be used to provide the single function *punch holes*.

```
relation can_simultaneously_provide( known_means, set of func )
    =::= { (an_air_cushion,{F}):
                provides_the_function( F, 'facilitate', 'movement' ),
            (an_axle,{F}):
                provides_the_function( F, 'punch', 'holes' ),
            (an_axle,{F1,F2}):
                provides_the_function( F1, 'support', 'wheel' ) and
                provides_the_function( F2, 'facilitate', 'rotation' ),
            (a_bicycle,{F}):
                provides_the_function( F, 'provide', 'transport' ),
            (a_chasis,{F}):
                provides_the_function( F, 'provide', 'support' ),
            (an_engine,{F}):
                provides_the_function( F, 'provide', 'energy' ),
            (a_frame,{F}):
                provides_the_function( F, 'provide', 'support' ),
            (a_handlebar_assembly,{F}):
                provides_the_function( F, 'change', 'direction' ),
            (a_molded_frame,{F1,F2}):
                provides_the_function( F1, 'provide', 'support' ) and
                provides_the_function( F2, 'support', 'passenger' ),
            (a_harness,{F}):
                provides_the_function( F, 'support', 'passenger' ),
            (a_pedal_assembly,{F}):
                provides_the_function( F, 'provide', 'energy' ),
            (a_saddle,{F}):
                provides_the_function( F, 'support', 'passenger' ),
            (a_skateboard,{F}):
                provides_the_function( F, 'provide', 'transport' ),
            (a_steering_assembly,{F}):
                provides_the_function( F, 'change', 'direction' ),
            (a_wheel_assembly,{F}):
                provides_the_function( F, 'facilitate', 'movement' ) }.
```

Figure 4.29: Relating means to the functions that they can provide.

In the cases discussed here the function instances that a `known_means` can provide simultaneously are instances of different functions. In general, however, two or more function instances provided by a `known_means` could be instances of the same function. For example, in Figure 4.30 two instances of the function *provide light* are being provided by the same design entity.

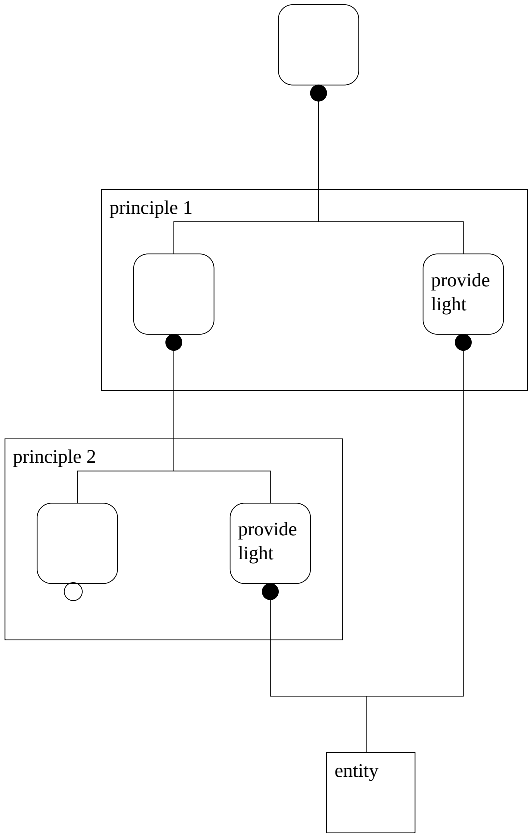


Figure 4.30: Embodying two instances of the function *provide light* with the same design entity.

4.5.2 Defining Company-specific Design Principles

In Figure 4.19 the notion of a generic design principle was presented. All the design principles that are available to designers working for specific company can be defined as specialisations of this generic design principle. Consider, for example, the principle of a bicycle which was seen in Figure 3.9. It is defined in Figure 4.31.

This application-specific principle is defined to be a specialisation of the generic notion of a principle (Line 8); the specialisation is specified by the extra properties that are defined in Lines 9–28.

It was seen in Figure 3.9 that a `bicycle` principle involves five `embodiments`. These are specified in Lines 9–13 of Figure 4.31. The functions which Figure 3.9 states are to be provided by these embodiments (`facilitate movement`, `provide energy`, `support passenger`, `change direction` and `provide support`) are specified in Lines 14–23 of Figure 4.31, using the generic relation `provides_the_function` which we saw defined earlier in Figure 4.13.

The context relationships between the embodiments which are shown in Figure 3.9

```
7  domain bicycle
8      =::= { B: principle(B) and
9             exists( B.e1 : embodiment ) and
10            exists( B.e2 : embodiment ) and
11            exists( B.e3 : embodiment ) and
12            exists( B.e4 : embodiment ) and
13            exists( B.e5 : embodiment ) and
14            provides_the_function( B.e1.intended_function,
15                             'facilitate', 'movement' ) and
16            provides_the_function( B.e2.intended_function,
17                             'provide', 'energy' ) and
18            provides_the_function( B.e3.intended_function,
19                             'support', 'passenger' ) and
20            provides_the_function( B.e4.intended_function,
21                             'change', 'direction' ) and
22            provides_the_function( B.e5.intended_function,
23                             'provide', 'support' ) and
24            drives( B.e2, B.e1 ) and
25            supports( B.e5, B.e1) and
26            supports( B.e5, B.e2) and
27            supports( B.e5, B.e3) and
28            supports( B.e5, B.e4) }.
```

Figure 4.31: Definition of a company-specific design principle.

are stated in Lines 24–28 of Figure 4.31. In Line 24 it is stated that a `drives` relationship must exist between the `embodiment e2` and `embodiment e1`. Line 25 states that a `supports` relationship must exist between the `embodiment e5` and `embodiment e1`. Line 26 requires that a `supports` relationship must exist between the `embodiment e5` and `embodiment e2`. Line 27 states that a `supports` relationship must exist between the `embodiment e5` and `embodiment e3`, while Line 28 requires that a `supports` relationship must exist between the `embodiment e5` and `embodiment e4`.

Ultimately, as was seen in Chapter 3, each embodiment introduced by a principle is realized by the introduction of a set of one or more design entity instances. Thus, if a design principle specifies a context relationship between some of its embodiments, that relationship will, ultimately, be realized by some analogous relationship between the sets of design entity instances which realize the embodiments. We will see later, in Section 4.5.4, how these relationships are defined.

4.5.3 Defining Company-specific Design Entities

Design entities represent classes of parts which provide physical realizations of the functions that are identified in the function decomposition of the product being designed. Company-specific design entities are defined as specialisations of the generic notion of a design entity which was presented in Section 4.4.5.

In every company there are particular properties of parts that are of general interest. For example, in the mechanical domain, properties such as mass, geometry and material may be considered important enough to be represented as attributes of all design entities. In such cases, a company will define its own "pseudo-generic" type of design entity as a specialisation, containing fields to represent these important properties, of the generic design entity. It will also define functions to compute important properties of these pseudo-generic design entities.

For example, Raleigh Leisure Vehicles Limited considers the *width*, *mass* and *material* to be important properties of all parts. Thus, as shown in Figure 4.32, it was considered appropriate to define the notion of a `raleigh_entity` which has these properties.

```
56 domain raleigh_entity
57     =::= { R: entity(R) and
58               exists( R.width    : real ) and
59               exists( R.mass     : real ) and
60               exists( R.material : raleigh_material ) and
61               R.mass = mass_of( R ) }.


67 domain raleigh_material
68     =::= { cfrp, titanium, aluminium, steel }.


236 function mass_of( raleigh_entity ) -> real
237     =::= { E -> 2:  E.material = cfrp,
238            E -> 3:  E.material = titanium,
239            E -> 5:  E.material = aluminium,
240            E -> 10: E.material = steel }.
```

Figure 4.32: The implementation of a company-specific design entity called `raleigh_entity`.

As can be seen in Lines 56–61, the concept of a `raleigh_entity` is defined to be a specialisation of the generic notion of an `entity`. The specialisation consists of an additional three fields, representing `width`, `mass` and `material`, with an equational constraint for estimating the `mass`. The `width` and `mass` fields are of type `real` but the `material` field is of type `raleigh_material`; according to the definition of the type `raleigh_material`, in Lines 67–68, this company's designers are restricted to using one of just four materials: `cfrp`, `titanium`, `aluminium` and `steel`.

In Line 61 the `mass` of a `raleigh_entity` is estimated using a function called `mass_of`. The implementation of this function is presented in Lines 236–240. In a real situation, the mass of an entity instance would, of course, depend on its volume as well as the constituent material; however, in order to simplify matters for the sake of this example, it is assumed that the `mass_of` all instances of `raleigh_entity` can be estimated by knowing just the `material` of the entity. Thus, for example, the `mass_of` a `raleigh_entity` whose `material` is `steel` is estimated to be 10 units (Line 240).

```
29 domain chassis
30     =::= { C: raleigh_entity(C) }.
```

Figure 4.33: A `chassis` design entity.

When a company has defined its own pseudo-generic concept of a design entity, it can define a repertoire of company-specific design entities. These company-specific design entities may be specialisations, with further additional fields, of the company's pseudo-generic concept of design entity or they may be merely synonyms of it. For example, in Figure 4.33 a design entity called `chassis` is implemented. This design entity is simply a synonym for `raleigh_entity` because, apparently, for the purposes of conceptual design, no attribute of a `chassis` is considered important, besides those that are already defined for a `raleigh_entity`.

A company-specific design knowledge-base, such as the one presented in Section C.1, will usually contain many different types of design entity that are available to designers for use in their schemes. An inspection of Section C.1 will show that, as well as `chassis`, it defines the following types of `raleigh_entity`: `air_cushion`, `axle`, `chain`, `engine`, `frame`, `handlebar_assembly`, `harness`, `molded_frame`, `pedal_assembly`, `saddle`, `steering_assembly`, `wheel_assembly`.

4.5.4 Defining Company-Specific Context Relationships

If a design principle specifies a context relationship between some of its embodiments, that relationship must, ultimately, be realized by some analogous relationship between the sets of design entity instances which realize the embodiments. We will now see how that is specified.

Consider, for example, the `bicycle` principle defined in Figure 4.31. Line 24 specifies that a `drives` relationship must hold between the first two embodiments introduced by the principle, those which embody the functions `provide energy` and `facilitate movement`.

The `drives` relationship between two embodiments is defined in Figure 4.34. This definition specifies that the `drives` relationship holds between two embodiments if and only if an analogous relationship, also called `drives`, holds between the two sets of entity instances that derive from these embodiments[3].

```
relation drives( embodiment, embodiment )
    =::= { (E1,E2): drives( { X | exists entity( X ):
                                    derives_from( X, E1 ) },
                             { Y | exists entity( Y ):
                                    derives_from( Y, E2 ) } ) }.
```

Figure 4.34: The meaning of the `drives` context relation between embodiments.

The analogous relationship between the sets of entity instances that derive from the embodiments involved in a context relationship must be defined. The `drives` relationship between sets of derived entity instances is shown in Figure 4.35. It is defined in terms of yet another analogous relationship, this time between individual entity instances: apparently, one set of entity instances drives another set of entity instances if there exists one individual entity instance in the first set which `drives` an individual entity instance in the second set.

[3]The relation `derives_from` used in this definition is a generic relation and, as such, is defined in Section B.1 of Appendix B. A detailed consideration of its definition would distract us from the focus of the current discussion. Suffice it to say that if a determined reader were to follow the definition she would discover that an entity instance can derive either directly or indirectly from an embodiment. An entity instance derives directly from an embodiment if it is used, without recourse to any design principle, as the means of implementing the function intended for the embodiment, while an entity instance derives indirectly from an embodiment if one or more principles were used in the reasoning that led to the entity instance being used to implement part of the function intended for the embodiment. The reasoning that leads to a particular `embodiment` being introduced into a scheme can be determined from its `reasons` field.

```
relation drives( set of entity, set of entity )
    =::= { (E1s,E2s): exists E1 in E1s, E2 in E2s:
                            drives( E1, E2 ) }.
```

Figure 4.35: The meaning of the drives context relation between sets of design entity instances.

The precise realization of the context relationship specified in a principle depends on which design entities are used to realize the embodiments that must satisfy the context relationship. Suppose that a `pedal_assembly` is the design entity used to `provide energy` and a `wheel_assembly` is the design entity used to `facilitate movement`. We can see in Figure 4.36 the relationship that would have to be satisfied between these two entity instances.

```
relation drives( entity, entity )
    =::= { (P,W): pedal_assembly(P) and wheel_assembly(W) and
                  is_in_the_same_scheme_as( P, W ) and
                  !exists chain(C):
                      is_in_the_same_scheme_as( P, C ) and
                      !exists mechanical_interface(M1):
                          M1.entity1 = P.id and
                          M1.entity2 = C.id and
                          M1.relationship = drives and
                      !exists mechanical_interface(M2):
                          M2.entity1 = W.id and
                          M2.entity2 = C.id and
                          M2.relationship = drives }.
```

Figure 4.36: The meaning of the drives context relation between a pedal assembly and a wheel assembly.

According to the definition of this relationship, if a `pedal_assembly` is to `drive` a `wheel_assembly`, they must be `in_the_same_scheme` and there must be a further design entity instance, a `chain`, in the same scheme. These entity instances must be interfaced in the following way: there must be a `mechanical_interface` between the `pedal_assembly` and the `chain` and another one between the `wheel_assembly` and the `chain`. This arrangement was used in Figure 3.17.

As we shall now see, a `mechanical_interface` is simply a specialisation of the generic notion of an `interface` which we encountered in Section 4.4.6. The definition of a `mechanical_interface` is given in Figure 4.37. It can be seen to be a specialisation of a company-specific notion of interface, called a `raleigh_interface`, which, from its definition in Figure 4.38, can be seen to be a specialisation of the generic notion of `interface`.

```
domain mechanical_interface
    =::= { S: raleigh_interface(S) and S.type = mechanical and
              exists( S.relationship : mechanical_relationship ) }.

domain mechanical_relationship
    =::= { controls, drives, supports }.
```

Figure 4.37: Modelling a `mechanical` interface.

It can be seen from Figure 4.37 that a `mechanical_interface` is a `raleigh_interface` whose `type` field contains the value `mechanical` and which also has an additional field called `relationship` that specifies the nature of the mechanical relationship involved in the interface; it can be seen that three kinds of relationship are supported: `controls`, `drives` and `supports`.

It can be seen from Figure 4.38 that a `raleigh_interface` is simply an `interface` with an additional field called `type` which specifies the class of relationship involved in the interface; it can be seen that two classes of relationship are supported: `spatial` and `mechanical`.

```
domain raleigh_interface
    =::= { I: interface(I) and
              exists( I.type : raleigh_interface_type ) }.

domain raleigh_interface_type
    =::= { spatial, mechanical }.
```

Figure 4.38: Modelling company-specific interfaces.

4.6 Representing Design Specifications

In Chapter 3 a detailed discussion on the contents of a design specification was presented. A constraint-based approach to modelling a design specification will be presented here.

A design specification comprises a set of requirements which define the functional and physical properties of the product to be designed. During the conceptual phase of design, a designer (or, possibly, a team of them) generates a set of alternative schemes which satisfy the requirements defined in the design specification. The designer(s) will select a subset of these schemes for further development during the later phases of the design process. Thus, a design specification can be regarded as an intensional specification of a set of schemes.

In Section 4.4.1, a generic model of a scheme was presented. This generic scheme provides a basis for the development of schemes for any product. Each requirement in the design specification can be regarded as a constraint on the schemes that the designer will develop. The requirements defined in the design specification can be modelled as constraints which are universally quantified over all instances of the scheme representation. For example, consider the following design specification:

> Design a product which exhibits the following properties:
>
> - provides the function *provide transport*;
> - is *recyclable*;
> - has a *width not greater than 2m*;
> - has *minimal mass* and
> - comprises a *minimal number of parts.*

In this specification there is one functional requirement and four physical requirements. The functional requirement states that the product must provide the function *provide transport.* The physical requirements state that the product must be *recyclable*, must have a *width not greater than 2m*, should have *minimal mass* and should comprise a *minimal number of parts.*

In the following sections a constraint-based model of this design specification will be developed in stages. A full constraint-based implementation of this design specification is presented in a Galileo module called `vehicle_spec.gal` presented in Section C.2. Parts of this design specification model will be presented in the following sections in order to demonstrate the modelling approach used in this dissertation.

4.6.1 Modelling Functional Requirements

It was explained above that a design specification can be regarded as an intensional specification of a set of schemes. This means that a design specification can be represented in Galileo as a specialisation of the generic notion of a scheme. This is illustrated in Figure 4.39, where a fragment from the design specification in Section C.2 is given. The specification for a new vehicle is called a `vehicle_scheme` and, as can be seen, is a specialisation of a `scheme`.

```
domain vehicle_scheme
    =::= { S: scheme( S ) and
              ...
```

Figure 4.39: A design specification is a specialisation of the generic notion of a scheme.

In the design specification whose representation we are considering here, the functional requirement is that the product can *provide transport*. In terms of the modelling approach being presented here, this requirement can be treated as a constraint on the kind of scheme that is acceptable as a `vehicle_scheme`. This is shown in Figure 4.40.

```
domain vehicle_scheme
    =::= { S: scheme( S ) and
              provides_the_function( S.structure.intended_function,
                                     'provide', 'transport' ) and
              ...
```

Figure 4.40: Modelling the functional properties of a scheme

Recall, from Section 4.4.1, that the generic model of a `scheme` has a field called `structure` which is of type `embodiment`. The `intended_function` field of this embodiment represents the function that is to be provided by the scheme. In the fragment from the vehicle design specification that is given in Figure 4.40, it is stated that a `vehicle_scheme` is a `scheme` whose `structure` provides the `intended_function` *provide transport*.

4.6.2 Modelling Categorical Physical Requirements

The example design specification presented earlier contains several physical requirements. These relate to the recyclability, width and mass of the product and the number of parts in it. Two of these requirements, those referring to recyclability and width, are categorical; that is, they *must* be satisfied. The other two, those referring to the mass of the product and the number of parts in it are preferences; they merely indicate that the mass and number of parts should be as small as possible.

In this section, we will consider how to incorporate categorical physical requirements into the Galileo representation of a design specification. The incorporation of preferences will be discussed in Section 4.6.3.

The requirement that the product be *recyclable* is a life-cycle requirement; it can be easily represented as a constraint, as shown in Line 9 of Figure 4.41.

```
5  domain vehicle_scheme
6      =::= { S: scheme( S ) and
7                 provides_the_function( S.structure.intended_function,
8                                        'provide', 'transport' ) and
9                 recyclable(S) and
10                ...
```

Figure 4.41: Modelling the physical requirement `recyclable`

Of course, the meaning of the relation `recyclable` used here is company-specific and, as such, must be defined in the company-specific design knowledge-base. Suppose that a scheme's being `recyclable` means that all the individual entity instances in the scheme should be made of recyclable materials; this meaning is defined in Figure 4.42, which is taken from the Galileo module `raleigh_knowledge.gal` presented in Section C.1.

```
194 relation recyclable( scheme )
195     =::= { S: all entity(E): is_in_the_scheme(E,S) implies
196                   recyclable(E) }.

197 relation recyclable( entity )
198     =::= { E: recyclable_material( E.material ) }.

199 relation recyclable_material( raleigh_material )
200     =::= { cfrp, aluminium, steel }.
```

Figure 4.42: Modelling the physical requirement `recyclable`

A `scheme` is `recyclable` if all of the entities in the `scheme` are `recyclable` (Lines 194–196). An `entity` is `recyclable` if its `material` is a `recyclable_material` (Lines 197–198). The recyclable materials are `cfrp`, `aluminium` and `steel` (Lines 199–200). It will be recalled, from Figure 4.32, that, normally, four materials are available to designers working for this company: `cfrp`, `titanium`, `aluminium` and `steel`. For designers working on this project, however, `titanium` is not allowed, because it is not deemed recyclable.

The implementation of the relation `recyclable` in Figure 4.42 is quite straightforward. However, this is not always the case with physical design requirements. Some physical requirements can be quite complex to implement. However, it is generally true that, for a particular company, physical requirements relate to a known set of critical product properties; consequently, it would be worth the company's while to spend the effort to develop a representation for these requirements and incorporate them into the company-specific design knowledge-base. We will now consider the representation of a slightly more complex physical requirement: the product should have a width no greater than two metres.

```
5  domain vehicle_scheme
6      =::= { S: scheme( S ) and
7                provides_the_function( S.structure.intended_function,
8                                       ’provide’, ’transport’ ) and
9                recyclable(S) and
10               exists( S.width : real ) and
11               !S.width = width_of( S ) and
12               S.width =< 2.0 and
13               ...
```

Figure 4.43: Modelling the requirement which limits `width`.

This requirement implies several features that should be incorporated into the design specification. The requirement implies that there exists a `width` attribute for the scheme, that this attribute will have a numeric value and that its value should not exceed two metres. In Figure 4.43 the design specification fragment presented in Figure 4.41 is extended to incorporate this requirement.

In Line 10 of Figure 4.43, the existence in every `vehicle_scheme` of a field called `width`, of type `real`, is stipulated. The manner in which the width of the scheme is computed is specified in Line 11. The value of the `width` of the scheme will be computed using a function called `width_of`. In Line 11 the ! operator is applied to the `width` of the `vehicle_scheme` because, as and when the `width` of the `vehicle_scheme` changes when entities are introduced during scheme development, the Galileo filtering system will need to update the value of this attribute. This is done by the filtering system making a sequence of non-monotonic assumptions about the value of this attribute, essentially retracting its old value and asserting the updated value whenever necessary; this is the standard semantics of the ! operator in Galileo [23]. The actual requirement that the `width` of the scheme be no more than two metres is specified on Line 12.

```
244 function width_of( scheme ) -> real
245     =::= { S -> width_of( { E | exists entity(E):
246                      is_a_part_of( E, S ) } ) }.

247 function width_of( set of entity ) -> real
248     =::= { Es -> sum( { E.width | exists E in Es:
249                      not exists E2 in Es:
250                          overlaps( E2, E ) and
251                          E2.width > E1.width } ) }.
```

Figure 4.44: Specifying how to compute the `width_of` a scheme.

The function `width_of` used to estimate the width of a scheme is presented in Figure 4.44. The `width_of` a scheme is estimated as the width of the set of entities which make up the scheme (Line 244–246 of `raleigh_knowledge.gal`). The `width_of` a set of entities is estimated, in this case, as the sum of the widths of entities which are not overlapped by larger entities (Lines 247–251). A relation called `overlaps` is used to determine whether two entities overlap each other or not (Line 250).

```
relation overlaps( entity, entity )
    =::= { (E1,E2): is_in_the_same_scheme_as( E1, E2 ) and
                    !exists spatial_interface(S):
                        is_in_the_same_scheme_as( S, E1 ) and
                        S.entity_1 = E1.id and
                        S.entity_2 = E2.id and
                        S.relationship = above or
                        S.relationship = under }.
```

Figure 4.45: The meaning of the relation `overlaps`.

The meaning of the relation `overlaps` is defined in Figure 4.45. Two design entities are considered to overlap if there exists a `spatial_interface` between the entities which defines one entity as being either `above` or `under` the other. Obviously, more accurate estimates are possible; our purpose here is merely to illustrate the flavour of how complex physical requirements can be implemented.

4.6.3 Modelling Design Preferences

In the example design specification being discussed in this chapter there are two design preferences. These relate to the *mass* of the product being designed and the *number of parts* used in it; in each case, the preference is to have a minimal value. These preferences are incorporated into the design specification in Figure 4.46.

```
5  domain vehicle_scheme
6      =::= { S: scheme( S ) and
7                provides_the_function( S.structure.intended_function,
8                                       'provide', 'transport' ) and
9                recyclable(S) and
10               exists( S.width : real ) and
11               !S.width = width_of( S ) and
12               S.width =< 2.0 and
13               exists( S.mass  : preference ) and
14               S.mass.intent = minimal and
15               !S.mass.value  = mass_of( S ) and
16               exists( S.number_of_parts : preference ) and
17               S.number_of_parts.intent = minimal and
18               !S.number_of_parts.value  = number_of_parts_in( S ) }.
```

Figure 4.46: Incorporating design preferences in the design specification model.

In Lines 13–15 the design preference related to the scheme's `mass` is represented. Line 13 states that each scheme has a parameter called `mass` which is of type `preference`; Line 14 states that the `intent` of the preference is that the mass should be `minimal` while Line 15 states that the `value` of the `mass` is computed using a function called `mass_of`. Similarly, Lines 16–18 define a preference specifying that the `number_of_parts` in the scheme should be minimal and that the actual number of parts is computed using a function called `number_of_parts_in`.

```
233 function mass_of( scheme ) -> real
234     =::= { S -> sum( { mass_of( E ) | exists entity(E):
235                         is_a_part_of( E, S ) } ) }.
```

Figure 4.47: Computing the mass of a scheme.

In Figure 4.47 the details of how the `mass_of` of a scheme is computed is presented. In Lines 233–235 of `raleigh_knowledge.gal` a function called `mass_of` is defined which computes the `mass_of` of a `scheme`; it computes the mass of a scheme as the sum of the masses of the individual design entities which are part of the scheme. The meaning of the relation `is_a_part_of`, which is used here, was considered in Section 4.4.6. The mass of an entity is computed using the function defined in Lines 236–240 in Figure 4.32.

The second preference in the design specification states that the number of parts used in the scheme should be minimal. In Figure 4.48 the details of how the number of parts in a scheme is computed are presented: it is the cardinality of the set of design entities that are part of the particular scheme.

```
function number_of_parts_in( scheme ) -> integer
    =::= { S -> cardinality( { E | exists entity(E):
                    is_a_part_of( E, S ) } ) }.
```

Figure 4.48: Computing the number of parts used in a scheme

In Section 4.4.7, where the generic concepts for comparing schemes were considered, the notion of one scheme dominating another was defined in terms of a relation called `improves_on`. It was stated there that this relation is project-specific. We are now in a position to consider how this relation would be specified for the vehicle design application; it is defined in Figure 4.49.

```
relation has_better_mass_than( vehicle_scheme, vehicle_scheme )
    =::= { (S1,S2): better_than( S1.mass, S2.mass ) }.

relation has_better_number_of_parts_than( vehicle_scheme, vehicle_scheme )
    =::= { (S1,S2): better_than( S1.number_of_parts, S2.number_of_parts ) }.

relation improves_on( vehicle_scheme, vehicle_scheme )
    =::= { (S1,S2): has_better_mass_than( S1, S2 ) or
                    has_better_number_of_parts_than( S1, S2 ) }.
```

Figure 4.49: Determining when one scheme improves on another.

Apparently, one `vehicle_scheme`, `S1`, `improves_on` another, `S2`, if and only if `S1` either `has_better_mass_than` or `has_better_number_of_parts_than` `S2`. The relations `has_better_mass_than` and `has_better_number_of_parts_than` are both defined in terms of the generic relation `better_than`, between preference instantiations, that was defined in Section 4.4.7.

4.6.4 Modelling Design For X Requirements

While the exploitation of Design For X (DFX) concepts during conceptual design is not a key concern in the research presented here, the incorporation of DFX concepts into the modelling approach presented here is quite straightforward. Indeed, the implementation of the categorical life-cycle requirement, related to the recyclability of the product being designed, has already been discussed. However, it is worth mentioning explicitly how the approach being presented here provides a basis for incorporating arbitrary DFX guidelines into the model of a scheme.

The modelling approach presented here is capable of modelling both design entity instances and their configuration, via `interfaces`. Consequently, repertoires of generic, company-specific or project-specific DFX guidelines, which refer to design entities and/or their configurations, can be developed. For example, some DFX relationship between the attributes which define a design entity can be implemented as a constraint quantified over all instances of the relevant design entity. Similarly, a DFX relationship that should be satisfied by configurations of design entities can be readily implemented as a constraint which is quantified over combinations of instances of the relevant design entities and the interfaces between them. There is ample literature available which demonstrates how DFX requirements can be stated in Galileo [49, 151].

4.7 Scheme Generation

Once a constraint-based model of the design specification has been developed, the designer (or team of designers) has the task of developing a set of alternative schemes for the required product. The constraint-based model of the design specification contains constraints relating to the following issues:

- constraints based on the functional requirements of the product as stated in the design specification;
- constraints based on the categorical physical requirements of the product as stated in the design specification;
- constraints based on the preferences regarding the values of particular design properties; and
- constraints relating to any explicit DFX requirements which are either explicit in the design specification or implicit in the company's design policy.

From a design perspective, the designers' task is to develop a number of alternative schemes which satisfy the design specification. However, from a constraint-processing point of view, their task is to search for a set of schemes which satisfy the constraint-based design specification representation, each scheme resulting from making different choices among the various means for providing the required functionality. This process has already been discussed in Chapter 3.

The selection of means for providing the required functionality is subject to the various constraints in the design specification. For example, if a designer selects a means to embody a particular function which is not capable of providing the required functionality this violates the constraint shown in Figure 4.14

As the designer selects a means for providing a particular function, further constraints are introduced into the scheme being developed, since each means has an associated set of constraints defining its properties. In this way the constraint-based model of the design comprises constraints representing the requirements stated in the design specification as well as constraints on functionality, scheme structure and life-cycle issues.

The interaction between a designer and a constraint-based model of the scheme she is developing (as well as the models of whatever schemes have already been developed for the product being designed) are governed by the constraint-filtering behaviour of the Galileo run-time system. This will be illustrated in Chapter 5.

In the remainder of this section, we will consider some factors which govern the behaviour that will be exhibited by the constraint filtering system. Recall that, in Section 4.4.2, we stated that a designer is concerned with producing *embodiments* for *intended functions* by *choosing*, from among the *known means*, those which will provide the required functionality. Recall also that, in Section 4.5.1, the collection of known means available for use by a company's designers is defined (see Figure 4.28) as a scalar domain, called `known_means`, which contains a list of symbolic names for these known means. Further, recall that, in Section 4.4.3, we saw that each `means`, whether it be a principle or a design entity, is represented as a structured domain.

When a designer selects a value for the `chosen_means` field of an `embodiment` she is saying that there should exist an instance of a `means` which is in some way identified by the chosen `known_means`. The particular `means` that should exist should be determinable from the `known_means` which has been selected by the designer.

We now point out a convention which should be used by knowledge engineers when selecting symbolic names to appear in the scalar domain `known_means` and when selecting domain names for the various specialisations of the domain `means` which represent the different principles and design entities that are available for use by designers.

Each `known_means` which a designer may select should have a corresponding `means`. The `known_means` should always be the name of its associated `means` prefixed with either the substring "`a_`" or the substring "`an_`". The company-specific design knowledge-base `raleigh_knowledge.gal` presented in Section C.1 of Appendix C satisfies this knowledge engineering convention.

```
88 all embodiment(E): E.chosen_means = a_bicycle implies
89     !exists bicycle( B ):
90         must_be_directly_used_for( B, E.intended_function ) and
91         causes( E, B.e1 ) and
92         causes( E, B.e2 ) and
93         causes( E, B.e3 ) and
94         causes( E, B.e4 ) and
95         causes( E, B.e5 ).
```

Figure 4.50: Modelling the use of a design principle for an embodiment.

When a designer selects a particular known means to provide the intended function for an embodiment, some processing is needed to ensure that the effects of selecting the known means are propagated throughout the expanding constraint network which represents the evolving model of the design. The knowledge engineering convention described above should be followed because the actual introduction into the design (in fact, into the constraint network which represents the design) of parameters representing principles and means is triggered by constraints in the company-specific knowledge-base which rely on this convention. In other words, constraints relying on this convention are what actually cause the existence, and consequently the arrival onto the designer's user interface, of constraint network parameters representing the various principles and means she has selected. For example, consider the constraint shown in Figure 4.50, which is taken from `raleigh_knowledge.gal` in Section C.1. This shows the effect of selecting the principle of `a_bicycle` as the `known_means` for realising an `embodiment`. According to this constraint, when the designer decides to use `a_bicycle` as the `chosen_means` for an `embodiment E`, it must be true that there exists some instance, `B`, of the principle `bicycle` which `must_be_directly_used_for` the `intended_function` of `E` (Lines 88–90). If there already exists some instance of the principle and if this instance is free to be used for the intended function of `E`, then it will be so used. However, since this constraint applies the `!` operator to the `exists` quantifier, the effect of this constraint is that an instance of the principle `bicycle` will be created if one does not already exist or if all pre-existing instances of the principle are being used in other embodiments and are, therefore, unavailable for use in embodiment `E`. A further effect of this constraint is that it notes that it was the use of the principle to embody `E` which causes embodiments `B.e1`, `B.e2`, `B.e3`, `B.e4` and `B.e5` – these are specific instances of the embodiments specified in the definition of the principle `bicycle` (see Figure 4.31).

The relation `must_be_directly_used_for` invoked in the above constraint is a generic relation defined in Section B.1. For convenience, it is quoted in Figure 4.51, where we can see that if a `means` `must_be_directly_used_for` a `func` they must both relate to the same scheme (Line 121) and the identifier of the `func` must be in the `funcs_provided` by the `means` (Line 122). Note that the use of the `!` operator in this relation definition (Line 122) means that, if the identifier of the `func` is not already in the `funcs_provided` by the means, it will be inserted into the set.

```
120 relation must_be_directly_used_for( means, func )
121     =::= { (M,F): is_in_the_same_scheme_as( M, F ) and
122                   F.id in !M.funcs_provided }.
```

Figure 4.51: The meaning of the relation `must_be_directly_used_for`.

Recall that it was said above (when discussing Figure 4.50) that, if all pre-existing instances of the `bicycle` principle are being used in other embodiments, they would, therefore, be unavailable for use in embodiment `E` and a new instance of the principle would be created. We will now consider which constraints would cause this. Refer to the constraint shown in Lines 39–40 of Figure 4.18. This stated that, for each means `M` (recall that a principle is a means), it must be true that the set of `funcs_provided` by `M` `is_a_possible_behaviour_of` of `M`. The pre-existing instances of the `bicycle` principle which are being used in other embodiments would violate this constraint because, as can be seen in Figure 4.29, a bicycle can provide only one instance of the function *provide transport*. The constraint filtering system maintains a quite sophisticated set of dependency records and the violation of the constraint in Figure 4.18 would cause the system to undo whatever action it had performed which caused the violation; this means that it would undo the insertion, caused by the relation in Figure 4.51, of the most recent `func_id` into the set of `funcs_provided` by the violating instance of the bicycle principle. When all pre-existing instances of the principle had been exhausted, this would trigger the `!` operator used in Line 89 Figure 4.50 to create a new instance of the principle.

In Lines 91–95 of Figure 4.50, the relation `causes` is invoked to state that the embodiments that have been introduced due to the use of the principle `bicycle` are *caused* by the `embodiment` whose `chosen_means` has been selected to be `a_bicycle`. The meaning of the `causes` relation is defined in `generic_concepts.gal` (in Section B.1) but is quoted in Figure 4.52. According to this definition, if an embodiment `P` `causes` another embodiment `E`, then, as well both embodiments being in the same scheme, `E` inherits all the `reasons` for `P` as well as having the identity number of the function instance in `P` itself as one of its `reasons`.

```
relation causes( embodiment, embodiment )
    =::= { (P,E): is_in_the_same_scheme_as( P, E ) and
                  E.reasons = {P.intended_func.id} union P.reasons }.
```

Figure 4.52: The meaning of the relation `causes`.

We have just seen, in Figure 4.50, a constraint which specifies some consequences of the user selecting a principle as the `chosen_means` for an embodiment. We will now consider what is implied by her choosing a design *entity* as the `chosen_means` for an embodiment. The consequences attached to using a design entity to embody a function are similar to those caused by using a design principle. The essential difference is that additional embodiments are introduced (which means that further functionality must be provided) when a design principle is used whereas, when a design entity is used, no further embodiments are introduced and no more functionality must be provided (at least in this part of the functional decomposition of the product).

The effect of using the `chassis` entity in a scheme is illustrated in Figure 4.53. The constraint in this figure is take from `raleigh_knowledge.gal` in Section C.1. It can be seen from this figure that, whenever a `chassis` is used as the `chosen_means` for an `embodiment`, there must exist an instance of the `chassis` design entity which `must_be_directly_used_for` the `intended_function` of the `embodiment`. Since the `!` operator is applied to the `exists` quantifier in this constraint (Line 100), an instance of the `chassis` design entity will be created, if necessary, to satisfy this constraint. The relation `must_be_directly_used_for` has already been discussed.

```
all embodiment(E): E.chosen_means = a_chassis implies
    !exists chassis( C ):
        must_be_directly_used_for( C, E.intended_function ).
```

Figure 4.53: Embodying a function using a design entity.

We have just seen that, as a designer chooses `known_means` for embodying the `intended_functions` of `embodiments`, new instances of design entities may be created. For example, in Figure 4.53 the introduction of an instance of the `chassis` design entity is required if the designer selects `a_chassis` as the `chosen_means` for an embodiment. She can only do this, of course, if the intended function of the embodiment is *provide support* because, as we have seen in Figure 4.29, this is the only function that can be provided by a `chassis`. Indeed, we can see in Figure 4.29 that a `chassis` can provide only one instance of this function so, if this functionality is required in two different embodiments within a design, two separate instances of the `chassis` design entity would be needed.

However, there are situations when one instance of a design entity can be used to embody more than one function instance. For example, consider the known means `a_molded_frame`. We can see, in Lines 150–152 of Figure 4.29, that this known means can provide two functions simultaneously: *provide support* and *support passenger*. If a designer selected this known means as the `chosen_means` to embody an instance of the function *provide support*, an instance of `molded_frame` design entity would be introduced, if none already existed; the constraint which would cause this is shown in Figure 4.54.

```
all embodiment(E): E.chosen_means = a_molded_frame implies
    !exists molded_frame( M ):
        must_be_directly_used_for( M, E.intended_function ).
```

Figure 4.54: Embodying a function using `a_molded_frame`.

If the designer later selected the same `known_means` to embody an instance of the function *support passenger* no new instance of the `molded_frame` design entity would be introduced. This is because both of these two function instances can be provided by the existing instance of the `molded_frame` design entity. In other words, the constraint in Lines 39–40 of Figure 4.18 which, as we saw above would be violated if the designer tried to use one instance of the bicycle principle to embody two instances of the function *provide transport*, would not be violated if she used one instance of the molded frame design entity to embody an instance of the function *support passenger* as well as an instance of the function *provide support*. This is because provision of two function instances, one of each of these two functions, `is_a_possible_behaviour_of` an instance of the molded frame design entity; in the definition of the relation `is_a_possible_behaviour_of` (Lines 82–85 in `generic_concepts.gal` in Section B.1), it can be seen that this relation is defined in terms of the relation `can_simultaneously_provide` and, as can be seen in Figure 4.29, a molded frame can simultaneously provide an instance of the function *provide transport* and one of the function *support passenger*.

The use of a design entity instance to provide more than one function instance is called *entity sharing*. The fact that the implementation of conceptual design reasoning we have described here can support entity sharing means that the collection of entity instances that are configured to provide the required functionality of a product is minimal. Entity sharing of the type just discussed was used in the scheme shown in Figure 3.20.

4.8 Summary

One of the most difficult aspects of providing computer-based support for conceptual engineering design is the modelling of products and specifications at various levels of abstraction. This chapter presented an approach to using constraints as a modelling paradigm for conceptual design.

Chapter 5

Illustration and Validation

In this chapter the approach to supporting conceptual design that has been proposed in this dissertation is demonstrated on two design problems. Firstly, in Section 5.1, a simple conceptual design problem is presented. The development of a set of alternative schemes for this problem is explained through the use of a series of interactions with a Galileo constraint filtering system. The various characteristics of the approach presented in this dissertation are explained in detail with respect to this design problem. Secondly, the conceptual design of an industrial product is presented in Section 5.2. This section describes how the approach proposed in this dissertation could be applied to support the design of a real-world product. The case-study design problem was provided by an electronic component manufacturer based in Cork called Bourns Electronics Ireland.

5.1 An Illustrative Example

In this section an example conceptual design problem is presented and a number of schemes are generated for it. The design problem considered is based on vehicle design. The presentation of this design problem comprises three phases. Firstly, in Section 5.1.1, a simple design specification will be presented and modelled using the constraint-based approach presented in Chapter 4. Secondly, in Section 5.1.2, an appropriate constraint-based design knowledge-base will be presented using the implementation approach presented in Chapter 4. Finally, in Section 5.1.3, the development of two schemes based on the design specification will be presented. The use of constraint filtering in the process of developing these schemes will be described using a number of screen-shots from a constraint filtering system that would be capable of reasoning about the extended version of the Galileo language that is recommended in this dissertation.

A full Galileo implementation of the design specification and the design knowledge-base used in this section are presented in Appendix C. The various project-specific concepts that are used to develop schemes for the vehicle design specification are based upon the generic design concepts presented in Appendix B.

5.1.1 An Example Design Specification

The design specification that is to be considered here is as follows:

> Design a product which satisfies the following requirements:
>
> - the product provides the function *provide transport*;
> - the product is fully *recyclable*;
> - the product has a *width not greater than 2m*;
> - the product has *minimal mass* and
> - the product comprises a *minimal number of parts*.

Using the techniques presented in Chapter 4, a constraint-based implementation of the design specification can be easily developed. Indeed, the modelling of this design specification has already been discussed in depth in Section 4.6. A full implementation of the design specification is presented in Section C.2 of Appendix C, in a Galileo module called `vehicle_spec.gal`.

This specification module imports a library containing a company-specific design knowledge-base (Line 2), a library of generic concepts (Line 3) and a library of concepts for comparing schemes using Pareto optimality (Line 4). The contents of the library of generic concepts has already been discussed (Chapter 4). The contents of the library containing company-specific design knowledge will be discussed later; here, we will focus on the contents of the vehicle specification module.

The notion of a vehicle scheme is defined in Lines 5–18. Line 6 specifies that a vehicle scheme must be a `scheme`, which is a generic concept imported in Line 3 from a library. Lines 7–18 specify that, in addition to possessing the characteristics of a generic scheme, a vehicle scheme must satisfy all the all the requirements which are defined in the design specification.

The functional property required by the design specification is modelled in Lines 7–8 using a relation imported from `generic_concepts.gal`. The requirement that the product be recyclable is modelled on Line 9. The physical requirement relating to the width of the product is modelled on Lines 10–12. The design preferences relating to the mass and number of parts of the product are modelled on Lines 13–15 and Lines 16–18, respectively, using functions imported from `raleigh_knowledge.gal`. The various relations used to compare alternative schemes, which are specific to this particular design specification, are defined on Lines 19–25; these relations are used to "plug-in" a definition for the relation `improves_on`, a relation which is expected by the generic system of libraries.

The constraint-based model presented in Section C.2 has been generated manually. However, the development of a computer tool for transforming the requirements defined in the design specification into an equivalent constraint-based model should not be a difficult task. However, these issues are beyond the scope of the research presented in this dissertation.

Once a constraint-based model of the design specification has been developed, a designer (or a team of designers) can begin to develop a set of alternative schemes by interacting with this model and with an appropriate design knowledge-base, using a constraint filtering system. However, before that process is discussed, the contents of an example constraint-based design knowledge-base will be discussed in Section 5.1.2.

5.1.2 An Example Design Knowledge-Base

In the scenario being discussed here, the knowledge-base that is being used contains a design principle based on the notion of a bicycle and a collection of design entities such as a wheel assembly, a pedal assembly, a saddle and handlebars. In Section C.1 of Appendix C, a constraint-based implementation of a suitable design knowledge-base is presented, in a Galileo module called `raleigh_knowledge.gal`.

A number of means are available in this design knowledge-base. These means are listed in a domain called `known_means` which is defined on Lines 41–46. For example, it can be seen that there is a means known as `a_bicycle`, another known as `a_skateboard` and a further one known as `a_wheel_assembly`.

The techniques used to implement the various design concepts contained in the

Galileo module raleigh_knowledge.gal were discussed in Section 4.5. Therefore, the implementation of this module will not be discussed any further, apart from the fact that, in the next section, particular aspects of the implementation will be highlighted when explaining the interactions, between the designer and the filtering system, that result in the development of a set of schemes.

5.1.3 Interactive Scheme Development

The filtering system being referred to in this section is based on previously published research on constraint-based reasoning for supporting concurrent engineering [19, 22]. However, the research presented in this dissertation takes a different perspective on the role that constraint filtering can play in designer support. In this research constraint filtering is used as a basis for providing interactive support to the human designer throughout the *entire* process of scheme development. This not only includes reasoning about the physical aspects of the scheme, but includes the design synthesis activities associated with developing a configuration of design entities from an initial statement of the required functionality that is to be provided by the product being designed.

The constraint filtering system being discussed here is assumed to be capable of reasoning about the extended version of the Galileo language that was discussed in Section 4.2. In addition, it is assumed that there are a number of different designers using this system. The meta-level language associated with this filtering system contains a number of pre-defined meta-level predicates such as #designer, #parameter, #was_introduced_by and #is_visible_to. These predicates can be used to write a constraint which states that a designer can only see those parameters which she introduced. This constraint can be implemented as follows:

```
all #designer(D), #parameter(P):
    #was_introduced_by(P,D) equiv
        #is_visible_to(P,D).
```

In the discussion which follows it is assumed that there are two different designers using the system, designer_1 and designer_2. In addition it is assumed that the above meta-level constraint is implicitly defined in the filtering system. Thus, when a screen-shot is presented, the name of the designer who is associated with that screen-shot will appear on the title of the interface that is depicted. In addition, the screen-shot will only contain parameters for which the named designer is responsible.

The interface which is depicted in the screen-shots is not intended to reflect an ideal interface with which a human designer would interact. Rather, it is intended as a means for explaining how the approach proposed in this dissertation could be used to develop a set of alternative schemes. The development of an appropriate interface with which a real designer would use is beyond the scope of this research.

Before considering these screen-shots in detail, some remarks should be made. The screen is divided into three areas. The title area, at the top of the screen, shows the name of the adviser system and the identity of the designer who is currently interacting with it – in Figure 5.1 the user is `designer_1`. The middle of the screen shows parameters that are visible to the user. The bottom, command area, of the screen shows the commands that have been entered by the user and any responses the system has made. Finally, note that the screen-shot shown in a figure shows the state of the screen *after* all the user's commands have been executed by the system.

Conceptual Design Adviser System – `designer_1`
> load vehicle_spec.gal Processing `vehicle_spec.gal`... Importing `raleigh_knowledge.gal`... Importing `generic_concepts.gal`... Importing `comparison.gal`... Done!

Figure 5.1: Loading the design specification model into the filtering system.

In Figure 5.1 the Galileo module called `vehicle_spec.gal`, which is defined in Section C.2, was loaded into the filtering system interpreter. This was done using the filtering system command `load`. Since `vehicle_spec.gal` imports the Galileo modules `raleigh_knowledge.gal`, `generic_concepts.gal` and `comparison.gal`, these modules are also loaded. These modules must be loaded into the filtering system before designers can begin to develop schemes for the product described in the design specification. The middle portion of the screen is empty because the user has not yet introduced any parameters.

```
Conceptual Design Adviser System – designer_1
[ ] scheme_1.scheme_name                      'my vehicle'
[ ] scheme_1.structure                        ▷
[ ] scheme_1.width                            _
[ ] scheme_1.mass                             ▷
[ ] scheme_1.number_of_parts                  ▷

> scheme_1 : vehicle_scheme
> expand scheme_1
> scheme_1.scheme_name ='my vehicle'
```

Figure 5.2: Introducing a scheme instance called `scheme_1`.

Figure 5.2 shows the state of the user interface after `designer_1` has invoked three commands. Firstly, an instance of the Galileo structured domain, `vehicle_scheme`, was introduced by the designer. This instance is being called `scheme_1`. Since `designer_1` introduced this parameter, `scheme_1` appears on her filtering system interface.

Then, the designer used the constraint filtering system command `expand` to examine the various fields associated with the parameter `scheme_1`. It can be seen that `scheme_1` is a structured parameter which comprises five fields: a scalar field called `scheme_name`, a structured field called `structure`, a scalar field called `width`, a structured field called `mass` and a structured field called `number_of_parts`[1]. These fields exist due to the domain definition of `vehicle_scheme` defined in Lines 5–18 of `vehicle_spec.gal` in Section C.2. There, a `vehicle_scheme` was defined to be a generic scheme (which has two fields, `scheme_name` and `structure` – see Lines 46–48 of Section B.1 of Appendix B) with three additional fields, `width`, `mass`

[1]Structured fields are indicated on the screen by the presence of a ▷, which is intended to "invite" the user to examine the field further by expanding it; the value of a scalar field whose value is known is shown; for a scalar field whose value is not yet known a "_" is shown.

and `number_of_parts`. Finally, in Figure 5.2 the designer asserted a value for the `scheme_name` field of the parameter `scheme_1`; the value asserted was '`my vehicle`'.

Conceptual Design Adviser System – `designer_1`	
[ ] `scheme_1.scheme_name`	*'my vehicle'*
[ ] `scheme_1.structure.intended_function.verb`	`provide`
[ ] `scheme_1.structure.intended_function.noun`	`transport`
[ ] `scheme_1.structure.intended_function.id`	0
[ ] `scheme_1.structure.chosen_means`	_
[ ] `scheme_1.structure.reasons`	{}
[ ] `scheme_1.width`	_
[ ] `scheme_1.mass.value`	_
[ ] `scheme_1.mass.intent`	`minimal`
[ ] `scheme_1.number_of_parts.value`	_
[ ] `scheme_1.number_of_parts.intent`	`minimal`
> expand scheme_1.structure	
> expand scheme_1.structure.intended_function	
> expand scheme_1.mass	
> expand scheme_1.number_of_parts	

Figure 5.3: Examining the initial state of the scheme.

In Figure 5.3 the initial state of `scheme_1` was examined by `designer_1`. In order to get a full picture of the design specification, the `structure` field (along with its sub-field `intended_function`), the `mass` field and the `number_of_parts` field were all expanded.

It can be seen from Figure 5.3 that the `intended_function` of the `structure` of `scheme_1` is to `provide transport`. This function was specified in the definition of the `vehicle_scheme` (Lines 7–8 of Section C.2). This function has an identifier (`id`) of 0 and the empty set as its set of `reasons`. This reflects the fact that this function is the first function to be introduced into the scheme; in other words, no other function was responsible for this function being introduced into the scheme. The assignment of the 0 identifier and the empty set of reasons is done by the constraint, quantified over all schemes, in Lines 49–50 of Section B.1; this constraint uses the relation defined in Lines 116–119 of Section B.1, which actually makes the assignments.

It can also be seen from this figure that the `mass` and `number_of_parts` associated with `scheme_1` are preferences. It can be seen that, in both cases, the `intent` is that these should have `minimal` values; this comes from Lines 14 and 17 of the specification in Section C.2.

Conceptual Design Adviser System – designer_1	
[ ] scheme_1.scheme_name	*'my vehicle'*
[ ] scheme_1.structure.intended_function.verb	provide
[ ] scheme_1.structure.intended_function.noun	transport
[ ] scheme_1.structure.intended_function.id	0
[ ] scheme_1.structure.chosen_means	_
[ ] scheme_1.structure.reasons	{}
[ ] scheme_1.width	_
[ ] scheme_1.mass	▷
[ ] scheme_1.number_of_parts	▷

> contract scheme_1.mass
> contract scheme_1.number_of_parts
> advise on scheme_1.structure.chosen_means
Advice: `scheme_1.structure.chosen_means` must be in the set:
{`a_bicycle, a_skateboard`}

Figure 5.4: Querying for a valid means for providing the function required by the design specification.

In Figure 5.4 the designer has hidden the details of the `mass` and `number_of_parts` fields of `scheme_1` using the filtering system command `contract`. The designer then asks for the set of consistent means that could be used to provide the `intended_function` of the `structure` of `scheme_1`. The designer is told that the current set of consistent means for this function comprises `a_bicycle` and `a_skateboard`. This response is due to the constraint defined on Lines 7–8 in Section B.1 which specifies that the means chosen for any `embodiment` (remember the `structure` of a scheme is an `embodiment`) must be able to provide the `intended_function` of that `embodiment`. The relation used in this constraint is defined in Lines 53–54 of Section B.1; it invokes the relation shown in Lines 132–164 of Section C.1, an inspection of which shows that two `known_means` can provide the function "provide transport".

Conceptual Design Adviser System – designer_1	
[ ] scheme_1.scheme_name	*'my vehicle'*
[ ] scheme_1.structure.intended_function.verb	provide
[ ] scheme_1.structure.intended_function.noun	transport
[ ] scheme_1.structure.intended_function.id	0
[ ] scheme_1.structure.chosen_means	a_bicycle
[ ] scheme_1.structure.reasons	{}
[ ] scheme_1.width	_
[ ] scheme_1.mass	$\triangleright$
[ ] scheme_1.number_of_parts	$\triangleright$
[ ] bicycle_1	$\triangleright$

> scheme_1.structure.chosen_means = a_bicycle
Info: Parameter bicycle_1 has been created...

Figure 5.5: Using the principle of a bicycle in scheme_1 to provide the function "provide transport".

Figure 5.5 depicts the scenario where the designer selects a_bicycle as the chosen_means for providing the intended_function of the structure of scheme_1. The effect of this is that a new parameter called bicycle_1 is automatically introduced. The parameter bicycle_1 is an instance of the structured domain bicycle. The introduction of this new parameter is due to the constraint defined on Lines 88–95 of Section C.1. This embodiment is illustrated in Figure 3.15.

Conceptual Design Adviser System – designer_1	
[ ] bicycle_1.type	a_principle
[ ] bicycle_1.funcs_provided	{0}
[ ] bicycle_1.e1.intended_function.verb	facilitate
[ ] bicycle_1.e1.intended_function.noun	movement
[ ] bicycle_1.e1.intended_function.id	1
[ ] bicycle_1.e1.chosen_means	a_wheel_assembly
[ ] bicycle_1.e1.reasons	{0}
[ ] bicycle_1.e2	▷
[ ] bicycle_1.e3	▷
[ ] bicycle_1.e4	▷
[ ] bicycle_1.e5	▷
[ ] wheel_assembly_1	▷

```
> focus on bicycle_1
> expand bicycle_1.e1
> expand bicycle_1.e1.intended_function
> bicycle_1.e1.chosen_means = a_wheel_assembly
Info: Parameter wheel_assembly_1 has been created...
```

Figure 5.6: Incorporating a `wheel_assembly` entity to provide the function "`facilitate movement`".

In Figure 5.6 the designer begins to explore the parameter `bicycle_1`. The designer uses the filtering system command "`focus on`" to clear the filtering system interface of all parameters except for the parameter of interest – in this case the `bicycle_1` parameter. Since `bicycle_1` is a structured parameter, the designer can use the `expand` command to explore its fields. It can be seen that `bicycle_1` is a design principle and that the `funcs_provided` by this principle is a singleton set containing the value 0; this means that `bicycle_1` provides one function, namely, the function whose `id` is 0 – this was seen in Figure 5.3 to be the function required in the design specification.

It can also be seen from Figure 5.6 that `bicycle_1` contains a number of other structured fields, namely, `e1`, `e2`, `e3`, `e4` and `e5`. These fields represent further embodiments which the designer must make in order to properly incorporate the `bicycle_1` design principle into `scheme_1`.

In Figure 5.6 the designer explores the parameter `bicycle_1.e1` by expanding it. It can be seen that the function to be embodied is `facilitate movement`. This function has an `id` of 1 since this is the next unique function identifier for this scheme.

The constraint which is responsible for determining this value is in Lines 21–22 of Section B.1. This constraint calls the relation defined in Lines 77–79 of Section B.1 which in turn calls the relation defined in Lines 66–70 of Section B.1. The meaning of this constraint and these relations is that a function identifier is only valid if it a positive integer and if only one function within a scheme has that integer as an identifier; in determining an identifier for a newly introduced function, the system merely chooses the next positive integer that has not already been used.

The `reasons` for the embodiment `bicycle_1.e1` is the singleton set containing the function identifier 0; this represents the fact that the function whose identifier is 0 is a reason for this embodiment.

Finally, in Figure 5.6, the designer chooses `a_wheel_assembly` as the `chosen_means` for this embodiment. This causes the automatic introduction of another new parameter, `wheel_assembly_1`, into the scheme, because it triggered the constraint in Lines 129–131 of Section C.1. This embodiment is illustrated in Figure 3.16.

Conceptual Design Adviser System – `designer_1`

[ ]	`bicycle_1`	▷
[ ]	`wheel_assembly_1.type`	`an_entity`
[ ]	`wheel_assembly_1.funcs_provided`	{1}
[ ]	`wheel_assembly_1.id`	1
[ ]	`wheel_assembly_1.width`	_
[ ]	`wheel_assembly_1.mass`	_
[ ]	`wheel_assembly_1.material`	_

> contract bicycle_1
> expand wheel_assembly_1

Figure 5.7: Examining the `wheel_assembly`.

In Figure 5.7 the designer explores the `wheel_assembly_1` parameter. When the designer expands this parameter she sees that it comprises several fields. The `type` field indicates that this new parameter represents a design entity. The `funcs_provided` by this entity is a singleton set which refers to a function whose identifier is 1, namely the function "facilitate movement" seen in Figure 5.6. The `id` field of wheel_assembly_1 indicates that it is the first design entity to be incorporated into this scheme. This value is computed from the constraint defined on Lines 13–14 of Section B.1. A number of other fields are illustrated in Figure 5.7. These are the fields width, mass and material associated with wheel_assembly_1. As the designer develops the scheme she may make choices about the values that will be associated with these parameters.

Conceptual Design Adviser System – designer_1	
[ ] bicycle_1.type	a_principle
[ ] bicycle_1.funcs_provided	{0}
[ ] bicycle_1.e1	▷
[ ] bicycle_1.e2.intended_function.verb	provide
[ ] bicycle_1.e2.intended_function.noun	energy
[ ] bicycle_1.e2.intended_function.id	2
[ ] bicycle_1.e2.chosen_means	a_pedal_assembly
[ ] bicycle_1.e2.reasons	{0}
[ ] bicycle_1.e3	▷
[ ] bicycle_1.e4	▷
[ ] bicycle_1.e5	▷
[ ] wheel_assembly_1	▷
[ ] pedal_assembly_1	▷
[ ] chain_1	▷
[ ] mechanical_interface_1	▷
[ ] mechanical_interface_2	▷

```
> contract wheel_assembly_1
> expand bicycle_1
> expand bicycle_1.e2
> expand bicycle_1.e2.intended_function
> bicycle_1.e2.chosen_means = a_pedal_assembly
Info: Parameter pedal_assembly_1 has been created...
Info: Parameter chain_1 has been created...
Info: Parameter mechanical_interface_1 has been created...
Info: Parameter mechanical_interface_2 has been created...
```

Figure 5.8: Using a pedal assembly to provide the function "provide energy".

In Figure 5.8 the designer selects a_pedal_assembly as the chosen_means to embody the function provide energy. This causes a new parameter, pedal_assembly_1, to be introduced. Although it is not apparent in Figure 5.8, the parameter pedal_assembly_1 is a design entity. If the designer were to expand pedal_assembly_1 we would see that its id field contains the value 2, reflecting the fact that it is the second design entity to be incorporated into the scheme.

Once the designer selects a_pedal_assembly as the chosen_means to embody the function provide energy a number of other parameters are also immediately introduced into the scheme. These are chain_1, mechanical_interface_1 and mechanical_interface_2. These parameters exist in order to fulfil the context relation drives that must exist between the embodiments for the functions provide power and facilitate movement, as specified in the principle for a bicycle. The need for this context relation is stated in Line 24 of Section C.1; its meaning is specified in Lines 173–185 of Section C.1, which is responsible for the introduction of the additional parameters in Figure 5.8. The parameter chain_1 exists in order to satisfy the context relation that the embodiment for the function provide power drives the embodiment for the function facilitate movement. According to the relation defined on Lines 173– 185 of Section C.1, there must be a mechanical_interface between pedal_assembly_1 and chain_1 and another between wheel_assembly_1 and chain_1. This will be explored in further detail in Figure 5.9. This scenario corresponds to that shown in Figure 3.17.

Conceptual Design Adviser System – designer_1	
[ ] bicycle_1	▷
[ ] wheel_assembly_1	▷
[ ] pedal_assembly_1	▷
[ ] chain_1	▷
[ ] mechanical_interface_1.entity_1	1
[ ] mechanical_interface_1.entity_2	3
[ ] mechanical_interface_1.type	mechanical
[ ] mechanical_interface_1.relationship	drives
[ ] mechanical_interface_2	▷
> contract bicycle_1	
> expand mechanical_interface_1	

Figure 5.9: Embodying the drives context relation.

The context relation drives requires a chain design entity acting between the design entities wheel_assembly_1 and pedal_assembly_1. In Figure 5.9 mechanical_interface_1 is being explored. It can be seen that mechanical_interface_1 implements a drives relationship between the entities whose identifiers are 1 and 3, namely, `wheel_assembly_1` and `chain_1`, respectively. (The parameter `chain_1` is the third design entity to be incorporated into this scheme; thus, if we were to expand it, we would see that its `id` field contains the value 3).

Conceptual Design Adviser System – `designer_1`	
[ ] `scheme_1`	▷
[ ] `bicycle_1`	▷
[ ] `wheel_assembly_1`	▷
[ ] `pedal_assembly_1`	▷
[ ] `chain_1`	▷
[ ] `mechanical_interface_1`	▷
[ ] `mechanical_interface_2`	▷
[ ] `frame_1`	▷
[ ] `mechanical_interface_3`	▷
[ ] `mechanical_interface_4`	▷
[ ] `saddle_1`	▷
[ ] `mechanical_interface_5`	▷
[ ] `handlebar_assembly_1`	▷
[ ] `mechanical_interface_6`	▷
> focus on all	

Figure 5.10: The state of `scheme_1` once means have been selected for each function.

In Figure 5.10 the state of `scheme_1` is presented after several more decisions have been made by the designer, namely after means have been selected to provide each function that is associated with the scheme. This scheme comprises a number of design entities, interfaces and principles. The scheme was based on the principle of a bicycle. The final scheme comprises six design entities and six interfaces. The bicycle principle introduced five functions into the scheme: `facilitate movement`, `provide energy`, `provide support`, `support passenger` and `change direction`. The introduction of these functions by the use of a bicycle design principle is caused by Lines 9–23 of Section C.1.

The `wheel_assembly_1` entity was used by the designer to provide the function `facilitate movement`. The `pedal_assembly_1` entity was used to provide the function `provide energy`. The `chain_1` entity was required to fulfill the `drives` context relationship that must exist between the embodiments of the functions `facilitate movement` and `provide energy` (Line 24, Section C.1). The interface between `wheel_assembly_1` and `chain_1` is embodied through `mechanical_interface_1`.

The interface between `pedal_assembly_1` and `chain_1` is embodied through `mechanical_interface_2`. The `frame_1` entity was used by the designer to provide the function `provide support`. The `supports` context relation between the embodiments of the functions `facilitate movement` and `provide support` (Line 25, Section C.1) is embodied by `mechanical_interface_3`. The `supports` context relation between the embodiments of the functions `provide energy` and `provide support` (Line 26, Section C.1) is embodied by `mechanical_interface_4`. The `saddle_1` entity was used by the designer to provide the function `support passenger`. The `supports` context relation between the embodiments of the functions `support passenger` and `provide support` (Line 27, Section C.1) is embodied by `mechanical_interface_5`. The `handlebar_assembly_1` entity is used to provide the function `change direction`. The `supports` context relation between the embodiments of the functions `change direction` and `provide support` (Line 28, Section C.1) is embodied by `mechanical_interface_6`. This scenario is also shown in Figure 3.19.

Conceptual Design Adviser System – `designer_1`	
[ ] `scheme_1`	▷
[ ] `bicycle_1`	▷
[ ] `wheel_assembly_1.type`	`an_entity`
[ ] `wheel_assembly_1.funcs_provided`	{1}
[ ] `wheel_assembly_1.id`	1
[ ] `wheel_assembly_1.width`	_
[ ] `wheel_assembly_1.mass`	2
[ ] `wheel_assembly_1.material`	`cfrp`
[ ] `pedal_assembly_1`	▷
[ ] `chain_1`	▷
[ ] `mechanical_interface_1`	▷
[ ] `mechanical_interface_2`	▷
[ ] `frame_1`	▷
[ ] `mechanical_interface_3`	▷
[ ] `mechanical_interface_4`	▷
[ ] `saddle_1`	▷
[ ] `mechanical_interface_5`	▷
[ ] `handlebar_assembly_1`	▷
[ ] `mechanical_interface_6`	▷
> expand wheel_assembly_1 > wheel_assembly_1.material = cfrp	

Figure 5.11: Selecting materials for the entities in `scheme_1`.

In Figure 5.11 `designer_1` has begun to select materials for the entities in `scheme_1`. In this figure the designer selects the material `cfrp` for the `wheel_assembly_1` design entity. Once the designer selects a material for this entity, its (relative) mass can be estimated by the system. Here the (relative) mass of the `wheel_assembly_1` entity is estimated to be 2 by the function `mass_of` defined on Lines 236–240 of Section C.1.

Conceptual Design Adviser System – `designer_1`	
[ ] `scheme_1.scheme_name`	*'my vehicle'*
[ ] `scheme_1.structure`	▷
[ ] `scheme_1.width`	_
[ ] `scheme_1.mass.value`	`12`
[ ] `scheme_1.mass.intent`	`minimal`
[ ] `scheme_1.number_of_parts.value`	`6`
[ ] `scheme_1.number_of_parts.intent`	`minimal`
[ ] `bicycle_1`	▷
[ ] `wheel_assembly_1`	▷
[ ] `pedal_assembly_1`	▷
[ ] `chain_1`	▷
[ ] `mechanical_interface_1`	▷
[ ] `mechanical_interface_2`	▷
[ ] `frame_1`	▷
[ ] `mechanical_interface_3`	▷
[ ] `mechanical_interface_4`	▷
[ ] `saddle_1`	▷
[ ] `mechanical_interface_5`	▷
[ ] `handlebar_assembly_1`	▷
[ ] `mechanical_interface_6`	▷
> expand scheme_1 > expand scheme_1.mass > expand scheme_1.number_of_parts	

Figure 5.12: The state of `scheme_1` once materials have been selected for all the entities from which it is configured.

In Figure 5.12 the state of `scheme_1` is shown after several more decisions have been made by the designer, namely after materials have been selected for all the entities from which it is configured. In this figure, the `mass` and `number_of_parts` fields of `scheme_1` have been expanded. It can be seen that the total mass of this scheme is estimated to be 12 units and that it comprises 6 parts.

The next few figures show a different scenario. They show snapshots from the development by another designer, designer_2, of a different scheme for the specification in Section C.2. As we will see, the system will automatically use Pareto optimality to compare this second scheme with the scheme that was developed by designer_1.

Conceptual Design Adviser System – designer_2	
[ ] scheme_2	▷
[ ] bicycle_2	▷
[ ] wheel_assembly_2	▷
[ ] pedal_assembly_2	▷
[ ] chain_2	▷
[ ] mechanical_interface_7	▷
[ ] mechanical_interface_8	▷
[ ] frame_2	▷
[ ] mechanical_interface_9	▷
[ ] mechanical_interface_10	▷
[ ] saddle_2	▷
[ ] mechanical_interface_11	▷
[ ] handlebar_assembly_2	▷
[ ] mechanical_interface_12	▷
>	

Figure 5.13: A second scheme, scheme_2, has been developed by the designer known as designer_2.

In Figure 5.13 a second scheme, scheme_2, is presented which has been developed by designer_2. Although designer_2 worked completely independently of designer_1, the exact same approach as that for scheme_1 was used and the same means to embody its functions. As we will see, however, she chooses different material from those chosen by designer_1.

Conceptual Design Adviser System – designer_2	
[ ] scheme_2	▷
[ ] bicycle_2	▷
[ ] wheel_assembly_2.type	an_entity
[ ] wheel_assembly_2.funcs_provided	{1}
[ ] wheel_assembly_2.id	1
[ ] wheel_assembly_2.width	_
[ ] wheel_assembly_2.mass	10
[ ] wheel_assembly_2.material	steel
[ ] pedal_assembly_2	▷
[ ] chain_2	▷
[ ] mechanical_interface_7	▷
[ ] mechanical_interface_8	▷
[ ] frame_2	▷
[ ] mechanical_interface_9	▷
[ ] mechanical_interface_10	▷
[ ] saddle_2	▷
[ ] mechanical_interface_11	▷
[ ] handlebar_assembly_2	▷
[ ] mechanical_interface_12	▷
> expand wheel_assembly_2	
> wheel_assembly_2.material = steel	

Figure 5.14: A material is selected for the wheel_assembly_2 entity used in scheme_2.

In Figure 5.14 designer_2 begins to select materials for the various design entities associated with scheme_2. She begins with wheel_assembly_2, selecting steel as its material. Thus, the mass of this entity is estimated to be 10 units (Line 240, Section C.1).

Conceptual Design Adviser System – `designer_2`	
[ ] `scheme_2`	▷
[ ] `bicycle_2`	▷
[ ] `wheel_assembly_2`	▷
[ ] `pedal_assembly_2.type`	`an_entity`
[ ] `pedal_assembly_2.funcs_provided`	{1}
[ ] `pedal_assembly_2.id`	2
[ ] `pedal_assembly_2.width`	_
[ ] `pedal_assembly_2.mass`	`10`
[ ] `pedal_assembly_2.material`	`steel`
[ ] `chain_2`	▷
[ ] `mechanical_interface_7`	▷
[ ] `mechanical_interface_8`	▷
[ ] `frame_2`	▷
[ ] `mechanical_interface_9`	▷
[ ] `mechanical_interface_10`	▷
[ ] `saddle_2`	▷
[ ] `mechanical_interface_11`	▷
[ ] `handlebar_assembly_2`	▷
[ ] `mechanical_interface_12`	▷

> contract wheel_assembly_2
> expand pedal_assembly_2
> pedal_assembly_2.material = steel
VIOLATION: The following constraint has been violated:
`alldif scheme(S1), scheme(S2):  not dominates( S1, S2 ).`

Figure 5.15: Due to her choice of materials, `scheme_2` is dominated by the scheme developed earlier.

In Figure 5.15 `designer_2` selects steel as the material for `pedal_assembly_2`. Upon the designer making this assertion, a constraint violation is detected. This constraint violation is related to the fact that the system has recognised that there exists a scheme which is dominated by another. The reason for this will be discussed in conjunction with Figure 5.16.

Conceptual Design Adviser System – `designer_2`	
[ ] `scheme_2.scheme_name`	*'another vehicle'*
[ ] `scheme_2.structure`	▷
[ ] `scheme_2.width`	_
[ ] `scheme_2.mass.value`	`20`
[ ] `scheme_2.mass.intent`	`minimal`
[ ] `scheme_2.number_of_parts.value`	`6`
[ ] `scheme_2.number_of_parts.intent`	`minimal`
[ ] `bicycle_2`	▷
[ ] `wheel_assembly_2`	▷
[ ] `pedal_assembly_2`	▷
[ ] `chain_2`	▷
[ ] `mechanical_interface_7`	▷
[ ] `mechanical_interface_8`	▷
[ ] `frame_2`	▷
[ ] `mechanical_interface_9`	▷
[ ] `mechanical_interface_10`	▷
[ ] `saddle_2`	▷
[ ] `mechanical_interface_11`	▷
[ ] `handlebar_assembly_2`	▷
[ ] `mechanical_interface_12`	▷

> contract pedal_assembly_2
> expand scheme_2
> expand scheme_2.mass
> expand scheme_2.number_of_parts

Figure 5.16: The state of `scheme_2` after materials have been selected for `wheel_assembly_2` and `pedal_assembly_2`.

In Figure 5.16 the state of `scheme_2` after materials have been selected for `wheel_assembly_2` and `pedal_assembly_2` is illustrated. It can be seen that the mass of `scheme_2` is currently estimated to be 20 units and that it comprises 6 parts. Therefore, this scheme is certainly dominated by `scheme_1` since, although both schemes have the same number of parts as each other, `scheme_1` has the smaller mass. This means that, since `scheme_2` does not improve on `scheme_1` on any design preference, `scheme_2` is dominated by `scheme_1`.

It is not obvious just from a constraint violation what corrective action is required of a designer in order to ensure that a particular scheme is no longer dominated. In the discussion of scheme development presented here these issues have not been addressed. The manner in which advice on what schemes dominate a particular scheme is presented to the designer has been regarded as an interface issue. The presentation here has not addressed the issue of a suitable interface for constraint-aided conceptual design. However, a number of possibilities are obvious. For example, a graphical tool could display a window which contains in a tabular form the schemes that have been developed against their current values for the various preferences that have been defined in the design specification. When a designer develops a scheme which is dominated by one or more schemes the dominating schemes could be highlighted in some way. Thus, the designer simply gets feedback stating that it is possible to develop a better scheme, but not how to do it. In this way, the designer's independence of thought is maintained.

5.1.4 Review of the Example Design Problem

In Section 5.1.3 the use of constraint filtering to provide interactive support to a designer developing constraint-based scheme models to satisfy a design specification has been presented.

A graphical representation of `scheme_1` is presented in Figure 5.17. Both of the schemes developed in this section were based on the same means for embodying each function in the scheme. In this figure it can be seen that there was an initial embodiment which had to be made for the function "*provide transport*". The bicycle design principle was used to provide this function. This design principle introduced the need for five further embodiments to be made: "*facilitate movement*", "*provide energy*", "*support passenger*", "*change direction*" and "*provide support*". These functions were embodied using five design entities: a wheel assembly, a pedal assembly, a saddle, a handlebar assembly and a frame. During the embodiment of the functions with these means a number of context relations had to be embodied between the entities used. These are all illustrated in Figure 5.17.

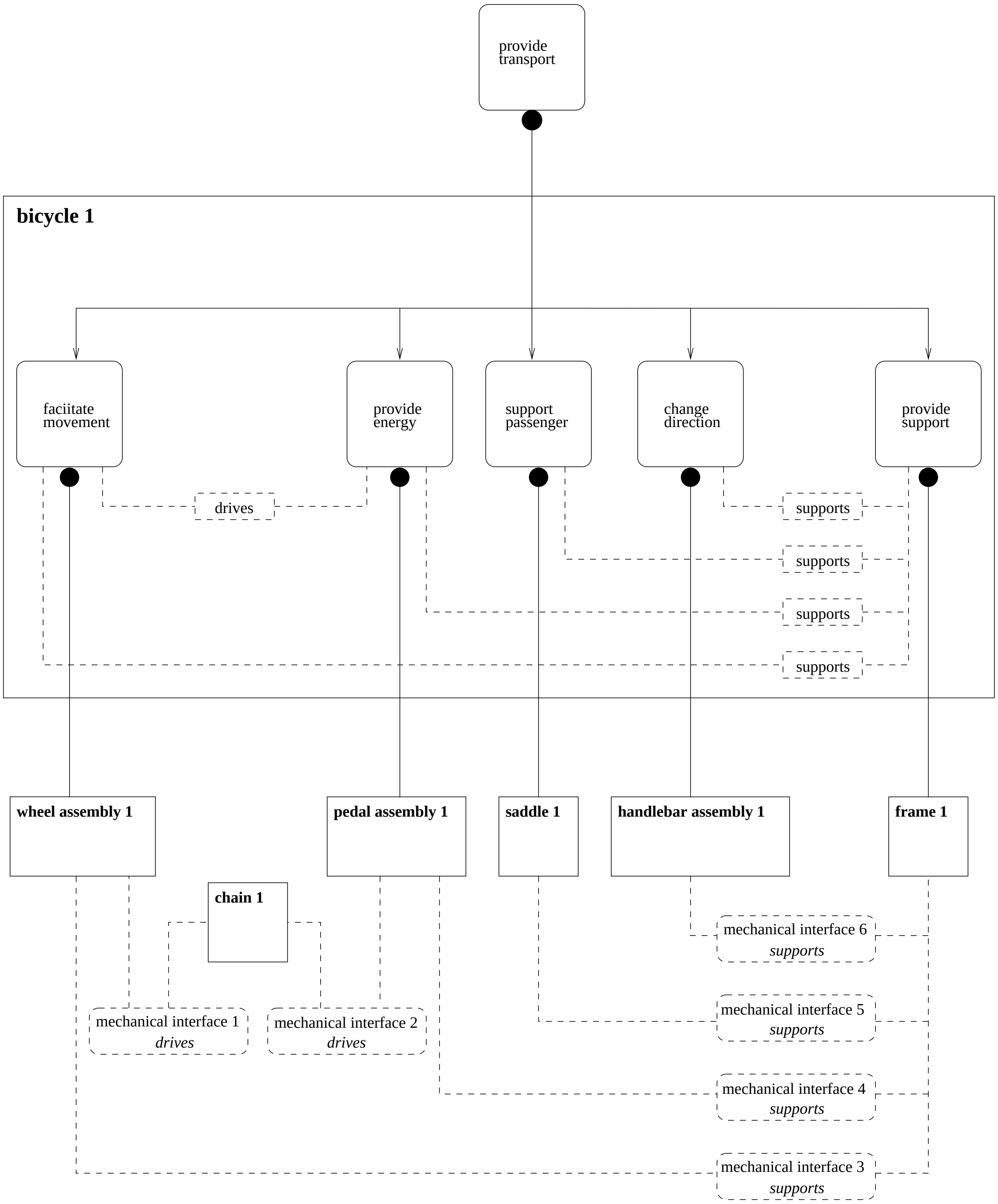


Figure 5.17: The structure of the scheme presented in Section 5.1.

5.2 An Industrial Case-Study

This section describes, through the use of an industrial case-study, how the approach to conceptual design proposed in this dissertation could be applied to support the design of a real-world product. The case-study was provided by an electronic component manufacturer based in Cork called Bourns Electronics. The section begins with a profile of the company. The case-study product is discussed in general terms and an informal design specification for it is presented. The knowledge that is required to design the product is presented and it is demonstrated how this can be formalised using the modelling approach developed in this research. A formal design specification is then presented, which incorporates all of the requirements that the product is required to satisfy. A number of schemes are also discussed which satisfy the requirements defined in the design specification.

5.2.1 A Profile of the Company

Bourns Electronics Incorporated was founded in 1947 in Altadena, California. Bourns has 11 manufacturing facilities worldwide. Amongst these are manufacturing plants in California, Utah, Mexico, Costa Rica, Ireland, China and Taiwan. In the early 1980's, a new subsidiary company, called Bourns Electronics Limited, was opened in Cork, Ireland to support growing European market demands. It is in this plant that the case-study discussed here was undertaken.

Bourns manufactures over 4,000 products, which are designed for use in virtually every type of electronic system. The Bourns product offering includes chip resistors/arrays, modular contacts, resistor networks, dials, multi-fuse resettable fuse arrays, surge products, encoders, panel controls, switches, inductive components, precision potentiometers, trimming potentiometers and linear motion potentiometers. The company's products are used in the automotive, industrial, medical, computer, audio/visual, telecommunications and aerospace industries.

5.2.2 Discrete Components from Bourns

Bourns' discrete components meet a variety of electronics design needs. They are small in size, and give design engineers solutions to address the growing portable electronics market and other industries where conserving board space is a critical issue.

Figure 5.18: A sample collection of discrete electronic components manufactured by Bourns Electronics

A selection of discrete components manufactured by Bourns is illustrated in Figure 5.18. In this figure components such as male and female modular contacts, sealed surface-mount technology (SMT) switches and thick-film chip resistors are illustrated. Modular contacts provide an off-the-shelf solution for facilitating connectivity in portable electronic systems. Sealed SMT switches are available in sizes starting at 3mm and are suitable for all popular soldering and cleaning methods, above and below the board. Thick-film chip resistors are available in surface-mount compatible packaging. Bourns also offers a wide selection of inductor and transformer components.

5.2.3 The Case-Study Product

The case-study product discussed here relates to the design of electrical contacts. More specifically, the case-study relates to systems of electronic contacts that involve male and female components which facilitate electrical contact. These contact systems are generally used in electronics applications where some part of the product must be in electrical contact with another part. For example, when fitting a battery pack into a portable computer or a mobile phone, there is a requirement for an electrical connection between the battery-pack and the remainder of the system. This type of electrical connection is often facilitated by the use of a modular contact system. An example of a modular contact system which is currently being marketed by Bourns Electronics is illustrated in Figure 5.19. This system is known as the Bourns 70AD modular contact system. The system comprises a male contact (highlighted on the left) and a female contact (highlighted on the right).

Figure 5.19: The Bourns 70AD Modular Contact System

There is a second modular contact system available from Bourns Electronics. This system is known as the Bourns 70AA modular contact system. However, this system comprises only a male contact. The male component of the system is illustrated in Figure 5.20.

Figure 5.20: The Bourns 70AA/Male Modular Contact

The female component of the Bourns 70AA modular contact system has not yet been designed. The conceptual design problem which will be addressed here will be the design of a female mating component for the Bourns 70AA/Male Modular Contact. The design of this component will complete a second modular contact system for the company.

5.2.4 The Design Specification

Based on the requirement for a complementary female modular contact for the 70AA/Male Modular Contact, a set of requirements for the desired component can be formulated. The set of requirements for the design of this new contact are presented in Table 5.1.

Property	***Requirement***
Required function	*provide structured contact*
Current	3 Amps
Contact pitch	2.54mm
Power consumption	minimal
Contact-solder resistance	minimal
Mass	minimal
General	End-to-end stackable

Table 5.1: An informal design specification for the Female Modular Contact that is required to complement the 70AA/Male Modular Contact

In Table 5.1 it can be seen that there are several different types of requirement in the design specification. Some of these requirements relate to the functional aspects of the product, while others relate to the physical and life-cycle aspects of the product. The function that is to be provided by the product is to "*provide structured contact*". The current that will be carried through the product will be 3 Amps and the product will have a contact pitch of 2.54mm. The contact pitch is the distance between adjacent points of contact on the component. The concept of contact pitch is illustrated in Figure 5.21.

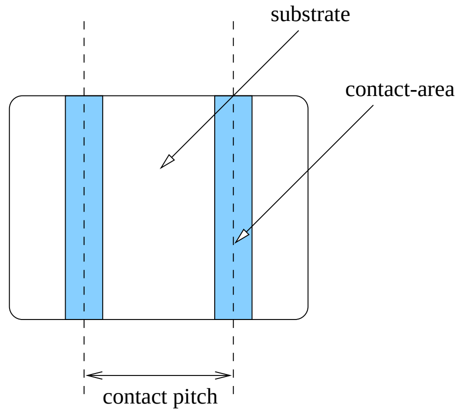


Figure 5.21: The meaning of the contact pitch requirement.

Also specified in Table 5.1 are a number of preferences about the physical aspects of the component. For example, the values of power consumption, contact-solder resistance and mass of the component should all be minimal. These requirements can be used to compare alternative schemes that are developed by the designer in order to satisfy the design specification.

The power consumption of the component relates to the total amount of power that is consumed by the component when it is in use. The contact-solder resistance is the total resistance of any conductive elements within the component.

Finally, there is a life-cycle requirement specified in the design specification, namely, that the product being designed should be end-to-end stackable. This requirement means that the contact pitch of a series of contacts that are placed end-to-end on a printed-circuit board (PCB) should be maintained. This requirement is illustrated in Figure 5.22.

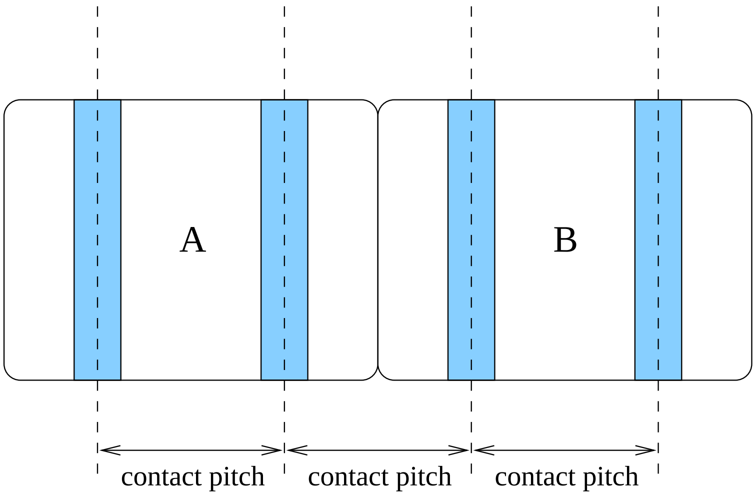


Figure 5.22: Graphical explanations of the life-cycle requirement defined in the design specification

A full Galileo implementation of the design specification discussed here is implemented in a module called `contact_spec.gal` presented in Section D.2 of Appendix D. However, for convenience, a segment of the Galileo model of the design specification is presented here in Figure 5.23[2].

This implementation of the design specification incorporates all of the requirements from the design specification discussed above. The functional property re-

[2]Please note that the line numbers in this figure correspond to the line numbers in `contact_spec.gal` presented in Section D.2.

```
domain contact_scheme
    =::= { S: scheme( S ) and
              provides_the_function( S.structure.intended_function,
              'provide', 'structured contact' ) and
              exists( S.contact_pitch : real ) and
              S.contact_pitch = 2.54 and
              exists( S.current : real ) and
              S.current = 3 and
              exists( S.power_consumption : preference ) and
              S.power_consumption.intent = minimal and
              !S.power_consumption.value  = power_consumption_of( S ) and
              exists( S.contact_to_solder_resistance : preference ) and
              S.contact_to_solder_resistance.intent = minimal and
              !S.contact_to_solder_resistance.value
                  = contact_to_solder_resistance_of( S ) and
              exists( S.mass : preference ) and
              S.mass.intent = minimal and
              !S.mass.value  = mass_of( S ) and
              end_to_end_stackable( S ) }.

relation has_better_contact_solder_resistance_than( scheme, scheme )
    =::= { (S1,S2): better_than( S1.contact_solder_resistance,
                    S2.contact_solder_resistance ) }.

relation has_better_mass_than( scheme, scheme )
    =::= { (S1,S2): better_than( S1.mass, S2.mass ) }.

relation has_better_power_consumption_than( scheme, scheme )
    =::= { (S1,S2): better_than( S1.power_consumption,
                    S2.power_consumption ) }.

relation improves_on( scheme, scheme )
    =::= { (S1,S2): has_better_power_consumption_than( S1, S2 ) or
                    has_better_contact_solder_resistance_than( S1, S2 ) or
                    has_better_mass_than( S1, S2 ) }.
```

Figure 5.23: A constraint-based model of the case-study design specification.

quired by the design specification is modelled in Lines 7–8. The contact pitch and current requirements are modelled on Lines 9–10 and Lines 11–12, respectively.

The design preferences relating to the power consumption, contact-to-solder resistance and mass of the product are modelled on Lines 13–15, Lines 16–18 and Lines 20–22, respectively. The value of the power consumption of the component is computed using a function called `power_consumption_of`. The value of the contact-to-solder resistance of the component is computed using a function called `contact_to_solder_resistance_of`. The value of the mass of the component is computed using a function called `mass_of`. Finally, the life-cycle requirement that the product be end-to-end stackable is modelled on Line 23. A relation called `end_to_end_stackable` is used. The various relations used to compare alternative schemes, which are specific to this particular design specification, are defined on Lines 24–35.

The meaning of the various functions and relations referred to in the design specification model presented in Figure 5.23 are part of the domain knowledge required to develop schemes for this product. In order to be able to develop a set of schemes for the product described in the design specification presented in Table 5.1 and Figure 5.23, an appropriate design knowledge-base must be available for designing electrical contact systems. This knowledge-base should contain a variety of means for providing typical functions in this domain. The knowledge-base should also contain various functions and relations which can be used to evaluate schemes from those perspectives which are relevant to the particular design problem. In the next section the contents of a suitable knowledge-base will be presented.

5.2.5 Modelling Design Knowledge for Bourns

In Section D.1 of Appendix D a full constraint-based implementation of a suitable design knowledge-base is presented in a Galileo module called `bourns_knowledge.gal`. Aspects of that knowledge-base will now be discussed, in order to demonstrate how the design knowledge required for the case-study can be modelled in a manner consistent with the approach presented in this dissertation.

```
domain known_means
    =::= { a_female_modular_contact, a_precious_metal_contact,
           a_substrate, a_molded_body, a_machined_body,
           a_thickfilm_termination, a_male_contact,
           a_conducting_strip, a_conducting_wire,
           a_metal_termination, a_thickfilm_conductor }.
```

Figure 5.24: Known means for use in designing electrical contact components.

The design knowledge-base that is presented here contains a number of known means which are useful when designing electrical contact components; these are presented in a domain called `known_means` in Figure 5.24.

```
124 relation can_simultaneously_provide( known_means, set of func )
125     =::= { (a_female_modular_contact,{F}):
126              provides_the_function(F,'provide','structured contact'),
127             (a_male_contact,{F}):
128              provides_the_function(F,'provide','structured contact'),
129             (a_substrate,{F}):
130              provides_the_function(F,'provide','support'),
131             (a_molded_body,{F}):
132              provides_the_function(F,'provide','support'),
133             (a_machined_body,{F}):
134              provides_the_function(F,'provide','support'),
135             (a_thickfilm_termination,{F}):
136              provides_the_function(F,'facilitate','external connection'),
137             (a_conducting_strip,{F1,F2,F3}):
138              provides_the_function(F1,'provide','contact') and
139              provides_the_function(F2,'connect','contact') and
140              provides_the_function(F3,'facilitate','external connection'),
141             (a_metal_termination,{F}):
142              provides_the_function(F,'facilitate','external connection'),
143             (a_precious_metal_contact,{F}):
144              provides_the_function(F,'provide','contact'),
145             (a_thickfilm_conductor,{F1,F2}):
146              provides_the_function(F1,'provide','contact') and
147              provides_the_function(F2,'connect','contact'),
148             (a_conducting_wire,{F}):
149              provides_the_function(F, 'connect', 'contact') }.
```

Figure 5.25: Modelling the functions that can be provided by a known means.

The various functions that can be provided by each of the known means is specified in the relation presented in Figure 5.25. For example, it can be seen from Lines 145–147 that `a_thickfilm_conductor` can simultaneously provide the functions `provide contact` and `connect contact`. The relation presented in this figure is structurally identical to that used in the example design problem discussed in Chapter 4 and Section 5.1.

```
30 domain female_modular_contact
31     =::= { F: principle(F) and
32             exists( F.e1 : embodiment ) and
33             exists( F.e2 : embodiment ) and
34             exists( F.e3 : embodiment ) and
35             exists( F.e4 : embodiment ) and
36             provides_the_function( B.e1.intended_function,
37                                  'provide', 'support' ) and
38             provides_the_function( B.e2.intended_function,
39                                  'provide', 'contact' ) and
40             provides_the_function( B.e3.intended_function,
41                                  'connect', 'contact' ) and
42             provides_the_function( B.e4.intended_function,
43                                  'facilitate', 'external connection' ) and
44             is_electrically_connected_to( B.e2, B.e3 ) and
45             is_electrically_connected_to( B.e3, B.e4 ) and
46             supports( B.e1, B.e2 ) and
47             supports( B.e1, B.e3 ) and
48             supports( B.e1, B.e4 ) }.
```

Figure 5.26: The female modular contact design principle.

The knowledge-base being discussed here contains one design principle, based on an idealised female modular contact. The Galileo representation of this principle is presented in Figure 5.26. The `female_modular_contact` design principle introduces the need for four embodiments to be made. These embodiments, `e1`, `e2`, `e3` and `e4`, are associated with the functions “provide support”, “provide contact”, “connect contact” and “facilitate external connection”, respectively. A number of context relations also appear in this design principle. Specifically, the embodiment of the function “provide contact” must be electrically connected to the embodiment of the function “connect contact” (Line 44) and the embodiment of the function “connect contact” must be electrically connected to the embodiment of the function “facilitate external connection” (Line 45). Additionally, the embodiment of the function “provide support” must support the embodiments of the functions “provide contact”, “connect contact” and “facilitate external connection” (Lines 46–48). The implementation of these relations can be found in the Galileo module `bourns_knowledge.gal` presented in Section D.1. However, the implementation of the `is_electrically_connected_to` relation will be discussed later in this section.

```
3 domain bourns_entity
4     =::= { R: entity(R) and
5               exists( R.width      : real ) and
6               exists( R.height     : real ) and
7               exists( R.length     : real ) and
8               exists( R.mass       : real ) and
9               exists( R.material   : known_material ) and
10              exists( R.resistance : real ) and
11              R.mass = mass_of( R ) and
12              R.resistance = resistance_of( R ) }.
```

Figure 5.27: The generic Bourns design entity.

In Figure 5.27 the generic design entity model appropriate for the case-study product is presented. The `bourns_entity` is a design entity which has a number of additional fields such as width, height, length, mass, material and resistance. There are a number of materials available which can be used as the material of choice for a Bourns design entity; these are presented in Lines 49–51 of Section D.1. The mass and resistance of the entity are computed using the functions `mass_of` and `resistance_of`, respectively (Lines 11 and 12). The implementation of these functions can be found in the Galileo module `bourns_knowledge.gal` presented in Section D.1. However, the implementation of the `resistance_of` function will be discussed later in this section.

```
22 domain conductor_entity
23     =::= { S: bourns_entity( S ) and
24               conductor_material( R.material ) }.


150 relation conductor_material( known_material )
151     =::= { gold, palladium_silver, palladium_gold, platinum_gold }.
```

Figure 5.28: A conductor entity for on the Bourns case-study.

In the company where this case-study was carried out, there are two classes of design entity: conductor entities and substrate entities. These entity classes represent significantly different entity types relevant to the design of electrical contact systems. In Figure 5.28 the model of a `conductor_entity` is presented. A conductor entity is based on the generic Bourns entity, but must be made of a conductor material (Line 24) There are a number of conductor materials available such as gold and palladium silver (Lines 150–151). In the Galileo module `bourns_knowledge.gal`, presented in Section D.1, several design entities are defined in terms of a conductor design entity.

```
domain bourns_interface
    =::= { I: interface(I) and
              exists( I.type : bourns_interface_type ) }.

domain bourns_interface_type
    =::= { spatial, physical, electrical }.
```

Figure 5.29: Suitable interfaces for the Bourns case-study.

In Figure 5.29 a system of interfaces for this case-study is presented. A `bourns_interface` is an interface which has a `type` field that can take three possible values: `spatial`, `physical` or `electrical`.

```
domain electrical_interface
    =::= { S: bourns_interface(S) and S.type = electrical and
              exists( S.relationship : electrical_relationship ) }.

domain electrical_relationship
    =::= { insulative, conductive }.
```

Figure 5.30: Modelling electrical interfaces.

In Figure 5.30 an `electrical_interface` is defined in terms of a `bourns_interface`. An electrical interface can be used to represent various relationships between design entities; in this case-study an electrical interface can either represent an insulative or conductive relationship.

```
relation is_electrically_connected_to( entity, entity )
    =::= { (E1,E2): E1 = E2 and conductor_entity(E1),
           (E1,E2): is_in_the_same_scheme_as( E1, E2 ) and
                    !exists electrical_interface(E):
                          E.entity1 = E1.id and
                          E.entity2 = E2.id and
                          E.relationship = conductive }.
```

Figure 5.31: The "electrically connected to" context relation.

The context relation "electrically connected to" can be modelled in terms of a conductive electrical interface between a pair of design entities. This is illustrated in Figure 5.31. This relation between design entities is used, in the same way as was done in Section 5.1, to implement the analogous relation between the embodiments within a principle.

```
function contact_to_solder_resistance_of( scheme ) -> real
    =::= { R -> sum( { E.resistance | exists conducting_entity(E):
                                      is_a_part_of( E, S ) }.
```

Figure 5.32: Computing the contact-to-solder resistance of a scheme.

In the design specification model presented in Figure 5.23 a number of functions are used to compute the values of various properties of a scheme. For example, a function called `contact_to_solder_resistance_of` is used to computer the contact-solder resistance of a scheme. An appropriate implementation of this function is presented in Figure 5.32. The contact-to-solder resistance of a scheme is computed as the sum of the resistances of all the conducting entities used in the scheme.

```
function resistance_of( bourns_entity ) -> real
    =::= { E -> resistivity_of( E.material ) * E.length
                / ( E.width * E.height ) }.

function resistivity_of( known_material ) -> real
    =::= { gold              -> 0.005,
           palladium_silver -> 0.03,
           palladium_gold   -> 0.05,
           platinum_gold    -> 0.05,
           alumina96         -> 1.0E14,
           alumina995        -> 1.0E14,
           beryllia995       -> 1.0E14 }.
```

Figure 5.33: Computing the resistance of a design entity.

As seen earlier in Figure 5.27, the resistance of an entity is computed using a function called `resistance_of`. The implementation of this function is illustrated in Figure 5.33. The resistance of an entity depends on its material and geometry. The implementation presented in this figure is based on the normal calculation performed in Bourns to determine the resistance of a design entity.

In Figure 5.34 a relation is presented which characterises the meaning of the requirement that a scheme be end-to-end stackable. Essentially, a contact component can be regarded as being end-to-end stackable if the length of its substrate element is twice the contact pitch of the component.

```
relation end_to_end_stackable( scheme )
    =::= { S: exists substrate_entity(E):
                  is_a_part_of( E, S ) and
                  E.length = 2 * S.contact_pitch }.
```

Figure 5.34: The meaning of the end-to-end stackable requirement.

In this section various aspects of the Galileo implementation of a design knowledge-base for developing schemes for the Bourns case-study product was presented. It can be seen that the approach presented in this dissertation for doing so is very flexible and expressive. It has been demonstrated that the approach is capable of modelling all the necessary design knowledge encountered in Bourns related to the design of a contact system which meets the design specification. A full Galileo implementation of the Bourns design specification and design knowledge-base are presented in Appendix D.

5.2.6 Generating Alternative Schemes

In this section a number of schemes are described based on the design knowledge-base presented in the previous section. The development of three schemes will be presented graphically. The schemes discussed here are intended to satisfy the design specification presented in Section 5.2.4. The purpose of this section is to demonstrate that the approach presented in this dissertation could be used to develop schemes for an industrial product.

In the presentation of these schemes no screen-shots will be used to demonstrate the interaction between the constraint filtering system and the designer. This has already been demonstrated in Section 5.1.3. Also, a presentation of the use of Pareto optimality will not be given here, since this would require a detailed sequence of screen-shots and, in addition, has already been discussed in Section 5.1.3.

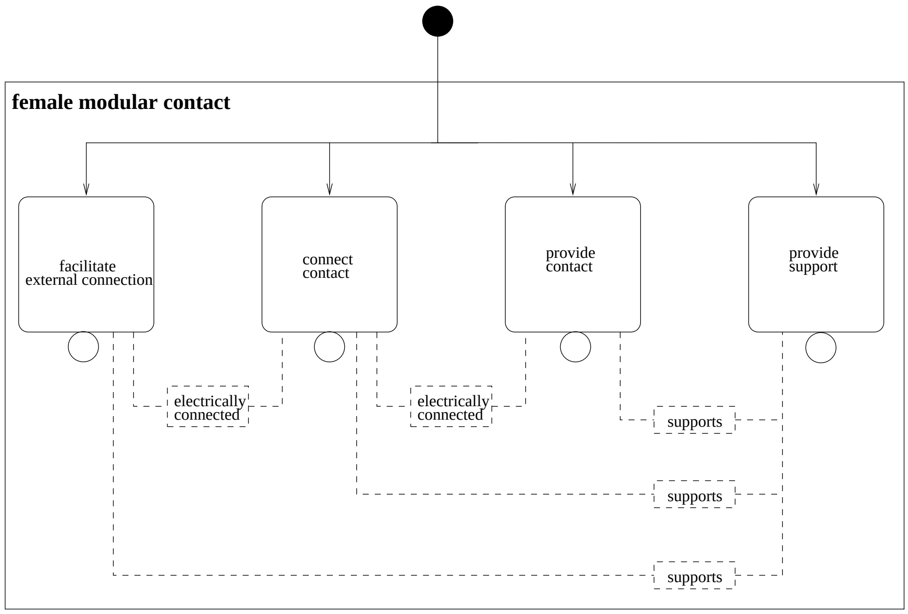


Figure 5.35: A graphical representation of the principle of a female modular contact.

In Figure 5.35 a graphical representation of the principle of a female modular contact is presented. The implementation of this principle is presented in Lines 30–48 of Section D.1. This principle requires that four embodiments are made in the design. These embodiments are associated with the functions *provide support*, *provide contact*, *connect contact* and *facilitate external connection*. There are a number of context relations defined between these functions, namely, that there is an *electrically connected* relation between the embodiments of the functions *provide contact* and *connect contact*, that there is an *electrically connected* relation between the embodiments of the functions *connect contact* and *facilitate external connection*, a *supports* relation between the embodiments of the functions *provide support* and *provide contact*, a *supports* relation between the embodiments of the functions *provide support* and *connect contact* and a *supports* relation between the embodiments of the functions *provide support* and *facilitate external connection*.

According to Lines 125–126 of Section D.1 it can be seen that the principle of a female modular contact can be used to provide the function *provide structured contact*.

In Figure 5.36 a number of design entities are presented graphically. Each of these design entities are implemented in Section D.1. For example, the molded body design entity is implemented in Lines 64–65.

By inspecting the relation defined on Lines 124–149 of Section D.1 it can be seen

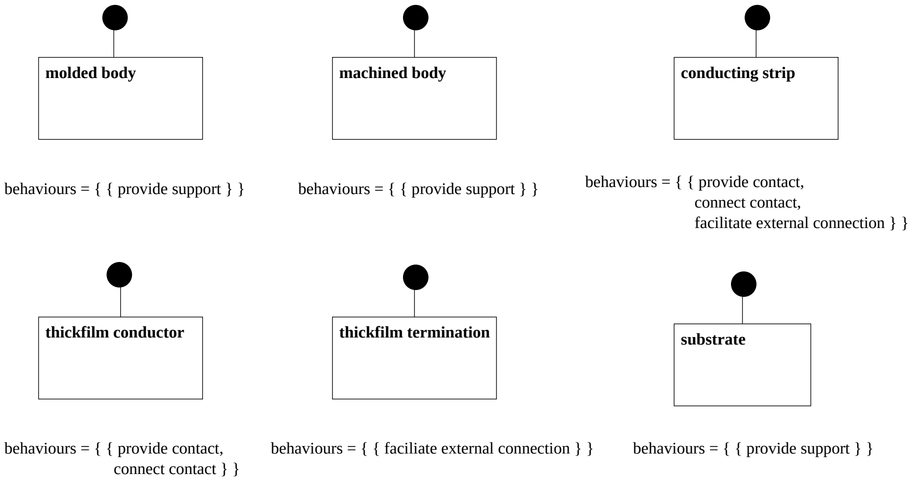


Figure 5.36: A graphical representation of the entities that will be used to develop schemes for the case-study product.

that a molded body can provide the function *provide support*, a machined body can provide the function *provide support*, a conducting strip can provide the functions *provide contact* and *facilitate external connection*, a thickfilm conductor can provide the function *provide contact*, a thickfilm termination can provide the function *facilitate external connection*, and a substrate can provide the function *provide support*.

Using the various means illustrated in Figure 5.35 and Figure 5.36 a number of schemes for the design specification presented in Section D.2 will be described.

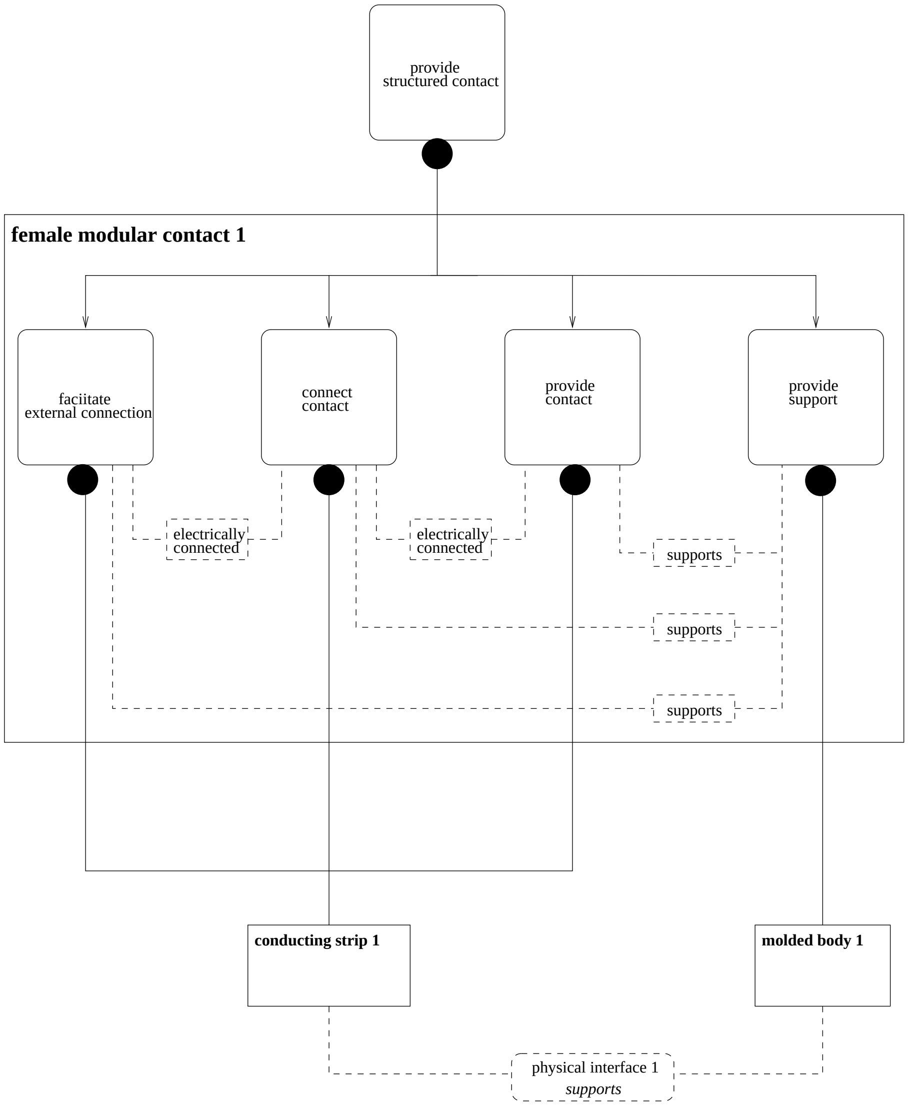


Figure 5.37: A graphical representation of the structure of a scheme which comprises a `molded body` and a `conducting strip`.

In Figure 5.37 a scheme is presented which could have been generated by interacting with a constraint-based filtering system using the Galileo modules presented in Appendix D. This scheme uses the design principle *female modular contact* to provide the function *provide structured contact*. As discussed already, this design principle requires the designer to consider four further embodiments in order to develop the scheme. The associated functions are *provide support*, *facilitate external connection*, *provide contact* and *connect contact*. The designer chooses to embody the function *provide support* with a *molded body* design entity and to embody the remaining three functions with the same design entity, namely, a *conducting strip*.

The *conducting strip* design entity is capable of providing three functions simultaneously: *provide contact*, *connect contact* and *facilitate external connection*.

Therefore, when the designer selects to provide the function *facilitate external connection* with a *conducting strip*, an instance of this design entity is introduced into the design due to the constraint defined in Lines 87–89 of Section D.1. When the designer then embodies the functions *connect contact* and *provide contact* with a *conducting strip*, no new entities are introduced into the design since the *conducting strip* entity introduced earlier can simultaneously provide all of these functions. This is due to the relation specified in Lines 124–149 of Section D.1.

The context relations that will need to be considered in the physical structuring of this scheme are the *supports* relation and the *electrically connected* relation. There will exist a *physical interface* between the *conducting strip* and the *molded body* entities. This *physical interface* will implement a supports relationship between these entities in other to embody the *supports* context relation. This is due to the relations defined on Lines 174–192 of Section D.1. Since the embodiment of the functions *facilitate external connection*, *provide contact* and *connect contact* is a single conductor design entity there is no requirement for an explicit interface for the *electrically connected* context relation. This is due to the relations defined on Lines 156–171 of Section D.1 and, in particular, Line 166.

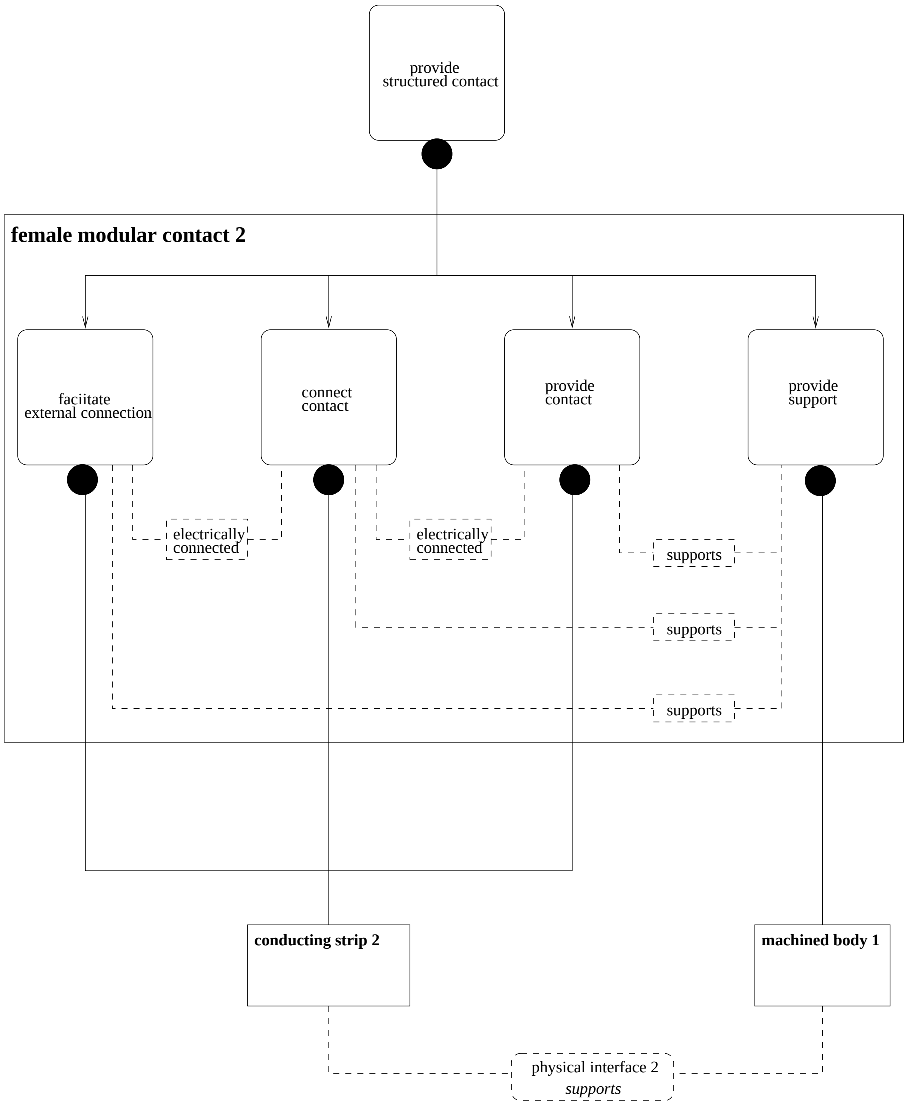


Figure 5.38: A graphical representation of the structure of a scheme which comprises a `machined body` and a `conducting strip`.

In Figure 5.38 a second scheme is presented. This scheme also uses the design principle *female modular contact* to provide the function *provide structured contact*. The designer chooses to embody the function *provide support* with a design entity called *machined body* and to embody the remaining three functions with the same design entity, namely, a *conducting strip*. Therefore, for similar reasons as for the scheme presented in Figure 5.37, the context relations that will need to be considered in the physical structuring of the scheme are the *supports* relation and the *electrically connected* relation. Again, there is only one interface required in this scheme. This is to implement the *supports* relation between the *conducting strip* and *machined body* design entities. No explicit interface is required for the *electrically connected* context relation.

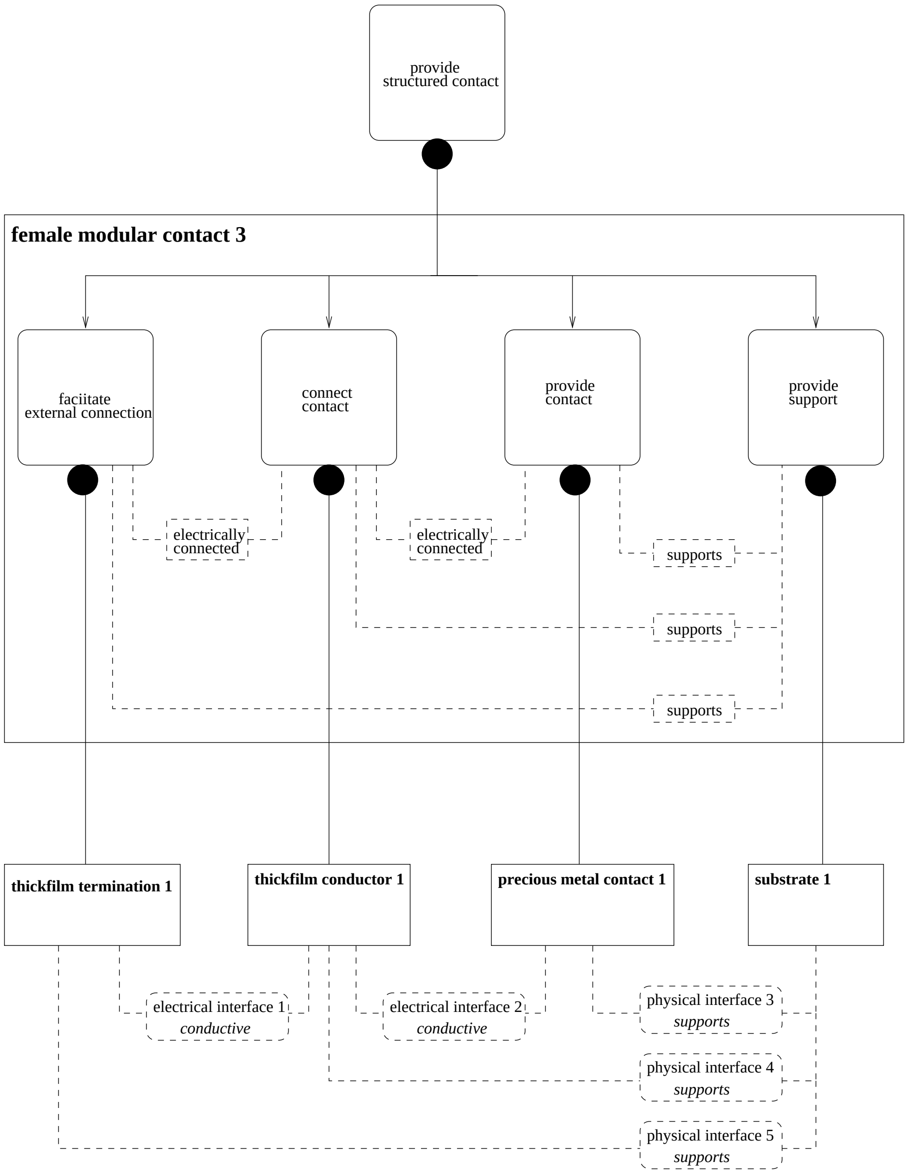


Figure 5.39: A graphical representation of the structure of a scheme which comprises a `substrate`, a `thick-film termination` and two distinct `thick-film conductor` entities.

In Figure 5.39 a third scheme is presented. This scheme also uses the design principle *female modular contact* to provide the function *provide structured contact.*

The function *provide support* was embodied with a *substrate* design entity, the function *facilitate external connection* was embodied with a *thick-film termination* design entity, the function *provide contact* was embodied with a *precious metal contact* design entity and the function *connect contact* was embodied with a *thick-film conductor* design entity.

In order to fulfill the various context relations required by the *female modular contact* design principle the designer must consider a number of relations amongst the design entities in this scheme. The *supports* context relation between the *substrate* and the *thick-film conductor* is implemented through a *physical interface* providing a supports relationship. The *supports* context relation between the *substrate* and the *precious metal contact* is implemented through a second *physical interface* providing a supports relationship. The *supports* context relation between the *substrate* and the *thick-film termination* is implemented through a third *physical interface* providing a supports relationship.

The *electrically connected* context relation between the *thick-film conductor* and the *precious metal contact* is implemented through an *electrical interface* providing a conductive relationship. The *electrically connected* context relation between the *thick-film conductor* and the *thick-film termination* is implemented through a second *electrical interface* providing a conductive relationship.

5.2.7 Review of the Industrial Case-Study

In the previous parts of Section 5.2 it was illustrated how the approach presented in this dissertation could be applied to support the design of a real-world product. The information for the case-study presented here was provided by an electronic component manufacturer based in Cork called Bourns Electronics. A brief profile of Bourns Electronics was given. The application of the case-study product was discussed in general terms and an informal design specification was presented. The knowledge that was required to design the product was presented and it was demonstrated how this can be formalised using the modelling approach presented in Chapter 4. A formal design specification for the case-study product was presented. The development of a number of alternative schemes for the case-study product was discussed.

While only three schemes have been discussed in this section many other schemes could have been generated. However, in order to respect the confidence of Bourns these schemes have not been presented here. Bourns were highly impressed with the approach. Indeed, one scheme that was developed using this approach was actually the same as a scheme which the company had developed independently and which they had recently patented: the scheme in question is not presented here at the request of the company. Bourns believes that a tool based on the approach presented here would be extremely valuable to them as a means for improving their new product development capability.

Chapter 6

Conclusions

In this dissertation a constraint-based approach to providing support to a human designer working during the conceptual phase of design has been presented. In this chapter the conclusions of the research will be discussed. This is done by relating the achieved results to the goals formulated in Chapter 1 and by discussing the results in the context of the thesis defended in this dissertation:

> *"It is possible to develop a computational model of, and an interactive designer support environment for, the engineering conceptual design process. Using a constraint-based approach to supporting conceptual design, the important facets of this phase of design can be modelled and supported."*

In this chapter a number of recommendations for further study are also presented.

6.1 The Goals of the Research Revisited

In Chapter 1 the goals of this research were presented; these were as follows:

1. To investigate whether constraint-based reasoning can be used as a basis for supporting conceptual design;

2. To determine if the expressiveness needs of conceptual design motivate the introduction of new features into Galileo.

It will be discussed in this section how these goals were achieved by the research presented in this dissertation.

6.1.1 Constraint-based support for conceptual design

In has been demonstrated in this dissertation how constraint-based reasoning can be used to support the human designer during conceptual design. In Chapter 3, a theory of conceptual design was presented upon which the research reported in this dissertation was based. In Chapter 4, a constraint-based implementation of this theory of conceptual design was presented. In Chapter 5, this approach was validated on two design problems: a toy design problem related to the design of a transportation vehicle, and an industrial case-study. Therefore, the goal to investigate whether constraint-based reasoning can be used as a basis for supporting conceptual design has been achieved, the answer provided by the investigation being positive.

6.1.2 Motivation for new features in Galileo

Although the pre-existing version of Galileo *could* have been used to support conceptual design, it was found that, by adding a few features to the language, program length could be reduced and the user-interface could be improved. These extensions to Galileo were discussed in Section 4.2 and related to: an enhancement to the semantics of quantification; the application of the "`!`" operator to the `exists` quantifier; the introduction of a new meta-level function, called `#parent_of`; and the introduction of a new keyword `hidden`. These extensions will be briefly discussed here.

In this dissertation, it was proposed that all quantified constraints should apply to *all* instances of the domain over which they are quantified, regardless of whether these instances are top-level parameters or merely fields of parameters.

It was also proposed in this dissertation that the `!` operator be applied to the `exists` quantifier. The operational semantics of its application, as proposed here, is that, if the requisite kind of parameter does not exist, the interpreter should automatically introduce a new parameter which satisfies the constraint.

While the standard version of Galileo already includes some meta-level relations and functions, the meta-level aspects of the language are still under development. The research reported in this dissertation motivated the need for a new meta-level function, called `#parent_of`. This function can be used to access any field of a structured parameter from any one of its other fields.

Finally, it was proposed that a new keyword, `hidden`, be introduced into the language. This keyword can be used to indicate which fields of a structured domain are visible on the user-interface of the constraint filtering system.

As already mentioned, the standard version of Galileo already includes some meta-level relations and functions. However, the meta-level aspects of the language are still under development and the range of relations and functions that currently exist is regarded as merely a first step in developing this aspect of the language. From the experience gained in carrying out the research presented in this dissertation a number of comments can be made regarding future developments for the meta-level aspects of Galileo.

It will now be shown that it would be desirable if the constraint filtering system could automatically derive the need for certain constraints from the presence, in a company-specific knowledge-base, of certain domain definitions. Such automatic inference would, however, need to be programmed into the filtering system. This could only be done by adding further meta-level constructs to Galileo.

In Chapter 4 an approach to implementing the various generic- and company-specific design concepts was presented. However, there is quite a high degree of commonality between certain aspects of the implementation of particular types of concepts. For example, consider Figure 4.53 and Figure 4.54. These figures are presented here as Figure 6.1 and Figure 6.2, without line numbers.

```
all embodiment(E): E.chosen_means = a_chassis implies
    !exists chassis( C ):
        must_be_directly_used_for( C, E.intended_function ).
```

Figure 6.1: The embodiment constraint presented in Figure 4.53.

```
all embodiment(E): E.chosen_means = a_molded_frame implies
    !exists molded_frame( M ):
        must_be_directly_used_for( M, E.intended_function ).
```

Figure 6.2: The embodiment constraint for a design entity presented in Figure 4.54.

It can be seen that these two constraints are structurally identical, except for the fact that a parameter of a different type is begin introduced. The domains `chassis` and `molded_frame` are specialisations of the domain `entity`. In addition, a convention has been used in this dissertation to relate a `known_means`, such as `a_chassis`, to an actual means, such as `chassis`. Therefore, it should be possible for the constraint filtering system to automatically infer a constraint from the definition of an entity. For example, consider the definition of an application-specific design entity, called `widget_entity`, which is a specialisation of the generic design entity, shown in Figure 6.3.

```
domain widget_entity
    =::= { W: entity(W) and ... }.
```

Figure 6.3: The definition of a specialisation of the generic design entity, called `widget_entity`.

From the entity definition presented in Figure 6.3, the constraint filtering system should be able to infer the necessary embodiment constraint. The constraint that should be inferred is shown in Figure 6.4.

```
all embodiment(E): E.chosen_means = a_widget_entity implies
    !exists widget_entity( W ):
        must_be_directly_used_for( W, E.intended_function ).
```

Figure 6.4: The inferred embodiment constraint that introduces an instance of the `widget_entity` design entity where required.

A similar process of automatically inferring embodiment constraints could also apply to design principles. As is the case for design entities, the embodiment constraints for design principles will be structurally similar. In Figure 4.50 an embodiment constraint for the `bicycle` design principle was shown. This constraint is shown, without line numbers, in Figure 6.5 for convenience.

```
all embodiment(E): E.chosen_means = a_bicycle implies
    !exists bicycle( B ):
        must_be_directly_used_for( B, E.intended_function ) and
        causes( E, B.e1 ) and
        causes( E, B.e2 ) and
        causes( E, B.e3 ) and
        causes( E, B.e4 ) and
        causes( E, B.e5 ).
```

Figure 6.5: The embodiment constraint for a design principle presented in Figure 4.50.

It should be possible for the filtering system to automatically infer this type of constraint from the definition of a principle. For example, consider the definition of an application-specific design principle, called widget_principle, which is a specialisation of the generic design principle, as shown in Figure 6.6.

```
domain widget_principle
    =::= { W: principle(W) and
              exists( W.e1 : embodiment ) and
              ...
              exists( W.en : embodiment ) and
              ... }.
```

Figure 6.6: The definition of a specialisation of the generic design principle, called widget_principle.

From the principle definition presented in Figure 6.6, the constraint filtering system should be able to infer the necessary embodiment constraint. The constraint that should be inferred is shown in Figure 6.7. Notice that the basic idea is the same as for the automatic derivation of the embodiment constraint for entities. The only difference is that a design principle introduces a number of embodiments into a scheme; the embodiment being fulfilled by the design principle causes these embodiments to be introduced.

```
all embodiment(E): E.chosen_means = a_widget_principle implies
    !exists widget_principle( W ):
        must_be_directly_used_for( W, E.intended_function ) and
        causes( E, W.e1 ) and
        ...
        causes( E, W.en ).
```

Figure 6.7: The inferred embodiment constraint that introduces an instance of the widget_principle design entity where required.

The addition of the capability to infer embodiment constraints from the definitions of means would dramatically reduce the size of the company- or application-specific knowledge-bases that must be developed by each company who wishes to use the approach presented in this dissertation to support conceptual design. This cannot be done at present. It would require the addition of further meta-level constructs to the language, a subject for further research.

6.2 Comparison with Related Research

The approach to supporting conceptual design presented in this dissertation is based on a combination of design theory, constraint processing techniques and Pareto optimality. In this section, the approach presented here will be compared with a number of approaches which have been reported in the literature. The approaches that are selected from the literature are regarded as being most similar to the approach presented in this dissertation and are categorised as being either design theory-driven (Section 6.2.1), constraint processing-driven (Section 6.2.2) or Pareto optimality-driven (Section 6.2.3).

6.2.1 Design Theory Approaches

The design theory upon which the approach presented in this dissertation is based assumes that products exist to provide some required functionality. There are a number of theories of design, such as "The Theory of Domains" [4] and "The General Procedural Model of Engineering Design" [86], that describe the parallelism between the decomposition of a functional requirement and the composition of a set of parts that fulfill that requirement.

The *function-means tree* approach to design synthesis is one approach that assists the designer is decomposing a functional requirement into an equivalent set of functions that can be provided by a set of known parts [28]. A function-means tree describes alternative ways of providing a top-level (root) function through the use of means. A means is a known approach to providing functionality. Two types of means can be identified in a function-means tree: principles and entities. A principle is defined by a collection of functions which, collectively, provide a particular functionality; it carries no other information than the lower-level functions to be used in order to provide a higher-level function. An entity represents a part or sub-assembly.

In the approach adopted here, the function-means tree concept was extended by adding context relations between the functions that define a design principle. This enables a computer to assist a designer to reason about the configuration of a set of design entities that obey the relationships that should exist between the functions in a design. It also helps to ensure that there is a valid mapping between the functional decomposition of a product and its physical composition in terms of parts.

The Scheme-Builder system [25, 125] uses function-means trees as a basis for structuring a design knowledge-base and generating schemes. The system interprets a function as an input-output transformation. The advantage of the system is that it is very systematic in terms of how functions are decomposed into sets of equivalent functions. However, its applications are limited to very highly parameterised design domains, such as mechatronics and control systems. The symbolic approach to representing function adopted in the research presented in this dissertation, coupled with the use of context relations in design principles, makes this approach far more flexible.

6.2.2 Constraint-Based Approaches

A number of systems have been developed for supporting aspects of conceptual design based on constraints. The "Concept Modeller" system was one of the earliest of such systems reported in the literature [137]. Aspects of the approach adopted in Concept Modeller were extended in a system called "Design Sheet" [27]. These systems focused on using constraint processing techniques to manage consistency within a constraint-based model of a design. In these systems conceptual designs are represented as systems of algebraic equations.

The approach presented in this thesis addresses a wider variety of issues that are crucial to successful conceptual design. The most important of these issues is design synthesis. In the approach presented here a designer is assisted in interactively synthesising a scheme for a design specification. In addition, a designer can develop multiple schemes for a design specification and be offered advice based on a comparison of these schemes. These are critical issues to supporting conceptual design which are not addressed in either Concept Modeller or Design Sheet.

The work presented in this dissertation builds on earlier work on interactive constraint processing for engineering design [22]. The earlier work focused on using constraint processing as a basis for interacting with a human designer who was working on a detailed model of design. The work presented in this dissertation builds on this work by demonstrating that the idea of using constraint processing as the basis for interacting with a human designer can be extended to support the development of a number of alternative schemes for a design specification from an initial statement of functional and physical requirements.

6.2.3 Pareto Optimality Approaches

As discussed in Section 2.3, the principle of Pareto optimality has been applied to a wide variety of problems in design. Most of these applications have used the principle of Pareto optimality in conjunction with evolutionary algorithms to generate a set of "good" design concepts [31, 70, 119]. These approaches focus on the automatic generation of design alternatives, an issue not of interest in the research presented here.

The use of the principle of Pareto optimality to monitor progress in design has been reported [123]. The approach focuses on the "tracking" of Pareto optimality to co-ordinate distributed engineering agents. Tracking Pareto optimality, in this case, means that the problem solver being used can automatically recognise Pareto optimality loss and the particular opportunity to improve the design. That approach inspired aspects of the approach to using Pareto optimality in the research presented in this dissertation. However, in this dissertation, Pareto optimality is used to compare two different schemes for a design specification rather than recognising when Pareto optimality is lost within an individual scheme. In this research, it is believed that the natural competition between designers can be harnessed to motivate improvements in the quality of schemes.

6.3 Recommendations for further study

This section presents a number of recommendations for further research in the area of constraint-aided conceptual design. These recommendations relate to:

1. Further prototyping and tool development;
2. The need for constraint filtering research;
3. The development of design knowledge-bases for different engineering domains;
4. An industrial-standard implementation;
5. CAD standard integration.

6.3.1 Further Prototyping and Tools Development

Although the pre-existing version of Galileo *could* have been used to support conceptual design, it was found that, by adding a few additional features to the language, program length could be reduced and the user-interface could be improved. Thus, in order to enable conceptual design support tools to be built, based on the approach presented in this dissertation, a new implementation of the Galileo filtering system incorporating the new features that have been proposed in this dissertation is required.

In order to make the process of design knowledge management more accessible to a user not familiar with constraint programming, one possible avenue for further study would be the development of tools which assist specification and maintenance of design knowledge. For example, a tool could be developed which would maintain consistency in the function-means tree database by ensuring that the various functions that are defined are mapped onto means for providing them. This tool would also be capable of assisting the human-user to define the various design principles, design entities, interfaces and evaluation utilities that are required for a particular company or engineering domain. Such a tool could be programmed using constraints, but it would require meta-level functions and relations.

6.3.2 Constraint Filtering Research

This research presented in this dissertation demonstrates that constraint filtering can be used to provide a high standard of designer support during conceptual phase of design. This phase of design can be characterised as a problem-solving process which is aimed at identifying a number of schemes for a design problem. Each of these schemes are defined by a relatively small number of variables and constraints, as opposed to the high number of variables and constraints used in a detailed design of a product. Therefore, conceptual design presents a more computationally modest problem to be addressed through constraint-filtering than is presented by more detailed phases of design. In addition, the benefits that can be realised by a company who have control over the conceptual design process are significant. Therefore, conceptual design represents an ideal problem domain to both the constraint processing research community and the industrial community.

Secondly, this research has demonstrated that there exists significant potential for designer support based on constraint filtering during the conceptual design process. Therefore, there is a significant practical justification for research which contributes to the efficiency of algorithms for constraint filtering. In particular, research which addresses consistency processing, constraint propagation, dynamic constraint satisfaction and constraint-based advice generation should be encouraged.

6.3.3 Knowledge-bases for Different Engineering Domains

This research has presented the design community with a evidence for the utility of a constraint-based approach to supporting one of the most difficult and important phases of product development. In order to exploit the approach presented in this dissertation, research that focuses on the development of conceptual design knowledge-bases should be encouraged. This research should attempt to identify generic knowledge in various engineering domains which relates to the development of constraint-based function-means databases that can be used in real-world design situations.

6.3.4 An Industrial Implementation

Another recommendation for further study relates to the development of a industrial-standard constraint-aided conceptual design system based on the research presented in this dissertation. This system should be capable of supporting the development of schemes in a graphical manner similar to existing CAD systems. Such work provides an interesting application for the use of visual constraint programming technologies.

A further enhancement of such a CAD system would be support for collaborative design. A CAD system capable of supporting collaborative conceptual design would be instrumental in assisting companies reap the benefits of effective conceptual design.

6.3.5 CAD Standard Integration

The final recommendation for further study relates to the integration of CAD standards with the approach to conceptual design presented in this dissertation. The possibility of using existing CAD standards, for example STEP, to translate the constraint-based models of schemes into a CAD system, such as AutoCAD or Pro-Engineer, for further design would significantly increase the utility of the approach in an industrial context.

6.4 Summary

As a summary of the conclusions of this dissertation, the central thesis of this research has been defended. Therefore, the following statement can be made:

> *"It is possible to develop a computational model of, and an interactive designer support environment for, the engineering conceptual design process. Using a constraint-based approach to supporting conceptual design, the important facets of this phase of design can be modelled and supported."*

Appendix A

An Overview of Galileo

This appendix presents an overview of the Galileo language. A brief discussion of the various Galileo implementations and tools is also presented. An example program is explained in detail before an example interaction is discussed. Much of the discussion presented here is adapted from the literature on Galileo [8, 22, 19]. This is done for the convenience of the reader. No claim is being made regarding the originality of the contents of this appendix.

A.1 The Galileo Language

In this research the constraint programming language Galileo has been used as the modelling language of choice. Galileo [22, 49] is a frame-based constraint programming language based on the First-Order Predicate Calculus (FOPC). Galileo offers designers a very rich language for describing the design of a product, the environment in which a product is being developed and the responsibilities of the various participants in an integrated product development environment [22, 49, 116].

A frame-based constraint programming language provides a designer with the expressiveness required to describe the various aspects of the design problem effectively. Frames can be used to represent the product being designed, the components from which it is configured or the materials from which it is made. Frames can also be used to describe the life-cycle environment in which the product will be manufactured, tested and deployed. Constraints between frames can be used to express the mutual restrictions between the objects in the design and the product's functionality, the component/material properties and the product life-cycle.

Amongst the many features of the Galileo language are the availability of predefined domains such as the real and integer numbers, arbitrary scalars and frame-like structured domains. It is possible to define sets and sequences in Galileo and structured domains can be defined and organised into inheritance hierarchies. The language comes with a number of predefined predicates such as equality and numeric inequality. There are a number of standard functions available which includes the complete range of arithmetic and trigonometric functions. There are also a number of set- and sequence-based predicates and functions available as standard. Compound constraints can be written using the standard logic connectives as well as the negation operator.

In Galileo constraints can be universally and/or existentially quantified. Quantifiers in Galileo can be nested arbitrarily. In Galileo the existence of certain parameters and constraints can be expressed as being dependent on certain conditions being met. This is due to the fact that Galileo is based on a generalisation of First-Order Logic known as First-Order Free Logic [24]. This is the means by which Dynamic Constraint Satisfaction Problems [107] can be easily modelled in Galileo.

A.2 Galileo Implementations and Tools

At present there are three computer tools available for processing Galileo programs. These tools range from interactive constraint filtering systems to compilation systems which generate code for developing constraint checking applications.

There are two implementations of interactive constraint filtering for the Galileo language. The first of these runs on the MS-DOS platform and was implemented in Prolog [22]. This implementation was developed in order to demonstrate that certain aspects of Concurrent Engineering could be readily supported through a constraint processing approach. The second constraint filtering system for Galileo was developed during a research project called CEDAS (an acronym for Concurrent Engineering Design Adviser System)[1]. This implementation of the filtering system was implemented using Haskell[2] and runs under the Solaris operating system.

The Galileo compilation system was also developed as part of the CEDAS project [49, 116]. This system is based on version 6 of the Galileo language and is implemented in Haskell. There are compilers currently available for the Solaris, HP-

[1]The CEDAS project was funded under the ESPRIT programme – the project number is 20501.
[2]Haskell is a non-strict, purely functional programming language [87].

UX and Windows NT operating systems. This compilation environment translates a Galileo specification of a set of design guidelines and a specification of a CAD database representation into a form which is appropriate for a particular CAD system. During the CEDAS project the Galileo compilation system was demonstrated using the Visula CAD system from Zuken-Redac [129].

A.3 Constraint Filtering and Galileo

In this section a brief description of constraint filtering in Galileo is presented. This presentation is adapted from the literature on Galileo [19].

A *constraint* restricts the values that may be assumed by a group of one or more *parameters*. A *constraint network* is a collection of such constraints [102].

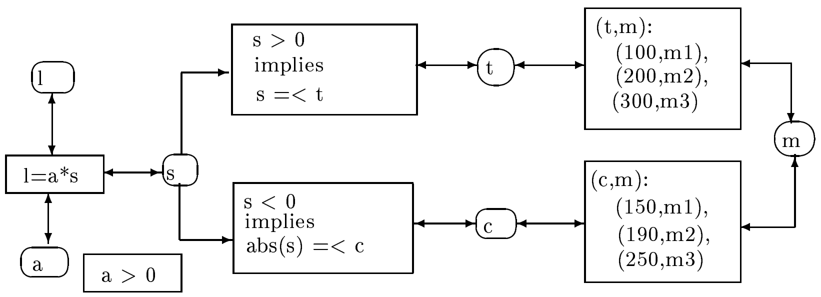


Figure A.1: Constraints on a load-carrying bar.

Consider the network shown graphically in Figure A.1, with parameters in oval nodes and constraints in rectangular nodes. The parameters l, a, s, m, t and c represent, respectively, the load on a bar, the cross-sectional area of the bar, the stress induced in the bar by the load, the material from which the bar is made, the tensile strength of this material, and its compressive-strength.

Constraint filtering is a form of processing which progressively restricts the ranges of possible values for parameters, by enforcing the restrictive effects of user-asserted constraints. This means that a constraint network can capture the impact of a decision — made concerning one phase of a product's life-cycle by an expert in that phase — on the other phases of the life-cycle. Suppose, for example, that a network represents the mutually constraining influences that exist between the functionality and production cost of a product. Such a network could, with equal ease, determine the impact of functionality decisions on production cost or, on the other hand, determine the impact of cost decisions on functionality.

Consider the network in Figure A.1 again. If, for example, a designer interacting with this network were to specify that the area of the bar is 2 and that the load to be carried by the bar will be greater than 240 (that is, if he were to impose the additional constraints that $a = 2$ and $l > 240$), the set of possible values for the tensile strength, s, would be restricted to half-open interval (120,300] and, since the tensile strength of $m1$ is only 100, the set of possible values for m would be restricted to $\{m2, m3\}$. This is an example of constraint filtering.

In the above situation, decisions about bar area and load resulted in restrictions on the material that could be used to make the bar. Restrictive influence could also flow in the opposite direction. Consider a scenario in which the above decisions made by a designer were followed by an engineer saying state that material $m3$ is unavailable; this could be expressed as the constraint $m <> m3$. As a consequence of this, m must be $m2$ and the set of possibilities for s is restricted to (120,200] which, in turn, means that the load l cannot exceed 400.

A.4 A Concurrent Engineering Application

In this section, a Galileo program called KLAUS2 [8], which is an advice system for printed wiring board design, will be discussed. Figure A.2 presents a modified version of KLAUS2, adapted from [8]. An example interaction with KLAUS2 will be discussed briefly in Section A.5. In this section, a number of extracts from KLAUS2 will be considered. Much of the discussion presented here is adapted from the literature on Galileo [8, 22]. This is done for the convenience of the reader. No claim is being made regarding the originality of the remainder of this appendix.

A Galileo program specifies a frame-based constraint network and comprises a set of declarations and definitions. The declaration statements include declarations of the parameters that exist in the network and the constraints that exist between these parameters. The definition statements include definitions for: partitions which divide the network into different regions that may be seen by different classes of user; application-specific domains, functions, and relations that are used in constraint declarations [8].

The statements in a Galileo program can be written in any order. In Figure A.2, the statements in the program have been listed in an order which facilitates the discussion here. The numbers which are used to annotate the program in Figure A.2

are not part of the program, but are present to faciliate discussion here.

```
1   'the facility where the board will be tested'(the_test_facility) : test_facility.
2   domain test_facility =::=
         ('the name of the equipment at'(name) : tester,
          'the maximum clock frequency testable at'(maxfreq) : frequency,
          'the maximum number of test points testable at'(maxtpoints) : counter).
3   domain tester =::= {'Dynamax Tester Mark I'(dynatest1),
                        'Dynamax Tester Mark II'(dynatest2),
                        'Erdsys TX'(erdsys)}.
4   domain frequency =::= {X : real(X) and 3.0 =< X =< 40.0}.
5   domain counter =::= {X : integer(X) and X >= 0}.
6   domain component =::= ('the type of'(type) : comp_type,
                           'the number of leads on'(nleads) : counter,
                           'the power consumed by'(power) : positive real).
7   domain comp_type =::= {processor,dsp,comm_cntrlr,divider,crystal,resistor,capacitor,diode}.
8   domain 'crystal oscillator'(osc_crystal) =::=
         {X : component(X) and X.type=crystal and
              exists(X.'the oscillation frequency of'(freq) : positive real)}.
9   domain 'frequency divider'(divider) =::= {X:component(X) and X.type=divider and ... }.
10  domain cpu =::= {X:component(X) and X.type=processor and ... }.
11  domain resistor =::= {X:component(X) and X.type=resistor and ... }.
12  all component(X): X.nleads in !lst_tpoints.
13  num_tpoints = sum(lst_tpoints).
14  num_tpoints =< the_test-facility.maxtpoints.
15  'the numbers of test points required by components'(lst_tpoints) : bag of counter.
16  'the total number of test points on the board'(num_tpoints) : counter.
17  not(exists tester(X): cost(X) < cost(the_test_facility.name) and satisfactory(X)).
18  function 'the total cost per hour of'(cost) -> real =::= {X -> 1.8 * price(X)}.
19  function 'the direct cost per hour of usage of'(price) -> real =::= datafile(qdb,TSTRCSTS).
20  relation 'capable of testing the board'(satisfactory) =::=
            {X : num_tpoints =< max_tpoints(X) and
                 (all osc_crystal(Y): Y.freq =< max_clock(X))}.
21  function 'the maximum number of nails usable by'(max_nails) -> integer =::= datafile(qdb,TSTRNALS).
22  function 'the clock limit for'(max_clock) -> real =::= datafile(qdb,TSTRCLKS).
23  all component(X): X.type in {processor,comm_cntrlr,dsp}
    implies exists(X.'the oscillator for'(crystal) : osc_crystal) and
            exists(X.'the pullup resistor for'(pullup) : resistor).
24  all osc_crystal(X):
    implies X.freq =< the_test_facility.maxfreq else
            exists(X.'an ancillary divider for'(a_d) : divider).
25  'the power consumptions of the components'(lst_powers) : bag of positive real.
26  'the total power consumed by the board'(total_power) : positive real.
27  all component(X): X.power in !lst_powers.
28  total_power = sum(lst_powers).
29  'the goal power consumption'(power_goal) : positive real.
30  'the maximum acceptable power consumption'(max_power) : positive real.
31  'the power consumption discrimination increment'(power_inc) : positive real.
32  total_power =< (power_goal to max_power step power_inc).
33  no_tpoints < 200.
34  hardness(no_tpoints < 200) = 0.9.
35  field 'the circuit designers perspective'(configuration) =::= {X : component(X)}.
36  field 'the perspective taken by test engineers'(testability) =::= {the_test_facility}.
37  permission({X:component(X)},configuration).
```

Figure A.2: Extracts from KLAUS2, a Galileo program.

A.4.1 Modelling with Frames

Frames can be used to represent the product being design, the components from which it is configured and the life-cycle machines being used to manufacture the product. For example, statement 1 of Figure A.2 declares the existence of a parameter called `the_test_facility` and specifies that it is of type `test_facility`. The string

[ ]Help [ ]File [ ]New [ ]Utilities [ ]Search [ ]Up [ ]Down [ ]Focus [ ]Toggle	
[ ]the name of the equipment at the facility where the board will be tested	Erdsys TX
[ ]the maximum clock frequency testable at the facility where the board will be tested	9.8
[ ]the maximum number of test points testable at the facility where the board will be tested	200
>>>	
KLAUS2 - a PWB Design Advisor (Testability)	

Figure A.3: An example of the KLAUS2 interface

delimited by apostrophes is a long synonym which will be displayed in all output given to users of the program[3]. Thus, it can be seen that this parameter represents `the facility where the board will be tested`.

This parameter is a frame, as can be seen in statement 2, which defines the application-specific domain `test_facility` referenced in statement 1. The statement specifies that a test facility is represented by a frame defined by three fields. For each field, its name, its long synonym and its domain are specified. In output given by the system to users, the long synonyms of the fields in a frame are concatenated with the long synonyms of the frame-valued parameters. This can be seen in Figure A.3, where the fields of the `the_test_facility` parameter are presented in a scrollsheet; thus, for example, the field `the_test_facility.maxfreq` is known as `the maximum clock frequency testable at the facility where the board will be tested`.

The fields of a `test_facility` frame, as specified in statement 2, have scalar application-specific domains. These domains are defined in statements 3, 4 and 5. Notice that, while scalar domains may be defined extensionally by listing each possible value (statement 3), they may also be defined intensionally by specifying an appropriate logical formula (statements 4 and 5). (Notice the use of long synonyms in statement 3; internally, the system will use values such as `erdsys`, but the user will see names like `Erdsys TX` – see Figure A.3.)

Frame-based inheritance is illustrated in statements 6, 7, 8, 9, 10 and 11. Statement 6 defines the frame domain `component`; for simplicity, only three fields are given. Notice, from statement 7, that there can be several different types of com-

[3]These synonym strings are optional and, if they are absent, the actual parameter names will appear on the user-interface – this is the case in Chapter 4 and Chapter 5.

ponent. By using inheritance, frames for these different types of component can be defined in terms of the generic component frame. In statement 8, for example, it is specified that an `osc_crystal` frame, which is used to represent a crystal oscillator, is a `component` frame which has the value `crystal` in its `type` field and which also has an additional field, `freq`, that represents the frequency at which the crystal oscillates. In statement 9, a `divider` frame, which is used to represent a frequency divider, is specified to be a `component` frame in which the `type` field has the value `divider` and which has several other fields – these additional fields are not specified here, however. A `cpu` frame is defined similarly in statement 10, and a `resistor` frame in statement 11.

A.4.2 Modelling with Constraints

In a frame-based constraint network, constraints can be used to represent the interdependencies that exist within the frame-based model of a product, and the life-cycle interdependencies that exist between the product and the life-cycle model.

Consider statement 12 in Figure A.2. This is a universally quantified constraint which specifies that the number of test-points required by each component on the board (which, for the purposes of this discussion, is taken as being equal to the number of leads on the component) must be recorded in a list. This list is referenced in statement 13, which states that the total number of test-points in the circuit being designed is the sum of the numbers of test-points required by the individual components. Statement 14 then expresses an important life-cycle interdependency between this total number of test-points and the intended test-facility. This constraint states that the total number of test-points on the board must be less than or equal to the maximum number of test-points testable at the facility where the board is being tested.

The two parameters `lst_tpoints` and `num_tpoints`, referenced in statements 12 and 13, are declared in statements 15 and 16 . Statement 15 illustrates another feature of the language – its ability to handle set-valued parameters. Indeed, the constraint uses a special kind of set, a *bag*, in which multiple copies of the same member are treated as distinct. This is needed in order to deal with the fact that different components can have the same number of leads.

While the life-cycle interdependency expressed in statement 12 required universal quantification, statement 17 shows a constraint which requires the other form of quantification in logic, that is, existential quantification. Statement 17 specifies that the cheapest tester available should be used, by specifying that there should not be any tester available which is capable of testing the board and which costs less than the chosen tester. This constraint is worth further discussion, because it illustrates usage of other features of Galileo that are only available because the language provides the full expressive power of the predicate calculus. This statement shows that while Galileo provides generic predicates such as $<$ and generic functions such as $*$, it also allows application-specific functions and predicates to be used.

The meaning of any application-specific functions and predicates must be defined. In Galileo, this can be done using either of the notions of extensional or intensional definition from set theory. Consider, for example, statement 18 which defines the meaning of the function `cost`. The meaning of a function is a set of mappings from inputs to outputs. In statement 18, the meaning of `cost` is defined intensionally; the total cost of using a tester is 1.8 times the direct cost of using the tester, which is denoted by the function symbol `price`. There is a finite number of possible testers, so the meaning of the `price` function is a finite set which can be defined extensionally.

Galileo allows extensional set definitions to be given either in the program text or in an external database file. Statement 19, for example shows that the meaning of the function `price` is defined by specifying that the set of pairs of values is in the database file TSTRCSTS. (It was found, in application experiments, that tying function and predicate definitions to database files is a very natural way of linking design adviser systems to corporate relational databases.)

As with a function, the meaning of a predicate is also a set, which can also be defined either extensionally or intensionally. In statement 20, the meaning of the predicate `satisfactory` is defined by specifying an intensional formula which uses universal quantification and two application-specific function symbols whose meanings are defined in statements 21 and 22.

A.4.3 Free-Logic and Non-Parametric Design

In parametric design, the overall architecture of the product and its life-cycle have already been determined and the task is merely one of establishing appropriate values for the parameters of this architecture. Design is not this simple. Parametric design must be accompanied by product structuring, in which the structure of the product or its life-cycle environment is determined.

Using the notion of conditional existence from free logic [98] enables a constraint processing inference engine to deduce that, when certain conditions are true, additional parameters must be introduced into a constraint network [24]. Consider, for example, statement 23. This is a universally quantified constraint which uses the notion of conditional existence from free logic to specify that if a parameter of domain `component` is used to represent a CPU, a communications controller or a digital signal processor device, then the parameter must have an extra slot to represent the oscillator which drives the device and must have a further slot to represent a pullup resistor. The `exists` tokens in this constraint are free logic existence specifiers, not existential quantifiers.

A more interesting usage of free logic appears in statement 24, which also uses modal logic. The `else` connective in this constraint comes from modal logic. The constraint specifies that, ideally, every crystal should oscillate at a frequency which does not exceed the maximum clock speed that is testable by the test facility. However, it then goes on to say that if this is not possible, then any crystal which oscillates at a faster frequency must have an ancillary divider circuit. Here, we see the constraint network extending itself by introducing a new parameter when a certain condition arises. This new parameter represents a new component, the necessity of whose existence has been inferred by the system. (This effect of this constraint will be seen in Section A.5.)

A.4.4 Optimization

Statement 17 used existential quantification to require that the cheapest possible tester be used. Since there is a finite number of possible testers, this statement illustrated optimization of a parameter which ranged over a discrete domain. Galileo also supports optimization of parameters which range over infinite domains, provided these domains are discretized into finite numbers of equivalence sets.

Consider, for example, statement 32. This specifies that the total power consumed by the board (see statements 25 through 28) should, ideally, not exceed the goal power consumption (statement 29), that it should certainly not exceed the maximum acceptable power consumption (statement 30), and that between those two values the optimization tolerance is equal to a value called the power consumption discrimination increment (statement 31).

A.4.5 Prioritization

By default, all constraints in a Galileo program are treated as being equally important, and are treated as hard constraints, that is, as constraints that must be satisfied. However, we can also specify that some constraints in a program are soft. The meaning of a constraint being soft is that the constraint should, if possible, be satisfied but, if there ever arises a situation in which violating the constraint would relieve an over-constrained situation, then it may be ignored. We can have as many soft constraints as we want in a Galileo program and can assign them different levels of hardness or priority. Constraint hardness is a number in [0,1], with 1 being the default and 0 being the hardness of a constraint that has been disabled completely. Statement 34 is a second-order constraint which specifies that the hardness of the first-order constraint in statement 33 is 0.9.

A.4.6 Multiple Perspectives and Interfaces

Galileo enables constraint networks to be divided into (possibly overlapping) regions called *fields of view*. A field of view is that region within a constraint network that is currently important to a user interacting with the network. A field of view can be either global or local. The global field of view consists of the entire constraint network. A local field of view contains only a subnetwork. Each field of view contains all the parameters that are of interest to the user, as well as all constraints which reference these parameters.

A field of view can be defined by specifying the set of parameters which it contains. In Figure A.2, for example, two of the fields of view that are provided by the KLAUS2 application are defined. Statement 35 defines a `configuration` field of view, which will be seen by a circuit designer, and specifies that it contains the set of all parameters of domain `component`. Statement 36 defines a `testability` field of view and specifies that its set of parameters contains just one parameter, `the_test_facility`.

A.4.7 Specifications and Decisions

Galileo programs are interactive. A user can augment the set of constraints in the initial network that is specified in a program, by inputting additional constraints to represent his design decisions. Thus, for example, if a test engineer decides to use an Erdsys TX tester, he can indicate this decision by inputting the following equational constraint: `the_test_facility.name` = `erdsys`. (Note that the test engineer would not have to type this constraint – the desired decision can be input by using a mouse to select appropriate options in a series of pull-up menus. Furthermore, because of the system's use of long synonyms, the engineer would think that he was entering the following decision: `the equipment at the facility where the board will be tested = Erdsys TX`. The test engineer need never know about such "unfriendly" tokens as `the_test_facility.name` or `erdsys`.)

Decisions like the above selection of a tester are parametric design decisions. A user can express architectural decisions by adding new parameters to the initial network that is defined by the program. Thus, for example, a circuit designer interacting with the KLAUS2 application can introduce new parameters to represent various parts of his evolving circuit. To introduce a CPU, for example, he can either introduce a parameter of domain `component` and specify that the `type` slot of this parameter has the value `processor`, or he can achieve exactly the same result, through frame-based inheritance, by introducing a parameter of domain `cpu`.

In Galileo, it can be specified which users of an application that supports multiple fields of view are allowed to introduce new parameters and what classes of parameters they are allowed to enter. Statement 37 of Figure A.2, for example, specifies that users of the `configuration` field of view are allowed to introduce parameters of domain `component` or of any domain (such as `osc_crystal`, `divider`, `resistor` or `cpu`) that is a sub-class of the `component` domain.

One further point should be made on the representation of design decisions in Galileo. Any syntactically well-formed Galileo constraint may be used to represent a design decision. We are not restricted to equations of the form seen above in the test engineer's selection of test equipment. Any sentence, atomic, compound or quantified, in first-order predicate calculus, including modal and free logic as well as classical logic, can be used to represent a design agent's decision.

A.4.8 Explanation

As well as specifying information by introducing new parameters and new constraints, the user of a Galileo program can ask for information. He can, for example, ask for the range of allowable values for any of the parameters in a network. He can also ask for justifications for these ranges – whenever the range of allowable values for a parameter is reduced by a constraint, the rationale for this reduction is noted by the run-time system as a dependency record which can be accessed later for explanation purposes. We will see an example of explanation in the example interaction in Section A.5.

A.4.9 "What If" Design Reasoning

A user can always withdraw any constraint or parameter that he has added. Thus, by introducing and withdrawing constraints and parameters, the user can investigate "what if" scenarios.

A.5 An Example Interaction

To illustrate how programs written in Galileo support design, it will be shown how the KLAUS2 application that we considered above supports interaction with two members of the product development team and how it provides system-mediated negotiation when these perspectives lead to conflicting decisions. This will be done by considering an example scenario.

Several points deserve emphasis about this scenario. First, this scenario presents only one of very many possible orders of interaction with KLAUS2. Second, it shows only a few steps in this interaction.

In the following scenario, it is assumed that the project leader has set up a database entry for a new project. The story is taken up when the test engineer, who in this case is the first team member to make some decisions about this new project, starts to interact with KLAUS2 about the project. Figure A.3 shows the interface presented by KLAUS2 to the test engineer after he has selected the test equipment to be used for the project. The largest window in this screen is a single-column spreadsheet, or "scrollsheet," in which each cell occupies one or more lines.

Figure A.4 shows the perspective of the circuit designer. This screen represents

```
[ ]Help    [ ]File    [ ]New    [ ]Utilities    [ ]Search
[ ]Up    [ ]Down    [ ]Focus    [ ]Toggle
---------------------------------------------------------
[ ]the cpu
[ ]the oscillator for the cpu
[ ]the pullup resistor for the cpu
---------------------------------------------------------
A crystal oscillator has been introduced
A pullup resistor has been introduced
>>>
---------------------------------------------------------
KLAUS2 - a PWB Design Advisor (Circuit Designer)
```

Figure A.4: The perspective of the circuit designer.

the circuit after the designer has introduced a CPU, and KLAUS2 has used its constraint information to induce automatically the need for an associated crystal and pullup resistor (see statement 23 in Figure A.2), and the designer has decided to accept these suggested components as part of his design.

Suppose that the designer, having accepted the crystal, specifies that it should oscillate at 25 Mhz. Now, however, the following constraint in the KLAUS2 program (statement 24 in Figure A.2) comes into play:

```
all X : osc_crystal(X)
implies X.freq =< the_test_facility.maxfreq else
        exists(X.'an ancillary divider for'(a_d) : divider).
```

and, because 25 Mhz exceeds the maximum testable frequency of 9.8 Mhz specified earlier by the test engineer (Figure A.3), the system introduces an ancillary divider circuit for this oscillator. The result can be seen in Figure A.5, where KLAUS2 is suggesting that the new component on the screen should be added to the circuit.

```
[ ]Help    [ ]File    [ ]New    [ ]Utilities    [ ]Search
[ ]Up    [ ]Down    [ ]Focus    [ ]Toggle
---------------------------------------------------------
[ ]the cpu
[ ]the oscillator for the cpu
[ ]the pullup resistor for the cpu
[ ]an ancillary divider for the oscillator for the cpu
---------------------------------------------------------
>>> cpu1.crystal.freq = 25
A divider has been introduced
>>>
---------------------------------------------------------
KLAUS2 - a PWB Design Advisor (Circuit Designer)
```

Figure A.5: A `divider` has been introduced.

The designer is surprised by the introduction of this component, so he asks KLAUS2 to justify it. In KLAUS2's explanation (Figure A.6), the constraint given above, which introduced the component, is paraphrased in natural language; note the use in this paraphrase of the natural language synonym "`an ancillary divider for`" defined in the constraint for the field "`a_d`". (The number (49) associated with this constraint in the explanation is system-generated, to enable easy reference to this constraint by the user.)

```
Justification
an ancillary divider for the oscillator for the cpu exists
because of the following constraint
      (49) every crystal oscillator must satisfy the following:
            ideally, the oscillation frequency of the crystal oscillator =<
            the maximum frequency testable at the facility
            where the board will be tested;
            otherwise, an ancillary divider for the crystal oscillator
            must exist and must be a frequency divider;
and because of the following parameter value(s):
      the maximum clock frequency testable at the facility where
             the board will be tested = 9.8;
      the oscillation frequency of the oscillator for the cpu = 25.

the maximum clock frequency testable at the facility where the board will
      be tested was established according to the perspective taken by
      test engineers.

the oscillation frequency of the oscillator for the cpu = 25
because you said so.
```

Figure A.6: The justification for the existance of the `divider`.

Despite this justification, the designer decides to reject this ancillary component. Now, however, the constraint which wanted to introduce the ancillary divider is violated, leading to the message shown in Figure A.7.

Choosing among the suggestions offered in this message, the designer decides to disable constraint (49). However, this constraint also refers to a parameter in the test perspective, so the decision to disable the constraint must be accepted by the test engineer. Whenever a user disables a constraint other than one he previously asserted himself, he is required to enter a free-text explanation of his action, which is saved for possible use in a design audit. These free-text explanations are also used in system-mediated negotiation between users of different perspectives.

When the test engineer next logs into KLAUS2, he is told that a constraint of interest to him was disabled by another user. Checking on this, he calls up the design state from the database. The system tells him which constraint was disabled and produces the free-text explanation given by the circuit designer.

```
VIOLATION:
an ancillary divider for the oscillator for the cpu
should exist, but is prohibited.

SUGGESTIONS:
(1) Retract the constraint
        (97) an ancillary divider for the oscillator for the cpu
                  should not exist.
(2) Disable the constraint
        (49) every crystal oscillator must satisfy the following:
                 ideally, the oscillation frequency of the crystal oscillator =<
                 the maximum frequency testable at the facility
                 where the board will be tested;
                 otherwise, an ancillary divider for the crystal oscillator
                 must exist and must be a frequency divider.
(3) Change the oscillation frequency of the oscillator for the cpu.
(4) Request that, in the perspective taken by test engineers,
        a change be made to
                  the maximum clock frequency testable at the facility where
                  the board will be tested.
```

Figure A.7: The violation message.

The test engineer decides that he is unwilling to allow this constraint to be disabled because of the difficulty in testing that would result. However, to compromise, he changes the test equipment to one which is able to handle a frequency of 25 MHz. After making this change, the test engineer reactivates the disabled constraint (49). Because of the higher frequency testable by the test equipment, the re-enabled constraint does not attempt to re-introduce an ancillary divider circuit, so no constraint violation occurs. The test engineer saves the new design state and starts to work on another project.

When the circuit designer next logs in, he is told that the test engineer has re-enabled the constraint but, since the unwanted divider circuit has not reappeared, the designer is content.

Suppose, however, that no such easy compromise was possible. For example, the designer might have selected an oscillation frequency that exceeded even the upper limit of the fastest available tester. In this case, the test engineer and designer will successively disable and reenable their shared constraint (49), offering free-text explanations to each other until one or the other gives way or appeals to the project leader. In this case, the test engineer could give give way by deciding to build a special divider test fixture; the circuit designer could give way by using a lower oscillation frequency.

Appendix B

Generic Design Concepts

This appendix presents the Galileo implementation of the various generic design concepts used in this dissertation to support conceptual design. The implementation of these concepts is discussed in detail in Chapter 4. Two Galileo modules are presented in this appendix. The first module is called `generic_concepts.gal`*. The second module is called* `comparison.gal`*. Both of these files contain the generic repetoire of concepts that a company can build upon for their own design projects.*

B.1 The contents of `generic_concepts.gal`

The `generic_concepts.gal` module contains Galileo implementations of the various generic concepts that are required to support reasoning during conceptual design. These concepts provide a basis for a particular company to define a repetoire of functions, design principles, design entities and evaluation functions that can be used to develop new products based on technologies known to the company. This module is organised as follows. Domain definitions and constraints quantified over these domains are presented first, in alphabetical order of the domain names. Relation definitions are then presented, followed by function definitions, both in alphabetical order.

```
module generic_concepts.

domain embodiment
    =::= ( hidden scheme_name : string,
           intended_function  : func,
           chosen_means       : known_means,
           reasons            : set of func_id ).

all embodiment(E):
    can_be_used_to_provide( E.chosen_means, E.intended_function ).

domain entity
    =::= { E: means(E) and
              E.type = an_entity and
              exists ( E.id : entity_id ) }.

all entity(E):
    has_a_unique_id(E).

domain entity_id
    =::= { I: positive integer(I) }.

domain func
    =::= ( verb : string,
           noun : string,
           id   : func_id ).

all func(F):
    has_a_unique_id(F).

domain func_id
    =::= { I: nonnegative integer(I) }.

domain interface
    =::= ( hidden scheme_name : string,
           entity_1 : entity_id ,
           entity_2 : entity_id ).

all interface(I):
    exists entity(E1), entity(E2):
        I.entity_1 = E1.id and
        I.entity_2 = E2.id and
        is_in_the_same_scheme_as( I, E1 ) and
        is_in_the_same_scheme_as( I, E2 ).

domain means
    =::= ( hidden scheme_name : string,
           type               : means_type,
           funcs_provided     : set of func_id ).
```

```
all means(M):
    is_a_possible_behaviour_of( M.funcs_provided, M ).

domain means_type
    =::= { a_principle, an_entity }.

domain principle
    =::= { P: means(P) and
              P.type = a_principle }.

domain scheme
    =::= ( scheme_name : string,
           structure   : embodiment ).

all scheme( S ):
    is_the_first_embodiment_in( S.structure, S.scheme_name ).

alldif scheme(S1), scheme(S2):
    not ( S1.scheme_name = S2.scheme_name ).

relation can_be_used_to_provide( known_means, func )
    =::= { (K,F): can_simultaneously_provide(K,Fs) and F in Fs }.

relation can_provide( set of func, string, set of func_id )
    =::= { (B,Sn,IDs): Fs = functions_identified_by( Sn, IDs ) and
                     Fs subset B }.

relation causes( embodiment, embodiment )
    =::= { (P,E): is_in_the_same_scheme_as( P, E ) and
                  E.reasons = {P.intended_function.id} union P.reasons }.

relation contains_only_one_entity_with_the_id( string, entity_id )
    =::= { (Sn,I): not existsdif entity(E1), entity(E2):
                      is_in_the_scheme(E1,Sn) and
                      is_in_the_scheme(E2,Sn) and
                      I = E1.id and I = E2.id }.

relation contains_only_one_func_with_the_id( string, func_id )
    =::= { (Sn,I): not existsdif func(F1), func(F2):
                      is_in_the_scheme(F1,Sn) and
                      is_in_the_scheme(F2,Sn) and
                      I = F1.id and I = F2.id }.

relation derives_from( entity, embodiment )
    =::= { (M,E): is_directly_used_for( M, E.intended_function ) or
                  is_indirectly_used_for( M, E.intended_function ) }.
```

```
relation has_a_unique_id( entity )
    =::= { E: exists positive integer(I): I = E.id and
              contains_only_one_entity_with_the_id( E.scheme_name, I) }.


relation has_a_unique_id( func )
    =::= { F: exists nonnegative integer(I): I = F.id and
         contains_only_one_func_with_the_id( #parent_of(F).scheme_name, I) }.


relation is_a_part_of( means, scheme )
    =::= { (M,S): M.scheme_name = S.scheme_name }.

relation is_a_possible_behaviour_of( set of func_id, means )
    =::= { (IDs,M): KM = known_means_of(M) and
                    can_simultaneously_provide( KM, B ) and
                    can_provide( B, M.scheme_name, IDs ) }.

relation is_a_reason_for( func, embodiment )
    =::= { (F,E): F.id in E.reasons }.

relation is_directly_used_for( means, func )
    =::= { (M,F): is_in_the_same_scheme_as( M, F ) and
                  F.id in M.funcs_provided }.

relation is_in_the_same_scheme_as( embodiment, embodiment )
    =::= { (E1,E2): E1.scheme_name = E2.scheme_name }.

relation is_in_the_same_scheme_as( embodiment, func )
    =::= { (E,F): E.scheme_name = #parent_of( F ).scheme_name }.

relation is_in_the_same_scheme_as( func, func )
    =::= { (F1,F2): #parent_of( F1 ).scheme_name
                     = #parent_of( F2 ).scheme_name }.

relation is_in_the_same_scheme_as( interface, means )
    =::= { (I,M): I.scheme_name = M.scheme_name }.

relation is_in_the_same_scheme_as( means, embodiment )
    =::= { (M,E): M.scheme_name = E.scheme_name }.

relation is_in_the_same_scheme_as( means, func )
    =::= { (M,F): M.scheme_name = #parent_of( F ).scheme_name }.

relation is_in_the_same_scheme_as( means, means )
    =::= { (M1,M2): M1.scheme_name = M2.scheme_name }.

relation is_in_the_scheme( embodiment, string )
    =::= { (E,Sn): Sn = E.scheme_name }.
```

```
relation is_in_the_scheme( func, string )
    =::= { (F,Sn): Sn = #parent_of(F).scheme_name }.

relation is_in_the_scheme( means, string )
    =::= { (M,Sn): S = M.scheme_name }.

relation is_indirectly_used_for( means, func )
    =::= { (M,F): exists embodiment(E):
                      is_a_reason_for( F, E ) and
                      is_directly_used_for( M, E.intended_function ) } .

relation is_the_first_embodiment_in( embodiment, string )
    =::= { (E,Sn): is_in_the_scheme( E, Sn ) and
                  E.intended_function.id = 0 and
                  E.reasons = {} }.

relation must_be_directly_used_for( means, func )
    =::= { (M,F): is_in_the_same_scheme_as( M, F ) and
                  F.id in !M.funcs_provided }.

relation provides_the_function( func, string, string )
    =::= { (F,V,N): V = F.verb and N = F.noun }.

relation is_the_reason_for( func, embodiment )
    =::= { (F,E): is_in_the_same_scheme_as( E, F ) and
                  F = E.intended_function and
                  E.reasons = {F.id} }.

relation is_used_to_embody_the_functionality_of( means, embodiment )
    =::= { (M,E): E.chosen_means = known_means_of( M ) and
                   is_directly_used_for( M, E.intended_function ) }.

function functions_identified_by( string,
                                  set of func_id ) -> set of func
    =::= { (Sn,IDs) -> { F | exists func(F):
                              exists I in IDs: F.id = I and
                              is_in_the_scheme(F,Sn) } }.
```

B.2 The contents of `comparison.gal`

The module `comparison.gal` contains a number of Galileo domains, relation and constraint definitions that can be used to compare alternative schemes using a numeric preference. These definitions provide a basis for using the principle of Pareto optimality to compare a set of alternative schemes against a set of preference criteria.

```
module comparison.

domain intention
    =::= { minimal, maximal }.

domain preference
    =::= ( value  : real,
           intent : intention ).

relation better_than( preference, preference )
    =::= { (P1,P2): P1.intent = minimal and
                    P2.intent = minimal and P1.value < P2.value,
           (P1,P2): P1.intent = maximal and
                    P2.intent = maximal and P1.value > P2.value }.

alldif scheme(S1), scheme(S2):
    not dominates( S1, S2 ).

relation dominates( scheme, scheme )
    =::= { (S1,S2): improves_on( S1, S2 ) and
                    not improves_on( S2, S1 ) }.
```

Appendix C

An Illustrative Example

This appendix presents all the Galileo code used in the toy running example related to the design of a product which provides the function `provide transport` *and which exhibits a required set of physical properties. This example is discussed in detail in Section 5.1. Aspects of the code presented here are also used as illustrative examples in the discussion in Chapter 4. Two Galileo modules are presented. The first module, called* `raleigh_knowledge.gal`*, can be regarded as a database of design knowledge that can be used to design vehicles. The second module, called* `vehicle_spec.gal`*, contains the full Galileo implementation of the specification for the required product. Using these two modules in conjunction with the modules presented in Appendix B, designers can be supported in developing a set of alternative schemes to meet the design specification for the product.*

C.1 The contents of `raleigh_knowledge.gal`

The `raleigh_knowledge.gal` module can be regarded as a database of design knowledge that can be used to design "leisure vehicles". Using this module in conjunction with the the `vehicle_spec.gal` module presented in Section C.2 and the modules presented in Appendix B, designers can be supported in developing a set of alternative schemes to meet the design specification of the product. This module is organised as follows. Domain definitions and constraints quantified over these domains are presented first, in alphabetical order of the domain names. Relation definitions are then presented, followed by function definitions, both in alphabetical order.

```
module raleigh_knowledge.

import generic_concepts.

domain air_cushion
    =::= { A: raleigh_entity(A) }.

domain axle
    =::= { A: raleigh_entity(A) }.

domain bicycle
    =::= { B: principle(B) and
           exists( B.e1 : embodiment ) and
           exists( B.e2 : embodiment ) and
           exists( B.e3 : embodiment ) and
           exists( B.e4 : embodiment ) and
           exists( B.e5 : embodiment ) and
           provides_the_function( B.e1.intended_function,
                                  'facilitate', 'movement' ) and
           provides_the_function( B.e2.intended_function,
                                  'provide', 'energy' ) and
           provides_the_function( B.e3.intended_function,
                                  'support', 'passenger' ) and
           provides_the_function( B.e4.intended_function,
                                  'change', 'direction' ) and
           provides_the_function( B.e5.intended_function,
                                  'provide', 'support' ) and
           drives( B.e2, B.e1 ) and
           supports( B.e5, B.e1) and
           supports( B.e5, B.e2) and
           supports( B.e5, B.e3) and
           supports( B.e5, B.e4) }.

domain chassis
    =::= { C: raleigh_entity(C) }.

domain chain
    =::= { C: raleigh_entity(C) }.

domain engine
    =::= { E: raleigh_entity(E) }.

domain frame
    =::= { F: raleigh_entity(F) }.

domain handlebar_assembly
    =::= { H: raleigh_entity(H) }.
```

```
domain harness
    =::= { H: raleigh_entity(H) }.

domain known_means
    =::= { an_axle, a_bicycle, a_skateboard, a_wheel_assembly,
           a_saddle, a_chain, a_chassis, a_steering_assembly,
           an_engine, a_harness, an_air_cushion, a_frame,
           a_handlebar_assembly, a_molded_frame,
           a_pedal_assembly, a_chain }.

domain mechanical_interface
    =::= { S: raleigh_interface(S) and S.type = mechanical and
              exists( S.relationship : mechanical_relationship ) }.

domain mechanical_relationship
    =::= { controls, drives, supports }.

domain molded_frame
    =::= { M: raleigh_entity(P) }.

domain pedal_assembly
    =::= { P: raleigh_entity(P) }.

domain raleigh_entity
    =::= { R: entity(R) and
              exists( R.width  : real ) and
              exists( R.mass   : real ) and
              exists( R.material : raleigh_material ) and
              E.mass = mass_of( E ) }.

domain raleigh_interface
    =::= { I: interface(I) and
              exists( I.type : raleigh_interface_type ) }.

domain raleigh_interface_type
    =::= { spatial, mechanical }.

domain raleigh_material
    =::= { cfrp, titanium, aluminium, steel }.

domain saddle
    =::= { S: raleigh_entity(S) }.

domain skateboard
    =::= { S: raleigh_entity(S) }.

domain spatial_interface
    =::= { S: raleigh_interface(S) and S.type = spatial and
              exists( S.relationship : spatial_relationship ) }.
```

```
domain spatial_relationship
    =::= { above, beside, under }.

domain steering_assembly
    =::= { S: raleigh_entity(S) }.

domain wheel_assembly
    =::= { W: raleigh_entity(W) }.

all embodiment(E): E.chosen_means = an_air_cushion implies
    !exists air_cushion( A ):
        must_be_directly_used_for( A, E.intended_function ).

all embodiment(E): E.chosen_means = an_axle implies
    !exists axle( A ):
        must_be_directly_used_for( A, E.intended_function ).

all embodiment(E): E.chosen_means = a_bicycle implies
    !exists bicycle( B ):
        must_be_directly_used_for( B, E.intended_function ) and
        causes( E, B.e1 ) and
        causes( E, B.e2 ) and
        causes( E, B.e3 ) and
        causes( E, B.e4 ) and
        causes( E, B.e5 ).

all embodiment(E): E.chosen_means = a_chain implies
    !exists chain( C ):
        must_be_directly_used_for( C, E.intended_function ).

all embodiment(E): E.chosen_means = a_chassis implies
    !exists chassis( C ):
        must_be_directly_used_for( C, E.intended_function ).

all embodiment(E): E.chosen_means = an_engine implies
    !exists engine( En ):
        must_be_directly_used_for( En, E.intended_function ).

all embodiment(E): E.chosen_means = a_frame implies
    !exists frame( F ):
        must_be_directly_used_for( F, E.intended_function ).

all embodiment(E): E.chosen_means = a_handlebar_assembly implies
    !exists handlebar_assembly( H ):
        must_be_directly_used_for( H, E.intended_function ).

all embodiment(E): E.chosen_means = a_harness implies
    !exists harness( H ):
        must_be_directly_used_for( H, E.intended_function ).
```

```
all embodiment(E): E.chosen_means = a_molded_frame implies
    !exists molded_frame( M ):
        must_be_directly_used_for( M, E.intended_function ).

all embodiment(E): E.chosen_means = a_pedal_assembly implies
    !exists pedal_assembly( P ):
        must_be_directly_used_for( P, E.intended_function ).

all embodiment(E): E.chosen_means = a_saddle implies
    !exists saddle( S ):
        must_be_directly_used_for( S, E.intended_function ).

all embodiment(E): E.chosen_means = a_skateboard implies
    !exists skateboard( S ):
        must_be_directly_used_for( S, E.intended_function ).

all embodiment(E): E.chosen_means = a_steering_assembly implies
    !exists steering_assembly( S ):
        must_be_directly_used_for( S, E.intended_function ).

all embodiment(E): E.chosen_means = a_wheel_assembly implies
    !exists wheel_assembly( W ):
        must_be_directly_used_for( W, E.intended_function ).

relation can_simultaneously_provide( known_means, set of func )
    =::= { (an_air_cushion,{F}):
                provides_the_function( F, 'facilitate', 'movement' ),
            (an_axle,{F}):
                provides_the_function( F, 'punch', 'holes' ),
            (an_axle,{F1,F2}):
                provides_the_function( F1, 'support', 'wheel' ) and
                provides_the_function( F2, 'facilitate', 'rotation' ),
            (a_bicycle,{F}):
                provides_the_function( F, 'provide', 'transport' ),
            (a_chasis,{F}):
                provides_the_function( F, 'provide', 'support' ),
            (an_engine,{F}):
                provides_the_function( F, 'provide', 'energy' ),
            (a_frame,{F}):
                provides_the_function( F, 'provide', 'support' ),
            (a_handlebar_assembly,{F}):
                provides_the_function( F, 'change', 'direction' ),
            (a_molded_frame,{F1,F2}):
                provides_the_function( F1, 'provide', 'support' ) and
                provides_the_function( F2, 'support', 'passenger' ),
            (a_harness,{F}):
                provides_the_function( F, 'support', 'passenger' ),
            (a_pedal_assembly,{F}):
                provides_the_function( F, 'provide', 'energy' ),
```

```
            (a_saddle,{F}):
                provides_the_function( F, 'support', 'passenger' ),
            (a_skateboard,{F}):
                provides_the_function( F, 'provide', 'transport' ),
            (a_steering_assembly,{F}):
                provides_the_function( F, 'change', 'direction' ),
            (a_wheel_assembly,{F}):
                provides_the_function( F, 'facilitate', 'movement' ) }.


relation drives( embodiment, embodiment )
    =::= { (E1,E2): drives( { X | exists entity( X ):
                                       derives_from( X, E1 ) },
                            { Y | exists entity( Y ):
                                       derives_from( Y, E2 ) } ) }.

relation drives( set of entity, set of entity )
    =::= { (E1s,E2s): exists E1 in E1s, E2 in E2s:
                            drives( E1, E2 ) }.

relation drives( entity, entity )
    =::= { (P,W): pedal_assembly(P) and wheel_assembly(W) and
                  is_in_the_same_scheme_as( P, W ) and
                  !exists chain(C):
                      is_in_the_same_scheme_as( P, C ) and
                      !exists mechanical_interface(M1):
                          M1.entity1 = P.id and
                          M1.entity2 = C.id and
                          M1.relationship = drives and
                      !exists mechanical_interface(M2):
                          M2.entity1 = W.id and
                          M2.entity2 = C.id and
                          M2.relationship = drives }.

relation overlaps( entity, entity )
    =::= { (E1,E2): is_in_the_same_scheme_as( E1, E2 ) and
                    !exists spatial_interface(S):
                        is_in_the_same_scheme_as( S, E1 ) and
                        S.entity_1 = E1.id and
                        S.entity_2 = E2.id and
                        S.relationship = above or
                        S.relationship = under }.

relation recyclable( scheme )
    =::= { S: all entity(E): is_in_the_scheme(E,S) implies
                  recyclable(E) }.

relation recyclable( entity )
    =::= { E: recyclable_material( E.material ) }.
```

```
relation recyclable_material( raleigh_material )
    =::= { cfrp, aluminium, steel }.

relation supports( embodiment, embodiment )
    =::= { (E1,E2):
           supports( { X | exists entity( X ):
                           derives_from( X, E1 ) },
                     { Y | exists entity( Y ):
                           derives_from( Y, E2 ) } ) }.

relation supports( set of entity, set of entity )
    =::= { (E1s,E2s): exists F in E1s: frame(F) and supports( F, E2s ) }.

relation supports( frame, set of entity )
    =::= { (F,Es): all E in (Es-{F}): supports( F, E ) }.

relation supports( entity, entity )
    =::= { (E1,E2): is_in_the_same_scheme_as( E1, E2 ) and
                    !exists mechanical_interface(M):
                        is_in_the_same_scheme_as( M, E1 ) and
                        M.entity_1     = E1.id and
                        M.entity_2     = E2.id and
                        M.relationship = supports }.

function known_means_of( means ) -> known_means
    =::= { M -> a_bicycle: bicycle(M),
           M -> a_skateboard: skateboard(M),
           M -> a_wheel_assembly: wheel_assembly(M),
           M -> a_saddle: saddle(M),
           M -> a_chassis: chassis(M),
           M -> a_steering_assembly: steering_assembly(M),
           M -> an_engine: engine(M),
           M -> a_harness: harness(M),
           M -> an_air_cushion: air_cushion(M),
           M -> a_frame: frame(M),
           M -> a_handlebar_assembly: handlebar_assembly(M),
           M -> a_molded_frame: molded_frame(M),
           M -> a_pedal_assembly: pedal_assembly(M),
           M -> a_chain: chain(M) }.

function mass_of( scheme ) -> real
    =::= { S -> sum( { mass_of( E ) | exists entity(E):
                    is_a_part_of( E, S ) } ) }.

function mass_of( raleigh_entity ) -> real
    =::= { E -> 2:  E.material = cfrp,
           E -> 3:  E.material = titanium,
           E -> 5:  E.material = aluminium,
           E -> 10: E.material = steel }.
```

```
function number_of_parts_in( scheme ) -> integer
    =::= { S -> cardinality( { E | exists entity(E):
                    is_a_part_of( E, S ) } ) }.

function width_of( scheme ) -> real
    =::= { S -> width_of( { E | exists entity(E):
                    is_a_part_of( E, S ) } ) }.

function width_of( set of entity ) -> real
    =::= { Es -> sum( { E.width | exists E in Es:
                    not exists E2 in Es:
                        overlaps( E2, E ) and
                        E2.width > E1.width } ) }.
```

C.2 The contents of `vehicle_spec.gal`

This module contains the full Galileo implementation of the specification for a product which provides the function `provide transport` and which exhibits a required set of physical properties. This constraint-based representation of the design specificiation has been developed from the informal set of design requirements described in Section 5.1. This module is organised as follows. A domain definition is presented first. Relation definitions are then presented, in alphabetical order.

```
module vehicle_spec.

import raleigh_knowledge.
import generic_concepts.
import comparison.

domain vehicle_scheme
    =::= { S: scheme( S ) and
              provides_the_function( S.structure.intended_function,
                                    'provide', 'transport' ) and
              recyclable(S) and
              exists( S.width : real ) and
              !S.width = width_of( S ) and
              S.width =< 2.0 and
              exists( S.mass  : preference ) and
              S.mass.intent = minimal and
              !S.mass.value  = mass_of( S ) and
              exists( S.number_of_parts : preference ) and
              S.number_of_parts.intent = minimal and
              !S.number_of_parts.value  = number_of_parts_in( S ) }.
```

```
relation has_better_mass_than( vehicle_scheme, vehicle_scheme )
    =::= { (S1,S2): better_than( S1.mass, S2.mass ) }.

relation has_better_number_of_parts_than( vehicle_scheme, vehicle_scheme )
    =::= { (S1,S2): better_than( S1.number_of_parts, S2.number_of_parts ) }.


relation improves_on( vehicle_scheme, vehicle_scheme )
    =::= { (S1,S2): has_better_mass_than( S1, S2 ) or
                    has_better_number_of_parts_than( S1, S2 ) }.
```

Appendix D

An Industrial Case-Study

This appendix presents all the Galileo code used in the industrial case-study carried out in association with Bourns Electronics Ireland. This case-study is discussed in detail in Section 5.2. Two Galileo modules are presented. The first module, called `bourns_knowledge.gal`, can be regarded as a database of design knowledge about designing the type of contact involved in the case-study. The second module, called `contact_spec.gal`, contains the full Galileo implementation of the specification for the case-study product. Using these two modules in conjunction with the `generic_concepts.gal` and `comparison.gal` modules, presented in Appendix B, designers can develop a set of alternative schemes to meet the design specification for the product.

D.1 The contents of `bourns_knowledge.gal`

The `bourns_knowledge.gal` module can be regarded as a database of design knowledge that can be used to design products at Bourns Electronics Ireland. Using this module in conjunction with the `contact_spec.gal` module presented in Section D.2 and the modules presented in Appendix B, designers can be supported in developing a set of alternative schemes to meet the design specification of the product. This module is organised as follows. Domain definitions and constraints quantified over these domains are presented first, in alphabetical order of the domain names. Relation definitions are then presented, followed by function definitions, both in alphabetical order.

```
module bourns_knowledge.

import generic_concepts.

domain bourns_entity
    =::= { R: entity(R) and
              exists( R.width      : real ) and
              exists( R.height     : real ) and
              exists( R.length     : real ) and
              exists( R.mass       : real ) and
              exists( R.material   : known_material ) and
              exists( R.resistance : real ) and
              R.mass = mass_of( R ) and
              R.resistance = resistance_of( R ) }.

domain bourns_interface
    =::= { I: interface(I) and
              exists( I.type : bourns_interface_type ) }.

domain bourns_interface_type
    =::= { spatial, physical, electrical }.

domain conducting_strip
    =::= { C: conductor_entity(C) }.

domain conducting_wire
    =::= { C: conductor_entity(C) }.

domain conductor_entity
    =::= { S: bourns_entity( S ) and
              conductor_material( R.material ) }.

domain electrical_interface
    =::= { S: bourns_interface(S) and S.type = electrical and
              exists( S.relationship : electrical_relationship ) }.

domain electrical_relationship
    =::= { insulative, conductive }.

domain female_modular_contact
    =::= { F: principle(F) and
           exists( F.e1 : embodiment ) and
           exists( F.e2 : embodiment ) and
           exists( F.e3 : embodiment ) and
           exists( F.e4 : embodiment ) and
           provides_the_function( B.e1.intended_function,
                                  'provide', 'support' ) and
           provides_the_function( B.e2.intended_function,
                                  'provide', 'contact' ) and
```

```
            provides_the_function( B.e3.intended_function,
                              'connect', 'contact' ) and
            provides_the_function( B.e4.intended_function,
                              'facilitate', 'external connection' ) and
            is_electrically_connected_to( B.e2, B.e3 ) and
            is_electrically_connected_to( B.e3, B.e4 ) and
            supports( B.e1, B.e2 ) and
            supports( B.e1, B.e3 ) and
            supports( B.e1, B.e4 ) }.

domain known_material
    =::= { gold, palladium_silver, palladium_gold, platinum_gold,
           alumina96, alumina995, beryllia995 }.

domain known_means
    =::= { a_female_modular_contact, a_precious_metal_contact,
           a_substrate, a_molded_body, a_machined_body,
           a_thickfilm_termination, a_male_contact,
           a_conducting_strip, a_conducting_wire,
           a_metal_termination, a_thickfilm_conductor }.

domain machined_body
    =::= { C: substrate_entity(C) }.

domain male_contact
    =::= { C: conductor_entity(C) }.

domain metal_termination
    =::= { C: conductor_entity(C) }.

domain molded_body
    =::= { C: substrate_entity(C) }.

domain physical_interface
    =::= { S: bourns_interface(S) and S.type = physical and
              exists( S.relationship : physical_relationship ) }.

domain physical_relationship
    =::= { solder, epoxy, print, plate }.

domain precious_metal_contact
    =::= { C: conductor_entity(C) }.

domain spatial_interface
    =::= { S: bourns_interface(S) and S.type = spatial and
              exists( S.relationship : spatial_relationship ) }.

domain spatial_relationship
    =::= { above, beside, under }.
```

```
domain substrate
    =::= { C: substrate_entity(C) }.

domain substrate_entity
    =::= { S: bourns_entity( S ) and
              substrate_material( R.material ) }.

domain thickfilm_conductor
    =::= { C: conductor_entity(C) }.

domain thickfilm_termination
    =::= { C: conductor_entity(C) }.

all embodiment(E): E.chosen_means = a_conducting_strip implies
    !exists conducting_strip( S ):
        must_be_directly_used_for( S, E.intended_function ).

all embodiment(E): E.chosen_means = a_conducting_wire implies
    !exists conducting_wire( W ):
        must_be_directly_used_for( W, E.intended_function ).

all embodiment(E): E.chosen_means = a_female_modular_contact implies
    !exists female_modular_contact( F ):
        must_be_directly_used_for( F, E.intended_function ) and
        causes( E, F.e1 ) and
        causes( E, F.e2 ) and
        causes( E, F.e3 ) and
        causes( E, F.e4 ).

all embodiment(E): E.chosen_means = a_male_contact implies
    !exists male_contact( M ):
        must_be_directly_used_for( M, E.intended_function ).

all embodiment(E): E.chosen_means = a_machined_body implies
    !exists machined_body( M ):
        must_be_directly_used_for( M, E.intended_function ).

all embodiment(E): E.chosen_means = a_metal_termination implies
    !exists metal_termination( T ):
        must_be_directly_used_for( T, E.intended_function ).

all embodiment(E): E.chosen_means = a_molded_body implies
    !exists molded_body( B ):
        must_be_directly_used_for( B, E.intended_function ).

all embodiment(E): E.chosen_means = a_precious_metal_contact implies
    !exists precious_metal_contact( P ):
        must_be_directly_used_for( P, E.intended_function ).
```

```
all embodiment(E): E.chosen_means = a_substrate implies
    !exists substrate( S ):
        must_be_directly_used_for( S, E.intended_function ).

all embodiment(E): E.chosen_means = a_thickfilm_conductor implies
    !exists thickfilm_conductor( T ):
        must_be_directly_used_for( T, E.intended_function ).

all embodiment(E): E.chosen_means = a_thickfilm_termination implies
    !exists thickfilm_termination( T ):
        must_be_directly_used_for( T, E.intended_function ).

relation can_simultaneously_provide( known_means, set of func )
    =::= { (a_female_modular_contact,{F}):
             provides_the_function(F,'provide','structured contact'),
           (a_male_contact,{F}):
             provides_the_function(F,'provide','structured contact'),
           (a_substrate,{F}):
             provides_the_function(F,'provide','support'),
           (a_molded_body,{F}):
             provides_the_function(F,'provide','support'),
           (a_machined_body,{F}):
             provides_the_function(F,'provide','support'),
           (a_thickfilm_termination,{F}):
             provides_the_function(F,'facilitate','external connection'),
           (a_conducting_strip,{F1,F2,F3}):
             provides_the_function(F1,'provide','contact') and
             provides_the_function(F2,'connect','contact') and
             provides_the_function(F3,'facilitate','external connection'),
           (a_metal_termination,{F}):
             provides_the_function(F,'facilitate','external connection'),
           (a_precious_metal_contact,{F}):
             provides_the_function(F,'provide','contact'),
           (a_thickfilm_conductor,{F1,F2}):
             provides_the_function(F1,'provide','contact') and
             provides_the_function(F2,'connect','contact'),
           (a_conducting_wire,{F}):
             provides_the_function(F,'connect','contact') }.

relation conductor_material( known_material )
    =::= { gold, palladium_silver, palladium_gold, platinum_gold }.

relation end_to_end_stackable( scheme )
    =::= { S: exists substrate_entity(E):
                  is_a_part_of( E, S ) and
                  E.length = 2 * S.contact_pitch }.
```

```
relation is_electrically_connected_to( embodiment, embodiment )
    =::= { (E1,E2):
           is_electrically_connected_to( { X | exists entity( X ):
                                              derives_from( X, E1 ) },
                                          { Y | exists entity( Y ):
                                              derives_from( Y, E2 ) } ) }.

relation is_electrically_connected_to( set of entity, set of entity )
    =::= { (E1s,E2s): exists E1 in E1s, E2 in E2s:
                        is_electrically_connected_to( E1, E2 ) }.

relation is_electrically_connected_to( entity, entity )
    =::= { (E1,E2): E1 = E2 and conductor_entity(E1),
           (E1,E2): is_in_the_same_scheme_as( E1, E2 ) and
                    !exists electrical_interface(E):
                        E.entity1 = E1.id and
                        E.entity2 = E2.id and
                        E.relationship = conductive }.

relation substrate_material( known_material )
    =::= { alumina96, alumina995, beryllia995 }.

relation supports( embodiment, embodiment )
    =::= { (E1,E2):
           supports( { X | exists entity( X ):
                             derives_from( X, E1 ) },
                     { Y | exists entity( Y ):
                             derives_from( Y, E2 ) } ) and

relation supports( set of entity, set of entity )
    =::= { (E1s,E2s): exists E in E1s:
                        substrate_entity(E) and
                        supports( E, E2s ) }.

relation supports( substrate_entity, set of entity )
    =::= { (S,Es): all E in (Es-{S}): supports( S, E ) }.

relation supports( entity, entity )
    =::= { (E1,E2): E1 = E2,
           (E1,E2): is_in_the_same_scheme_as( E1, E2 ) and
                    !exists physical_interface(P):
                      is_in_the_same_scheme_as( P, E1 ) and
                      M.entity_1     = E1.id and
                      M.entity_2     = E2.id }.

function contact_to_solder_resistance_of( scheme ) -> real
    =::= { R -> sum( { E.resistance | exists conducting_entity(E):
                                        is_a_part_of( E, S ) }.
```

```
function density_of( known_material ) -> real
    =::= { M -> 2: substrate_material(M),
           M -> 4: conductor_material(M) }.

function known_means_of( means ) -> known_means
    =::= { M -> a_female_modular_contact: female_modular_contact(M),
           M -> a_male_contact: male_contact(M),
           M -> a_precious_metal_contact: precious_metal_contact(M),
           M -> a_substrate: substrate(M),
           M -> a_molded_body: molded_body(M),
           M -> a_machined_body: machined_body(M),
           M -> a_thickfilm_termination: thickfilm_termination(M),
           M -> a_conducting_strip: conducting_strip(M),
           M -> a_metal_termination: metal_termination(M),
           M -> a_thickfilm_conductor: thickfilm_conductor(M),
           M -> a_conducting_wire: conducting_wire(M) }.

function mass_of( scheme ) -> real
    =::= { S -> sum( { mass_of( E ) | exists entity(E):
                    is_a_part_of( E, S ) } ) }.

function mass_of( bourns_entity ) -> real
    =::= { E -> density_of( E.material )
                * E.length * E.width * E.height }.

function power_consumption_of( scheme ) -> real
    =::= { S -> S.contact_to_solder_resistance * S.current * S.current }.

function resistance_of( bourns_entity ) -> real
    =::= { E -> resistivity_of( E.material ) * E.length
                / ( E.width * E.height ) }.

function resistivity_of( known_material ) -> real
    =::= { gold             -> 0.005,
           palladium_silver -> 0.03,
           palladium_gold   -> 0.05,
           platinum_gold    -> 0.05,
           alumina96         -> 1.0E14,
           alumina995        -> 1.0E14,
           beryllia995       -> 1.0E14 }.
```

D.2 The contents of contact_spec.gal

This module contains the full Galileo implementation of the specification for the industrial case-study product discussed in Chapter 5. This constraint-based representation of the design specificiation has been developed from the informal set of design requirements described in Section 5.2. This module is organised as follows. A domain definition is presented first. Relation definitions are then presented, in alphabetical order.

```
module contact_spec.

import bourns_knowledge.
import generic_concepts.
import comparison.

domain contact_scheme
    =::= { S: scheme( S ) and
              provides_the_function( S.structure.intended_function,
              'provide', 'structured contact' ) and
              exists( S.contact_pitch : real ) and
              S.contact_pitch = 2.54 and
              exists( S.current : real ) and
              S.current = 3 and
              exists( S.power_consumption : preference ) and
              S.power_consumption.intent = minimal and
              !S.power_consumption.value  = power_consumption_of( S ) and

              exists( S.contact_to_solder_resistance : preference ) and

              S.contact_to_solder_resistance.intent = minimal and
              !S.contact_to_solder_resistance.value
                  = contact_to_solder_resistance_of( S ) and
              exists( S.mass : preference ) and
              S.mass.intent = minimal and
              !S.mass.value  = mass_of( S ) and
              end_to_end_stackable( S ) }.

relation has_better_contact_solder_resistance_than( scheme, scheme )
    =::= { (S1,S2): better_than( S1.contact_solder_resistance,
                    S2.contact_solder_resistance ) }.

relation has_better_mass_than( scheme, scheme )
    =::= { (S1,S2): better_than( S1.mass, S2.mass ) }.

relation has_better_power_consumption_than( scheme, scheme )
    =::= { (S1,S2): better_than( S1.power_consumption,
                    S2.power_consumption ) }.

relation improves_on( scheme, scheme )
    =::= { (S1,S2): has_better_power_consumption_than( S1, S2 ) or
                    has_better_contact_solder_resistance_than( S1, S2 ) or
                    has_better_mass_than( S1, S2 ) }.
```

Bibliography

[1] A. Agogino, J. E. Barreto, B. Chidambaram, A. Dong, A. Varma, and W. H. Wood. The Concept Database: A design information system for concurrent engineering with application to mechatronics design. In *Proceedings of 6th Conference on Design Engineering and System Division*, number 96-45, pages 89–92, 1997.

[2] L. K. Alberts. *YMIR: A domain-independant onthology for the formal representation of engineering design knowledge.* PhD thesis, University of Twente, Enchede, The Netherlands, 1993.

[3] K. Andersson. A vocabulary for conceptual design – part of a design grammar. In John Gero and Fay Sudweeks, editors, *Proceedings of the IFIP WG-5.3 Workshop on Formal Design Methods for Computer-Aided-Design*, pages 139–152, Tallinn, Estonia, June 1993. Key Centre of Design Computing, University of Sidney.

[4] Mogens Myrup Andreasen. The Theory of Domains. In *Proceedings of Workshop on Understanding Function and Function-to-Form Evolution*, Cambridge University, 1992.

[5] Mogens Myrup Andreasen and Lars Hein. *Integrated Product Development.* IFS Publications Ltd./Springer-Verlag, Bedford, UK, 1987.

[6] M. Asimow. *Introduction to Design.* Prentice-Hall Inc., Englewood Cliffs, NJ, USA, 1962.

[7] D. Bahler, C. Dupont, and J. Bowen. An axiomatic approach that supports negotiated resolution of design conflicts in Concurrent Engineering. In *Artificial Intelligence in Design*, pages 363–379, 1994. Kluwer Academic Press, Netherlands.

[8] Dennis Bahler and James Bowen. Constraint logic and its applications in production: an implementation using the galileo4 language and system. In A. Artiba and S. E. Elmaghraby, editors, *The Planning and Scheduling of Production Systems: Methodologies and applications*, chapter 8, pages 227–270. Chapman & Hall, 1997.

[9] Nigel Ball, Pater Mattews, and Ken Wallace. Managing conceptual design objects: An alternative to geometry. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '98*, pages 67–86, The Netherlands, July 1998. Kluwer Academic Publishers.

[10] Medjdouband Benachir and Bernard Yannou. Topological enumeration heuristics in constraint-based space layout planning. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '98*, pages 271–290, The Netherlands, July 1998. Kluwer Academic Publishers.

[11] P. J. Bentley and J. P. Wakefield. Conceptual evolutionary design by genetic algorithms. *Engineering Design and Automation*, 3(2):119–131, 1997.

[12] Peter Bentley. *Generic Evolutionary Design of Solid Objects using a Genetic Algorithm*. PhD thesis, University of Huddersfield, 1996.

[13] Sanjay Bhansali, Glenn A. Kramer, and Tim J. Hoar. A principled approach towards symbolic geometric constraint satisfaction. *Journal of Artificial Intelligence Research*, 4:419–443, June 1996.

[14] S. Bhatta, A. Goel, and S. Prabhakar. Innovation in analogical design: A model-based approach. In J. S. Gero and F. Sudweeks, editors, *Artificial Intelligence in Design*, pages 57–74, 1994. Kluwer Academic Press, Netherlands.

[15] Anton Biasizzo and Franc Novak. A methodology for model-based diagnosis of analog circuits. Technical Report CSD-TR-95-11, Jožef Stephan Institute, Slovenia, 1995.

[16] William P. Birmingham and Allen Ward. What is Concurrent Engineering? *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 9:67–68, 1995. Guest Editorial in a Special Issue on Concurrent Engineering.

[17] Geoffrey Boothroyd. Design for Manufacture and Assembly: The Boothroyd-Dewhurst Experience. In George Q. Huang, editor, *Design for X: Concurrent Engineering Imperatives*, chapter 1, pages 19–40. Chapman & Hall, London, 1996.

[18] A. H. W. Bos. Aircraft conceptual design by genetic/gradient-guided optimisation. *Engineering Applications of Artificial Intelligence*, 11:377–382, 1998.

[19] James Bowen. Using dependency records to generate design coordination advice in a constraint-based approach to Concurrent Engineering. *Computers in Industry*, 33:191–199, 1997.

[20] James Bowen. Personal communication with Prof. James Bowen, 1999.

[21] James Bowen and Dennis Bahler. Full first-order logic in knowledge representation – a constraint-based approach. Technical Report TR-91-29, North Carolina State University, 1991. Computer Science Technical Report.

[22] James Bowen and Dennis Bahler. Frames, quantification, perspectives and negotiation in constraint networks in life-cycle engineering. *International Journal for Artificial Intelligence in Engineering*, 7:199–226, 1992.

[23] James Bowen, Peter O'Grady, and Larry Smith. A constraint programming language for life-cycle engineering. *International Journal for Artificial Intelligence in Engineering*, 5:206–220, 1990.

[24] James A. Bowen and Dennis Bahler. Conditional existance of variables in generalised constraint networks. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 215–220, 1991.

[25] R. H. Bracewell and J. E. E. Sharpe. Functional descriptions used in computer support for qualitative scheme generation = "Schemebuilder". *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):333–345, 1996.

[26] K. N. Brown, C. A. McMahon, and J. H. Sims Williams. Constraint unification grammars: Specifying languages of parametric designs. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '94*, pages 239–256, The Netherlands, August 1994. Kluwer Academic Publishers.

[27] M. J. Buckley, K. W. Fertig, and D. E. Smith. Design sheet: An environment for facilitating flexible trade studies during conceptual design. In *AIAA 92-1191 Aerospace Design Conference*, February 1992. Irvine, California.

[28] J. Buur. *A Theoretical Approach to Mechatronics Design*. PhD thesis, Technical University of Denmark, Lyngby, 1990.

[29] Jacob Buur and Mogens Myrup Andreasen. Design models in mechatronic product development. *Design Studies*, 10(3):155–162, July 1989.

[30] J. Cagan and W. J. Mitchell. A grammatical approach to network flow synthesis. In John Gero and Fay Sudweeks, editors, *Proceedings of the IFIP WG-5.3 Workshop on Formal Design Methods for Computer-Aided-Design*, pages 153–166, Tallinn, Estonia, June 1993. Key Centre of Design Computing, University of Sidney.

[31] Matthew I. Campbell, Jonathan Cagan, and Kenneth Kotovsky. A-design: Theory and implementation of an adaptive agent-based method of conceptual design. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '98*, pages 579–598, The Netherlands, July 1998. Kluwer Academic Publishers.

[32] L. Candy, E. A. Edmonds, and D. J. Patrick. Interactive knowledge support to conceptual design. In *AI System Support for Conceptual Design – Proceedings of the 1995 Lancaster International Workshop in Engineering Design*, Lancaster International Workshop in Engineering Design, pages 260–278, March 1995.

[33] Sophie Canet-Magnet and Bernard Yannou. Ifnot: A function-based approach for both proactive and reactive integration of new productive technologies. In *Proceedings of the 11th International FLAIRS Conference*, pages xx–xx, Sundial Beach Resort, Sanibel Island, Florida, USA, May 1998.

[34] A. Chakrabarti and T. P. Bligh. A two-step approach to conceptual design of mechanical parts. In J. S. Gero and F. Sudweeks, editors, *Artificial Intelligence in Design*, pages 21–38, 1994. Kluwer Academic Press, Netherlands.

[35] Amaresh Chakrabarti and Lucienne Blessing. Guest editorial: Representing functionality in design. *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):251–253, 1996.

[36] Amaresh Chakrabarti and Thomas P. Bligh. An approach to functional synthesis of mechanical design concepts: Theory, applications and emerging research issues. *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):313–331, 1996.

[37] Amaresh Chakrabarti and Ming Xi Tang. Generating conceptual solutions on FuncSION: Evolution of a functional synthesiser. In J. S. Gero and F. Sudweeks, editors, *Artificial Intelligence in Design '96*, pages 603–622, 1996.

[38] V. Chankong and Y. Haimes. *Multiobjective Decision Making*. North-Holland, New York, 1983.

[39] Don Clausing. *Total Quality Development : A Step-By-Step Guide to World-Class Concurrent-Engineering*. ASME Press, 345, East 47th Stret, New York, NY 10017, USA, 1993. American Society of Mechanical Engineers.

[40] K. Clibborn, E. Edmonds, and L. Candy. Representing conceptual design knowledge with multi-layered logic. In *AI System Support for Conceptual Design – Proceedings of the 1995 Lancaster International Workshop in Engineering Design*, Lancaster International Workshop in Engineering Design, pages 93–108, March 1995.

[41] Mentor Graphics Corporation. http://www.mentor.com. Web-site.

[42] Andrew W. Court. A review of automatic configuration design. Technical Report 047/1995, Engineering Design Centre in Fluid Mechanics, University of Bath, July 1995.

[43] Indraneel Das. *Nonlinear Multicriteria Optimization and Robust Optimality*. PhD thesis, Rice University, Houston, Texas, April 1997.

[44] Frank Dikker. *A Knowledge-Based Approach to Evaluation of Norms in Engineering Design*. PhD thesis, Univerity of Twente, June 1995.

[45] J. R. Dixon, M. F. Orelup, and R. V. Welch. A research progress report: robust parametric designs and conceptual models. In *Proceedings of the 1993 NSF*

Design and Manufacturing Systems Conference, Charlotte, North Carolina, January 1993. Society of Manufacturing Engineers.

[46] Maurice Dohmen. *Constraint-Based Feature Validation.* PhD thesis, Technische Universiteit Delft, December 1997.

[47] I. Donaldson and K. MacCallum. The role of computational prototypes in conceptual models for engineering design. In J. S. Gero and F. Sudweeks, editors, *Artificial Intelligence in Design*, pages 3–20, 1994. Kluwer Academic Press, Netherlands.

[48] Ian Donaldson. Semantic consensus in design concept libraries. In *Workshop: Semantic Basis for Sharing of Knowledge and Data in Design*, AID-94, Lausanne, Switzerland, August 1994.

[49] Marc van Dongen, Barry O'Sullivan, James Bowen, Alex Ferguson, and Mike Baggaley. Using constraint programming to simplify the task of specifying DFX guidelines. In Kulwant S. Pawar, editor, *Proceedings of the 4th International Conference on Concurrent Enterprising*, pages 129–138, University of Nottingham, October 1997.

[50] A. H. B. Duffy. Learning in design. *IEEE Expert – Intelligent Systems and their Applications*, pages 71–76, June 1997. Special Issue on Artificial Intelligence in Design.

[51] A. H. B. Duffy, M. M. Andreasen, K. J. MacCallum, and L. N. Reijers. Design Co-ordination for Concurrent Engineering. *Journal of Engineering Design*, 4(4):251–265, December 1993.

[52] A. H. B. Duffy and S. Legler. A methodology for rationalising past designs for reuse. In M. Tichem, T. Storm, M. M. Andreasen, and A. H. B. Duffy, editors, *Proceedings of the 4th WDK Workshop on Product Structuring*, WDK Product Structuring Workshop Series, Technical University of Delft, October 1998.

[53] Clive. Dym. *Engineering Design: A Synthesis of Views.* Cambridge University Press, 1994.

[54] Per Elgård. Industrial practices with product platforms in the USA. In M. Tichem, T. Storm, M. M. Andreasen, and A. H. B. Duffy, editors, *Proceedings of the 4th WDK Workshop on Product Structuring*, WDK Product Structuring Workshop Series, Technical University of Delft, October 1998.

[55] P. A. van Elsas and J. S. M. Vergeest. Displacement feature modelling for conceptual design. *Computer-Aided Design*, 30(1):19–27, January 1998.

[56] Freek Erens and Karel Verhulst. Architectures for product families. In *Proceedings of the 2nd WDK Workshop on Product Structuring*, pages 45–60, Delft University of Technology, June 1996.

[57] Gunnar Erixon. Modular Function Deployment (MFD), support for good product structure creation. In M. Tichem, T. Storm, M. M. Andreasen, and K. J. MacCallum, editors, *Proceedings of the 2nd WDK Workshop on Product Structuring*, pages 181–196, Delft University of Technology, June 1996.

[58] Gunnar Erixon. Modular Function Deployment (MFD) – industrial experiences. In M. Tichem, T. Storm, M. M. Andreasen, and A. H. B. Duffy, editors, *Proceedings of the 3rd WDK Workshop on Product Structuring*, WDK Product Structuring Workshop Series, pages 149–158, Technical University of Delft, June 1997.

[59] Boi Faltings and Eugene C. Freuder, editors. *IEEE Intelligent Systems and their applications*, volume 13. IEEE Computer Society, July–August 1998. Special Issue on Configuration – Getting it Right.

[60] Yousri El Fattah. Constraint logic programming for structure-based reasoning about dynamic physical systems. *Artificial Intelligence in Engineering*, 1:253–264, 1996.

[61] P. A. Fitzhorn. Formal graph languages of shape. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing*, 4(3), 1990.

[62] Gerhard Fleischanderl, Gerhard Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems and their applications*, 13(4):59–68, July–August 1998. Special Issue on Configuration – Getting it Right.

[63] Pat Fothergill, Jorg Forster, Jose Angel Lakunza, Fco Plaza, and Ines Arana. DEKLARE: A methodological approach to re-design. In *Proceedings of Conference in Integration in Manufacturing*, Vienna, Austria, September 1995.

[64] J. K. Fowlkes, Wayne F. Ruggles, and Groothuis John D. Advanced FAST diagramming. In *Proceedings of the SAVE Conference*, pages 45–52, Newport Beach, California, 1972.

[65] M. J. French. *Engineering Design: The Conceptual Stage*. Heinemann Educational Books, London, 1971.

[66] Eugene C. Freuder. The role of configuration knowledge in the business process. *IEEE Intelligent Systems and their applications*, 13(4):29–31, July–August 1998. Special Issue on Configuration – Getting it Right.

[67] Xiao-Shan Gao and Shang-Chou Chou. Solving geometric constraint systems. I. a global propagation approach. *Computer-Aided Design*, 30(1):47–54, January 1998.

[68] Xiao-Shan Gao and Shang-Chou Chou. Solving geometric constraint systems. II. a symbolic approach and decision of Rc-constructibility. *Computer-Aided Design*, 30(2):115–122, February 1998.

[69] Esther Gelle and Ian Smith. Dynamic constraint satisfaction with conflict management in design. In Michael Jampel, Eugene Freduer, and Michael Maher, editors, *OCS'95: Workshop on Over-Constrained Systems at CP'95*, pages 33–40, Cassis, Marseilles, Sep 1995.

[70] J. S. Gero and S. Louis. Improving pareto optimal designs using genetic algorithms. *Microcomputers in Civil Engineering*, 10(4):241–249, 1995.

[71] Ashok K. Goel and Eleni Stroulia. Functional device models and model-based diagnosis in adaptive systems. *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):355–370, 1996.

[72] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[73] S. R. Gorti, S. Humair, R. D. Sriram, S. Talukdar, and S. Murthy. Solving constraint satisfaction problems using ATeams. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 10:1–19, 1995.

[74] Xiaohong Guan and Ken J. MacCallum. Adopting a minimum commitment principle for computer aided geometric design systems. In J. S. Gero and F. Sudweeks, editors, *Artificial Intelligence in Design '96*, pages 623–639, 1996.

[75] M. J. Hague, A. Taleb-Bendiab, and M. J. Brandish. An adaptive machine learning system for computer supported conceptual design. In John Sharpe, editor, *AI System Support for Conceptual Design – Proceedings of the Lancaster Internation Workshop on Engineering Design 1995*, pages 1–16, Berlin, March 1995. Springer-Verlag. 3-540-76000-8.

[76] Djamila Haroud, Sylvie Boulanger, Esther Gelle, and Ian Smith. Management of conflict for preliminary engineering design tasks. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 9:313–323, 1995.

[77] Alois Haselböck and Markus Stumptner. An integrated approach for modelling complex configuration domains. In *Proceedings of the 13th International Conference on Artificial Intelligence, Expert Systems and Natural Language*, volume 1, pages 625–634, Avignon, France, May 1993.

[78] Carolyn C. Hayes and Harold C. Sun. Using a manufacturing constraint network to identify cost-critical areas of design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 9:73–87, 1995.

[79] J. Heisserman and R. Woodbury. Geometric design with boundary solid grammars. In John Gero and Fay Sudweeks, editors, *Proceedings of the IFIP WG-5.3 Workshop on Formal Design Methods for Computer-Aided-Design*, pages 79–100, Tallinn, Estonia, June 1993. Key Centre of Design Computing, University of Sidney.

[80] A. E. Howe, P. R. Cohen, J. R. Dixon, and M. K. Simmons. Dominic: a domain-independant program for mechanical engineering design. *Artificial Intelligence in Engineering*, 1(1):23–28, 1986.

[81] Walter Hower. *On Constraint Satisfaction and Computer-Aided Layout Design.* PhD thesis, Universität Koblenz-Landau, December 1995.

[82] Wynne Hsu and Irene M. Y. Woon. Current research in the conceptual design of mechanical products. *Computer-Aided Design*, 30(5):377–389, 1998.

[83] K. Hua and B. Faltings. Exploring case-based building design - CADRE. *Artificial Intelligence for Engineering Design, Analysis and MAnufacture*, 7(2):135–143, 1993.

[84] Kefeng Hua, Boi Faltings, Djamila Haroud, Graham Kimberley, and Ian Smith. Dynamic constraint satisfaction in a bridge design system. In Georg Gottlob and Wolfgang Nejdl, editors, *Expert Systems in Engineering - Principles and Applications*, volume 462 of *Lecture Notes in Artificial Intelligence, Subseries of Lecture Notes in Computer Science*, pages 217–232, Vienna, Austria, September 24–26, 1990. Springer-Verlag, Berlin/Heidelberg. International Workshop.

[85] George Q. Huang, editor. *Design for X: Concurrent Engineering Imperatives.* Chapman & Hall, London, 1996.

[86] Vladimir Hubka and W. Ernst Eder. *Engineering Design: General Procedural Model of Engineering Design.* Serie WDK - Workshop Design-Konstruktion. Heurista, Zurich, 1992.

[87] Paul Hudak, John Peterson, and Joseph H. Fasel. A Gentle Introduction to Haskell. Available from http://www.haskell.org, March 1997.

[88] Michael G. Hudson and Ian C. Parmee. The application of genetic algorithms to conceptual design. In John Sharpe, editor, *AI System Support for Conceptual Design – Proceedings of the Lancaster Internation Workshop on Engineering Design 1995*, pages 17–36, Berlin, March 1995. Springer-Verlag. 3-540-76000-8.

[89] H. Iivonen and A. Riitahuhta. Case-based reasoning in conceptual design. In *Proceedings of the Tenth CIM-Europe Annual Conference*, volume 5 of *Sharing CIM Solutions*, pages 307–314, Copenhagen, Denmark, October 1994.

[90] H. Iivonen, S. Silakoski, and A. Riitahuhta. Case-based reasoning and hypermedia in conceptual design. In *Proceedings of the 10th International Conference on Engineering Design, ICED-95*, International Conference Engineering Design Series, pages 1483–1488, August 1995.

[91] M. Ishii and T. Tomiyama. A synthetic reasoning method based on a physical phenomenon knowledge base. In *AI System Support for Conceptual Design – Proceedings of the 1995 Lancaster International Workshop in Engineering Design*, Lancaster International Workshop in Engineering Design, pages 109–123, March 1995.

[92] J. M. Juran and F. M. Gryna. Designing for Quality. In *Quality Planning and Analysis*, Industrial Engineering Series, chapter 12, pages 253–286. McGraw-Hill, New York, 1993.

[93] H. Kawakami, O. Katai, T. Sawaragi, and S. Iwai. Knowledge acquisition method for conceptual design based on value engineering and axiomatic design theory. *Artificial Intelligence in Engineering*, 1:187–202, 1996.

[94] L. B. Keat, C. L. Tan, and K. Mathur. Design model: Towards an integrated representation for design semantics and syntax. In *AI System Support for Conceptual Design – Proceedings of the 1995 Lancaster International Workshop in Engineering Design*, Lancaster International Workshop in Engineering Design, pages 124–137, March 1995.

[95] Terry W. Knight. Designing a shape grammar: Problems of predicability. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '98*, pages 499–516, The Netherlands, July 1998. Kluwer Academic Publishers.

[96] Klaas Jan de Kraker. *Feature Conversion for Concurrent Engineering.* PhD thesis, Technische Universiteit Delft, November 1997.

[97] S. R. T. Kumara and S. V. Kamarthi. Application of adaptive resonance networks for conceptual design. *Annals of the CIRP*, 42:213–216, 1992.

[98] K. Lambert and B. van Fraassen. *Derivation and Counterexample: An Introduction to Philosophical Logic.* Dickenson Publishing Company, Enrico, CA, 1972.

[99] Ulf Liedholm. Conceptual design of product and product families. In M. Tichem, T. Storm, M. M. Andreasen, and A. H. B. Duffy, editors, *Proceedings of the 4th WDK Workshop on Product Structuring*, October 1998.

[100] R-S. Lossack, Y. Umeda, and T. Tomiyama. Requirement, function and physical principle modelling as the basis for a model of synthesis. In *Computer-Aided Conceptual Design '98*, pages 165–179. Lancaster University, May 1998. Proceedings of the 1998 Lancaster International Workshop on Engineering Design.

[101] B. Lotter. *Manufacturing Assembly Handbook*. Butterworths, Boston, 1986.

[102] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[103] M. L. Maher and D. M. Zhang. CADSYN: a case-based design process model. *Artificial Intelligence for Engineering Design, Analysis and MAnufacture*, 7(2):97–110, 1993.

[104] Michael E. McGrath, Michael T. Anthony, and Amram R. Shapiro. *Product Development: Success through Product and Cycle-time Excellence*. The Electronic Business Series. Butterworth Heinemann, Stoneham, MA, 1992.

[105] Harald Meyer auf'm Hofe. Partial satisfaction of constraint hierarchies in reactive and interactive configuration. In Walter Hower and Ruttkay. Zsòfia, editors, *Workshop on Non-Standard Constraint Processing, ECAI 96*, August 1996. Budapest, Hungary.

[106] L. D. Miles. *Techniques of Value Analysis and Engineering*. McGraw-Hill, New York, 1972.

[107] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI 90, Eighth National Conference on Artificial Intelligence*, volume 1, pages 25–32, Boston, MA, July–August 1990. AAAI Press, Menlo Park, CA, U.S.A.

[108] C. J. Moore, J. C. Miles, and D. W. G. Rees. Decision support for conceptual bridge design. *Artificial Intelligence in Engineering*, 11:259–272, 1997.

[109] Y. Nagai and S. Terasaki. A constraint-based knowledge compiler for parametric design problem in mechanical engineering. Technical Report TM-1270, ICOT, Japan, June 1993.

[110] Igor Novak, Franc Mozetič, Marina Santo-Zarnik, and Anton Biasizzo. Enhancing Design For Test for active analog filters by using CLP($\Re$). *Journal of Electronic Testing: Theory and Applications*, 4(4):315–329, 1993.

[111] Barry O'Sullivan. A constraint-based support tool for early-stage design within a Concurrent Engineering environment. In Kulwant S. Pawar, editor, *Proceedings of the 4th International Conference on Concurrent Enterprising*, pages 119–28, University of Nottingham, October 1997.

[112] Barry O'Sullivan. Conflict management and negotiation for Concurrent Engineering using Pareto optimality. In R. Mackay N. Mårtensson and S. Björgvinsson, editors, *Changing the ways we work – Shaping the ICT-solutions for the next century*, Advances in Design and Manufacturing, pages 359–368, Amsterdam, October 1998. IOS Press. Proceedings of the Conference on Integration in Manufacturing, Göteborg, Sweden.

[113] Barry O'Sullivan. The paradox of using constraints to support creativity in conceptual design. In Alan Bradshaw and John Counsell, editors, *Computer-Aided Conceptual Design '98*, pages 99–122. Lancaster University, May 1998. Proceedings of the 1998 Lancaster International Workshop on Engineering Design.

[114] Barry O'Sullivan. Supporting the design of product families through constraint-based reasoning. In M. Tichem, T. Storm, M. M. Andreasen, and A. H. B. Duffy, editors, *Proceedings of the 4th WDK Workshop on Product Structuring*, WDK Product Structuring Workshop Series, Technical University of Delft, October 1998.

[115] Barry O'Sullivan and James Bowen. A constraint-based approach to supporting conceptual design. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '98*, pages 291–308, The Netherlands, July 1998. Kluwer Academic Publishers.

[116] Barry O'Sullivan, James Bowen, and Alexander B. Ferguson. A new technology for enabling computer-aided EMC analysis. In *Workshop on CAD Tools for EMC (EMC-York 99)*, July 1999.

[117] Jahir Pabon, Robert Young, and Walid Keirouz. Integrating parametric geometry, features and variational modeling for conceptual design. *International Journal of Systems Automation: Research and Applications (SARA)*, 2:17–36, 1992.

[118] G. Pahl and W. Beitz. *Engineering Design: A systematic approach*. Springer, London, 2nd edition, 1995.

[119] Ian C. Parmee. Adaptive search techniques for decision support during preliminary engineering design. In *Proceedings of Informing Technologies to Support Engineering Decision Making*, EPSRC/DRAL Seminar, Institution of Civil Engineers, London, November 1994.

[120] M. Pearce, A. K. Goel, J. L. Kolodner, C. Zimring, L. Sentosa, and R. Billington. Case-based design support: a case-study in architectural design. *IEEE Expert*, 7(5):14–20, October 1992.

[121] Feng Peien, Xu Guorong, and Zhang Mingjun. Feature modelling based on design catalogues for principle conceptual design. *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):347–354, 1996.

[122] Henk Jan Pels. Modularity in product design. In M. Tichem, T. Storm, M. M. Andreasen, and A. H. B. Duffy, editors, *Proceedings of the 3rd WDK Workshop on Product Structuring*, WDK Product Structuring Workshop Series, pages 137–148, Technical University of Delft, June 1997.

[123] Charles J. Petrie, Teresa A. Webster, and Mark R. Cutkosky. Using Pareto optimality to coordinate distributed agents. *AIEDAM*, 9:269–281, 1995.

[124] Henry Petroski. *Paconius and the Pedestal for Apollo: A Paradigm of Error in Conceptual Design*, chapter 2, pages 15–28. Cambridge University Press, 1994.

[125] I. Porter, J. M. Counsell, and J. Shao. Knowledge representation for mechatronic systems. In Alan Bradshaw and John Counsell, editors, *Computer-Aided*

Conceptual Design '98, pages 181–195. Lancaster University, May 1998. Proceedings of the 1998 Lancaster International Workshop on Engineering Design.

[126] M. J. Pratt. Solid modelling and the interface between design and manufacturing. *IEEE Computer Graphics and Applications*, pages 52–59, 1984.

[127] Lena Qian and John S. Gero. Function-behaviour-structure paths and their role in analogy-based design. *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):289–312, 1996.

[128] Joerg Rahmer and Anji Voss. Supporting explorative configuration. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '98*, pages 483–498, The Netherlands, July 1998. Kluwer Academic Publishers.

[129] Zuken Redac. http://www.redac.co.uk. Web-site.

[130] S. Y. Reddy, K. W. Fertig, and D. E. Smith. Constraint management methodology for conceptual design tradeoff studies. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences and Computers in Engineering Conference*, August 1996. Irvine, California.

[131] Sudhakar Y. Reddy. Hider: A methodology for early-stage exploration of the design space. In *Proceedings of the 1996 ASME Design Engineering Technical Conferences and Computers in Engineering Conference*, August 1996. Irvine, California.

[132] Barry de Roode. Mapping between product structures. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '98*, pages 445–459, The Netherlands, July 1998. Kluwer Academic Publishers.

[133] Daniel Sabin and Eugene C. Freuder. Configuration as composite constraint satisfaction. In *AAAI-96 Fall Symposium on Configuration*, pages 28–36, 1996. also in: Proceedings, Artificial Intelligence and Manufacturing Research Planning Workshop 1996.

[134] Daniel Sabin and Rainer Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems and their applications*, 13(4):42–49, July–August 1998. Special Issue on Configuration – Getting it Right.

[135] A. R. Sabouni and O. M. Al-Mourad. Quantitative knowledge based approach for preliminary design of tall buildings. *Artificial Intelligence in Engineering*, 11:143–154, 1997.

[136] Marc J. Schniederjans. *Goal Programming: methodology and applications*. Kluwer Academic Publishers, Boston, 1995.

[137] David Serrano. *Constraint Management in Conceptual Design*. PhD thesis, Massachusetts Institute of Technology, October 1987.

[138] Kristina Shea and Jonathan Cagan. Generating structural essays from languages of discrete structures. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '98*, pages 365–384, The Netherlands, July 1998. Kluwer Academic Publishers.

[139] Shuichi Shimizu and Masayuki Numao. Constraint-based design for 3D shapes. *Artificial Intelligence*, 91:51–69, 1997.

[140] Martin Stacey, Helen Sharp, Marian Petre, George Rzevski, and Rodney Buckland. A representation scheme to support conceptual design of mechatronic systems. In J. S. Gero and F. Sudweeks, editors, *Artificial Intelligence in Design '96*, pages 583–602, 1996.

[141] Roger B. Stake and Michael Blackenfelt. Modularity in use - experiences from the Swedish industry. In M. Tichem, T. Storm, M. M. Andreasen, and A. H. B. Duffy, editors, *Proceedings of the 4th WDK Workshop on Product Structuring*, WDK Product Structuring Workshop Series, Technical University of Delft, October 1998.

[142] Leon Sterling. Of using constraint logic programming for design of mechanical parts. In Leon Sterling, editor, *Intelligent Systems*, chapter 6, pages 101–109. Plenum Press, New York, 1993.

[143] G. Stiny and J. Gips. *Algorithmic Aesthetics*. University of California Press, Berkeley, 1978.

[144] Markus Stumptner and Franz Wotawa. Model-based reconfiguration. In John Gero and Fay Sudweeks, editors, *Artificial Intelligence in Design '98*, pages 45–64, The Netherlands, July 1998. Kluwer Academic Publishers.

[145] Robert H. Sturges, Kathleen O'Shaughnessy, and Mohammed I. Kilani. Computational model for conceptual design based on extended function logic. *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):255–274, 1996.

[146] K. Sun and B. Faltings. Supporting creative mechanical design. In *Artificial Intelligence in Design*, pages 39–56, 1994. Kluwer Academic Press, Netherlands.

[147] Cadence Systems. http://www.cadence.com. Web-site.

[148] Anna C. Thorton. *Constraint Specification and Satisfaction in Embodiment Design*. PhD thesis, Cambridge University, UK, July 1993.

[149] Anna C. Thorton. A support tool for constraint processes in embodiment design. In T. K. Hight and F. Mistree, editors, *ASME Design Theory and Methodology Conference*, pages 231–239, Minneapolis, USA, September 1994.

[150] Marcel Tichem. *A Design Coordination Approach to Design for X*. PhD thesis, Technische Universiteit Delft, September 1997.

[151] Marcel Tichem and Barry O'Sullivan. Knowledge processing for timely decision making in DFX. In Hubert Kals and Fred van Houten, editors, *Integration of Process Knowledge into Design Support Systems*, pages 219–228, Dordrecht, March 1999. Kluwer Academic Publishers. Proceedings of the 1999 CIRP International Design Seminar, University of Twente, Enschede, The Netherlands.

[152] Juha Tiihonen, Timo Lehtonen, Timo Soininen, Antti Pulkkinen, Reijo Sulonen, and Asko Riitahuhta. Modelling configurable product families. In M. Tichem, T. Storm, M. M. Andreasen, and A. H. B. Duffy, editors, *Proceedings of the 4th WDK Workshop on Product Structuring*, WDK Product Structuring Workshop Series, Technical University of Delft, October 1998. Proceedings to appear.

[153] E. Tjalve. *A Short Course in Industrial Design*. Newnes-Butterworths, London, 1979.

[154] Tetsuo Tomiyama. A Japanese view on Concurrent Engineering. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 9:69–71, 1995.

[155] Y. Ulmeda and T. Tomiyama. Functional reasoning in design. *IEEE Expert Intelligent Systems and their applications*, 12(2):42–48, March–April 1997.

[156] Karl T. Ulrich and Steven D. Eppinger. *Product Design and Development.* McGraw-Hill, New York, 1995.

[157] Yasushi Umeda, Masaharu Ishii, Masaki amd Yoshioka, Yoshiki Shimomura, and Tetsuo Tomiyama. Supporting conceptual design based on the function-behaviour-state modeler. *Artificial Intelligence for Engineering Design and Manufacture*, 10(4):275–288, 1996.

[158] Rainer Weigel and Boi Faltings. Abstraction techniques for configuration systems. In *Working Notes of AAAI 1996 Fall Symposium on Configuration*, Boston, USA, 1996. The working notes are also available as AAAI Technical Report FS-96-03.

[159] P. R. Wilson and M. J. Pratt. A taxonomy of features for solid modelling. In H. W. McLaughlin and J. L Encarnaça o, editors, *Geometric Modelling for CAD Applications*, pages 76–94. North-Holland, 1988.

[160] William H. Wood and Alice M. Agogino. A case-based conceptual design information server for Concurrent Engineering. *Computer-Aided Design Journal*, 28(5):361–369, 1996.

[161] Zhihui Yao. *Constraint Management for Engineering Design.* PhD thesis, Cambridge University Engineering Department, April 1996.

[162] R. Yokoyama and K. Ito. Multi-objective optimization in unit sizing of a gas turbine co-generation plant. *Journal of Engineering for Gas Turbines and Power*, 117(1):53–59, January 1995. Transactions of the ASME.

[163] F. Zhao and M. L. Maher. Using analogical reasoning to design buildings. *Engineering with Computers*, 4:107–122, 1988.