Fractal-Based Re-Design

A thesis submitted to the
University of Wales, Cardiff
for the degree of

Doctor of Philosophy

by

Yan Wu, BEng.

Manufacturing Engineering Centre
Cardiff University
United Kingdom

2006

UMI Number: U584963

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI U584963
Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Thesis Summary

Engineering conceptual design is a knowledge-intensive process that generates solutions to a product specification. It is a process that can benefit from past experience of similar designs. In reality however, designers often have limited time to build up the necessary experience and are, in any event, unlikely to become experts in all relevant fields. Hence there is a need to capture, store and reuse valuable knowledge. Currently available conventional CAD systems offer limited possibilities for the re-use of existing designs. Techniques from the field of Artificial Intelligence (AI) may be applied to aid the conceptual design phase, which is known as the area of intelligent computer-aided design.

The aim of this work is to identify and externalise design knowledge using a fractal-like model, to understand the role of design knowledge in conceptual design and to use design knowledge as a guide for every stage of concept development. This research provides a framework for supporting conceptual design, which uses the techniques of Case-Based Reasoning (CBR) and fractal theory, for reasoning about the design and development of computer-based design aids.

The framework is comprised of three parts. The first is case representation. This research proposes a new representation technique, Fractal-like Design Modelling (FDM), which integrates design knowledge in a graph-based form and has fractal-specific characteristics. The second is case retrieval. Based on FDM, the similarity between a new design and the existing designs is assessed by concurrently applying a feature-based similarity measure and a structure-based similarity measure. The third is case adaptation. With the help of fractal characteristics, an approach of adaptive design is developed by performance revision and by goal-oriented substitution. These three parts work together to achieve an automated, case-based, conceptual design method: Fractal-Based Re-design.

Dedication

This dissertation is dedicated to my family for their support during this work.

Acknowledgements

I would like to thank my supervisor Prof. D. T. Pham for his excellent supervision, continuous encouragement, and support. He is a brilliant supervisor.

Contents

Abstract ii
Dedication iii
Acknowledgements iv
Declaration v
Contents vi
List of Figures x
List of Tables xiii
Abbreviations xiv
List of Symbols xvi

Chapter 1 - Introduction **1**
1.1 Overview 1
1.2 Engineering design and product development 1
1.3 Research topic 3
1.4 Research objectives 5
1.5 Outline of the thesis 5

Chapter 2 - Literature Review **8**
2.1 Overview 8
2.2 A review of conceptual design research 9
2.2.1 Engineering design 9
2.2.2 Conceptual design 17
2.2.3 Intelligent design 18
2.2.3.1 Conceptual design modelling 20
2.2.3.2 Concept generation 33
2.2.3.2.1 Creative design 34
2.2.3.3 Concept selection 36
2.2.4 Artificial Intelligence techniques in design 37
2.2.5 Discussion 42

2.3 A review of Case-Based Reasoning for design 43
2.3.1 Represent a design case 45
2.3.2 Existing approaches to measuring similarity 49
2.3.3 Existing methods of adaptation 52
2.3.4 Discussion 56
2.4 Fractal based thinking 57
2.5 Summary 61

Chapter 3 - Fractal-like Design Modelling Using Attributed Graphs 63
3.1 Preliminaries 63
3.2 Fractal-like design modelling 63
3.2.1 Representing a design case in attributed graphs 64
3.2.2 Representing knowledge related to design objects 66
3.2.3 Representing design knowledge related to the design process in a fractal model 72
3.2.4 An illustrative example 80
3.3 Characteristics of FDM 84
3.3.1 Self-similarity 84
3.3.2 Self-organisation 86
3.3.3 Goal-orientation 86
3.3.4 Dynamism 87
3.4 Discussion 87
3.5 Summary 90

Chapter 4 - Similarity Assessment on Attributed Graphs in Design Case Retrieval 91
4.1 Preliminaries 91
4.2 Compared model generation 92
4.3 Similarity measure 93
4.3.1 Structure-based similarity measure (S_s) 95
4.3.2 Feature-based similarity measure (S_f) 100

4.3.3 Similarity assessment 106
4.4 An illustrative example 110
4.5 Discussion 120
4.6 Summary 122

Chapter 5 - Fractal-based Adaptation and Fractal-Based Re-design 123
5.1 Preliminaries 123
5.2 Fractal-based adaptation 124
5.2.1 Performance revision 126
5.2.1.1 An illustrative example 129
5.2.2 Goal-oriented substitution 134
5.2.2.1 An illustrative example 139
5.3 Fractal-Based Re-design (FBR) 143
5.4 Discussion 145
5.5 Summary 146

Chapter 6 - A Case Study of Fractal-Based Re-design in Automotive Body Design **147**
6.1 Preliminaries 147
6.2 Automotive body design 147
6.3 A case study of Fractal-Based Re-design in automotive body design 148
6.4 Discussion 163
6.5 Summary 165

Chapter 7 - Conclusions **166**
7.1 Preliminaries 166
7.2 Conclusions 166
7.3 Contributions 169
7.4 Recommendations 170

Appendix A An overview of the ICAD system 172

Appendix B Code for case base 175
Appendix C Code for similarity measure and case retrieval 187
Appendix D Code for performance retrieval 193
Appendix E Code for goal-oriented substitution 198
Appendix F Code for Fractal-Based Re-design 208

References 227

List of Figures

Chapter 2

Figure 2.1: Layout of axiomatic design process 13

Figure 2.2: The Pahl and Beitz model of the design process 16

Figure 2.3: A generalisation of feature-based representation 47

Figure 2.4: A feature-based representation of a bolt design 48

Figure 2.5: A graph-based representation of a bolt design 48

Figure 2.6: Similarity among graphs 53

Figure 2.7: Conceptual structure of FrMS 59

Chapter 3

Figure 3.1(a): A function model of a car body 69

Figure 3.1(b): Representation of a function model of a car body in case base 69

Figure 3.2(a): A feature model of a car body 70

Figure 3.2(b): Representation of a feature model of a car body in case base 70

Figure 3.3(a): A structure model of a car body 73

Figure 3.3(b): Representation of an assembly model of a car body in case base 74

Figure 3.4: An illustration of a basic fractal unit 77

Figure 3.5: An example of representing an engine compartment as a fractal 78

Figure 3.6: An overview of the fractal structure 79

Figure 3.7: A structure of a fractal-like design model 81

Figure 3.8: An example of representing a car body as a fractal-like design model 82

Figure 3.9: Self-similar fractals with different internal structures 85

Figure 3.10: A summary of fractal-like design model 89

Chapter 4

Figure 4.1: A compared model 94

Figure 4.2: Obtaining the subordinate relationships 98

Figure 4.3: Structure-based similarity measure on function model 98

Figure 4.4: Structure-based similarity measure on structure model 99

Figure 4.5: Features for comparison 101

Figure 4.6: Tolerance band 101
Figure 4.7: Grades for features 101
Figure 4.8: Feature-based similarity measure on feature model 103
Figure 4.9: Feature-based similarity measure on optional features 104
Figure 4.10: Feature-based similarity measure 105
Figure 4.11: Similarity assessment 108
Figure 4.12: An illustration of similarity assessment method 109
Figure 4.13: Car body represented by style lines 111
Figure 4.14: Input tolerance band 112
Figure 4.15: Input weights for features 112
Figure 4.16: Structure models of *d0*, *d1*, *d2* 116
Figure 4.17: Weights for feature-based similarity measure 118
Figure 4.18: Weights for structure-based similarity measure on function model 118
Figure 4.19: Weights for structure-based similarity measure on structure model 118
Figure 4.20: The retrieved case *d1* 119

Chapter 5

Figure 5.1: Comparison of performance 127
Figure 5.2: The process of performance revision 131
Figure 5.3: Performance selection 132
Figure 5.4: Re-design case selection 132
Figure 5.5: An illustration of the process of re-design 133
Figure 5.6: The process of input goal propagation 136
Figure 5.7: The process of generation of simplified GDG 137
Figure 5.8: Definition of generation of substitution 138
Figure 5.9 (a): Structure of the goals of a case of a saloon car body 140
Figure 5.9 (b): The goals of a case of a saloon car body 141
Figure 5.10: The GDG of a case of a saloon car body 142
Figure 5.11: The framework of FBR 144

Chapter 6

Figure 6.1 (a): The model of automotive body in the case base :Cb001 149

Figure 6.1 (b): The model of automotive body in the case base: Cb002 150

Figure 6.1 (c): The model of automotive body in the case base: Cb003 151

Figure 6.1 (d): The model of automotive body in the case base: Cb004 152

Figure 6.1 (e): The model of automotive body in the case base: Cb005 153

Figure 6.1 (f): The model of automotive body in the case base: Cb006 154

Figure 6.1 (g): The model of automotive body in the case base: Cb007 155

Figure 6.2: Inputs to query the FBR system 159

Figure 6.3: Selection of retrieval or adaptation 159

Figure 6.4: The result of retrieval 160

Figure 6.5: Selection of adaptation methods 160

Figure 6.6: Tracing simplified GDG in lisp-listener 161

Figure 6.7: The result of adaptation 162

List of Tables

Table 4.1: Comparison of the feature models of *d0* and *d1* and of *d0* and *d2* using S_f 111

Table 4.2: Comparison of the optional features of *d0* and *d1* and of *d0* and *d2* using S_f 114

Table 4.3: Summary of comparison results 114

Table 6.1: A design specification for the automotive design 158

Table 6.2: Detail of goal-oriented adaptation 161

Table 6.3: Comparison of the design specification and result of adaptation 164

Abbreviations

ABG	Archi Bond Graph
ACM	Artifact-Centered Modelling
AI	Artificial Intelligence
AIM-D	Axiomatic Information Model for Design
ARIZ	Algorithm for Inventive Problem Solving
BFU	Basic Fractal Unit
B-rep	Boundary Representation
CAD	Computer-Aided Design
CBR	Case-Based Reasoning
CN	Customer Needs
CSG	Constructive Solid Geometry
CSP	Constraint Satisfaction Problem
CUP	Conceptual Understanding and Prototyping
DMT	Design Mereotopology
DP	Design Parameters
DS	Design Schematics
EDIT	Engineering Design Integrated Taxonomy
FBS	Function-Behaviour-Structure
FDM	Fractal-like Design Modelling
FEBS	Function-Environment-Behavior-Structure
FR	Functional Requirements
FrMS	Fractal Manufacturing System
GA	Genetic Algorithm
GDG	Goal Dependency Graph
ICAD	Intelligent Computer-Aided Design
IDL	ICAD Design Language
MPG	Model Process Graph
MT	Mereotopology

PV	Process Variable
STEP	STandard for the Exchange of Product model data
VR	Virtual Reality

List of Symbols

D_s	Dissimilarity
k_i	Grades of features,
N_e	Number of equal features
N_m	Number of missing features
$N_{feature}$	Position of a design on the lists ranking its feature similarity with a given design
$N_{function}$	Position of a design on the lists ranking its function similarity with a given design
$N_{structure}$	Position of a design on the lists ranking its structure similarity with a given design
S	Overall similarity measure
S_f	Feature-based similarity
S_s	Structure-based similarity measure
w_1	Weight for feature similarity measures
w_2	Weight for function similarity measures
w_3	Weight for structure similarity measures

Chapter 1

Introduction

1.1 Overview

This chapter briefly introduces the research presented in this thesis. The specific topic of the current research is discussed. This is followed by the objectives of the research. The chapter ends with a description of the structure of the thesis.

1.2 Engineering design and product development

Engineering design is a systematic, intelligent generation and evaluation of specifications for artefacts whose form and function achieve stated objectives and satisfy specified constraints (Dym, 1994). Engineering design includes the roles of marketing, finance, planning, and overall management. There are many different models of the engineering design process, but they all include the following elements in some form or another (Kroll et al., 2001).

- A stage to identify and analyse a need prior to initiating conceptual design.
- A conceptual design stage to create new ideas that satisfy the need.

- Activities through which a concept is turned into an overall product or system layout.
- A stage to finalise the design details.

Conceptual design is considered a very important phase of the product development life cycle. It is a process of generating and implementing the fundamental ideas that characterise a product. Great opportunities exist at this stage. Conceptual design has a powerful impact on manufacturing productivity and product quality, as many manufacturing processes are indirectly determined at this stage. The concept generated at this stage affects the basic shape generation and material selection of the concerned product. It is difficult, or even impossible, to compensate for or to correct a poor design concept formulated at the conceptual design phase in the subsequent phase of detailed design.

Computers, which have been widely used in many areas in engineering, e.g. simulations, analysis, and optimisation, have few applications at the conceptual design stage. This is because information at the early stage of design is usually imprecise and incomplete, making it difficult to utilise computer-based systems. Artificial Intelligence (AI) is well suited to support conceptual design. The work in AI has the following directions: pursuing systemic and intellectual integration; building robots (both physical and computational); modelling rationality; supporting collaboration; enhancing communication; obtaining the broad reaches of knowledge

needed for intelligent action; deepening the mathematical foundations of the field. As a result of the application of AI techniques to conceptual design, an area of research known as *intelligent design* has emerged. This area of research examines how to provide computer support for modelling and automating the cognitive processes and knowledge representations which engineers apply to design problems.

1.3 Research topic

Engineering conceptual design is a knowledge-intensive process that generates solutions to a product specification. It is a process that can benefit from past experience of similar designs. In reality, however, designers often have little time to build up the necessary experience and are unlikely to become experts in all relevant fields. Hence, there is a need to capture, store and reuse valuable knowledge. At present, most common CAD systems can only help designers to construct geometric models step by step. They offer few possibilities for the reuse of existing designs (Wang et al., 2002). The need for computational frameworks to enable engineering product development, by effectively supporting the formal representation, capture, retrieval and reuse of product knowledge, becomes more critical (Szykman et al., 2001).

Fractal theory, which has been adopted in the field of manufacturing system design and analysis (Warnecke, 1993), promises to help address these design representation

and automation issues. The fractal structure has the potential to model combinations of different types of knowledge for different purposes, while the fractal specific characteristics can benefit the automation of the design process by providing design knowledge as a guide. This research was aimed at developing a systematic approach to intelligent design. In particular, the research was concerned with the application of case-based reasoning and fractal theory to conceptual design. It targeted the case-based design process and attempted to develop methods for providing computer support to automate it.

As the core of the system, a comprehensive design case representation, called Fractal-like Design Modelling (FDM), has been introduced. The design model integrates various aspects of design information, including knowledge related to design objects and design processes. Moreover, the design model has fractal characteristics, which can greatly benefit the process of case-based reasoning. The model is employed to assess the similarity between a new design and the existing designs, and to adapt a retrieved design to suit a new situation. The similarity of design models is measured by considering both the features and the structures of the design. The obtained design model is then adapted by the guidance of the integrated design knowledge according to different purposes of re-design. These include performance revision and goal-oriented substitution. In addition, the research also concerned user preference at every stage of the design process and attempted to develop a tool that can fulfil the designer's requirements.

1.4 Research objectives

The main objectives of this research were:

1) To identify and externalise design knowledge using a fractal-like model.

2) To understand the role of design knowledge in conceptual design.

3) To use design knowledge as a guide for every stage of concept development.

4) To provide a framework for supporting conceptual design, using the techniques of case-based reasoning and fractal theory, for reasoning about design and development of computer-based design aids.

1.5 Outline of the thesis

This thesis comprises six chapters and six appendices. The remainder of its structure is as follows:

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

Chapter 3 presents a fractal-like design modelling technique for representing the various aspects of design knowledge in attributed graphs.

Chapter 4 describes an approach for measuring the similarity of design models based

on the graph representation described in Chapter 3.

Chapter 5 addresses the fractal-based adaptation strategies and presents a systematic approach for automating the adaptive design.

Chapter 6 presents a case study demonstrating the application of the proposed approach to a conceptual design problem.

Chapter 7 presents the conclusions of the research and recommendations for further study.

Appendix A provides an overview of the ICAD system for conceptual design.

Appendix B gives the ICAD code for case representation.

Appendix C shows the ICAD code for the approach of graph-based similarity measure and case retrieval.

Appendix D lists the ICAD code for the adaptation approach of performance retrieval.

Appendix E presents the ICAD code for the adaptation approach of goal-oriented

substitution.

Appendix F contains the ICAD code for Fractal-Based Re-design, which integrates case retrieval, performance revision, and goal-oriented substitution.

Chapter 2

Literature Review

2.1 Overview

This chapter surveys the background literature relevant to the work presented in this thesis. The background of conceptual engineering design is reviewed from four perspectives. First, engineering design as an essential activity of product development is reviewed. Next, the conceptual design stage as a part of the entire design process is highlighted. At the same time, some critical issues of intelligent engineering design are discussed. These include the modelling for conceptual design, concept generation, and concept selection. Then some AI techniques applied in design are reviewed. Case-Based Reasoning (CBR) techniques have been applied to many aspects of the engineering design problem. This chapter also reviews the literature on the application of CBR techniques to engineering design. It will be shown that, while CBR has been applied to many aspects of design, there is considerable scope for research into using CBR techniques to support the conceptual phase of design. Finally, this chapter gives an introduction to fractal theory and its relevance to this research.

2.2 A review of conceptual design research

2.2.1 Engineering design

The UK-based Institution of Engineering Designers and the engineering design lecturer organisation SEED Ltd (Sharing Experience in Engineering Design) has defined engineering design as follows (Hurst, 1999).

"Engineering design is the total activity necessary to establish and define solutions to problems not solved before, or new solutions to problems which have previously been solved in a different way. The engineering designer uses intellectual ability to apply scientific knowledge and ensures the product satisfies an agreed market need and product design specification whilst permitting manufacture by the optimum method. The design activity is not complete until the resulting product is in use providing an acceptable level of performance and with clearly identified methods of disposal."

In other words, engineering design is such a process that uses scientific knowledge and methodologies to create an engineering product or a plan. An engineering design methodology provides knowledge including (Roozenburg & Eekels, 1995):

- Models of design and development processes, representing the structure of thinking and action in designing,
- Methods and techniques to be used within these processes, and

- A system of concepts and corresponding terminology.

The majority of the authors of the established design methodologies present the design activity as a linear process passing through a number of discrete phases. For example, French (French, 1985) splits the design process into four main phases: analysis of the problem, conceptual design, embodiment of schemes, and detailing. These phases are conducted one after the other in a logical sequence that leads the designer from a need (or set of requirements) to the final design solution. Feedback loops are often added to allow a return to previous phases if required (Daniel et al., 2004).

Two widely accepted methodologies of the engineering design process are discussed in this section.

Axiomatic design

Suh (Suh, 1990) argues that design involves four distinct aspects of engineering and scientific endeavour:

- The problem definition from a "fuzzy" array of facts and myths into a coherent statement of the problem;
- The creative process of devising a proposed physical embodiment of solutions;
- The analytical process of determining whether the proposed solution is correct or

rational;

- The ultimate check of the fidelity of the design product to the original perceived needs.

Axiomatic design defines the design process as the creation of synthesised solutions that satisfy requirements by mapping within the four domains of Customer Needs (CN), Functional Requirements (FR), Design Parameters (DP), and Process Variables (PV). The four-domain structure is schematically illustrated in Figure 2.1. As the mapping process is non-unique, the final outcome of the design depends on a designer's individual creative process. Two design axioms are introduced as the principles that the mapping technique must satisfy to produce a good design, and as a basis for comparing and selecting designs (Suh, 1990):

- Axiom 1 The Independence Axiom ---- maintain the independence of FRs.
- Axiom 2 The Information Axiom ---- minimise the information content of the design.

Axiom 1 states that during the design process, when going from the FRs in the functional domain to the DPs in the physical domain, the mapping must be such that a perturbation in a particular DP must affect only its referent FR. Axiom 2 states that, among all the designs that satisfy the Independence Axiom (Axiom 1), the one with minimum information content is the best design. Based on the two axioms of design, a number of derived corollaries are discussed in Suh's book (Suh, 1990).

As shown in Figure 2.1, axiomatic design views the design process as a stepwise decomposition process where design functions are elaborated to more detailed functions as the design progresses. In axiomatic design, the design process is not a one step process unless only one decomposition is required. For each domain, designers are encouraged to decompose the top level objects to detailed objects and apply two design axioms along the way.

Systematic design

The theory of systematic design is based on the notion that the design process must be carefully planned and systematically executed. According to Pahl and Beitz (Pahl & Beitz, 1996), a systematic approach:

- *Defines the goals* by formulating the overall goal, the individual sub-goals and their importance;
- *Clarifies the boundary conditions* by defining the initial and marginal constraints;
- *Dispels prejudice* to ensure the most wide-ranging possible search for solutions and to avoid logical errors;
- *Searches for variants*, that is to find a number of possible solutions or combinations of solutions from which the best can be selected;
- *Evaluates* based on the goals and the requirements;
- *Makes decisions*. This is facilitated by objective evaluations.

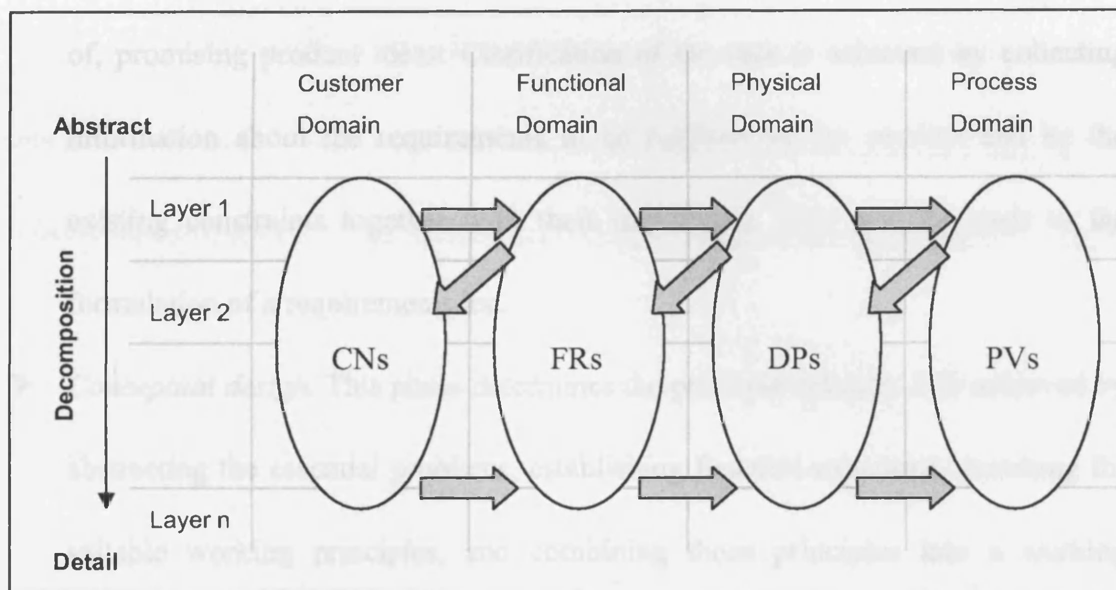


Figure 2.1: Layout of axiomatic design process

According to the systematic design theory, the design process must be split, first into phases and then into distinct steps, each with its own working methods. Pahl and Beitz (Pahl & Beitz, 1996) defines the following four main phases for the design process, as shown in Figure 2.2:

- *Product planning and clarifying the task.* Product planning, based on the company's goals, is the systematic search for, and the selection and development of, promising product ideas. Clarification of the task is achieved by collecting information about the requirements to be fulfilled by the product and by the existing constraints together with their importance. This activity leads to the formulation of a requirements list.
- *Conceptual design.* This phase determines the principle solution. It is achieved by abstracting the essential problems, establishing function structures, searching for suitable working principles, and combining those principles into a working structure. Conceptual design results in the specification of principle.
- *Embodiment design.* In this stage, designers, starting from a concept (working structure, principle solution), determine the construction structure (overall layout) with technical and economic criteria. Embodiment design results in the specification of layout.
- *Detail design.* This is the phase of the design process in which the arrangement, forms, dimensions, and surface properties of all the individual parts are finally decided, the materials are specified, production possibilities are assessed, costs

are estimated, and all the drawings and other production documents are produced. The result of the detail design phase is the specification of production.

Based on Pahl and Beitz's theory, Aleixos et al. (Aleixos et al., 2004) proposed a new five-step approach, which distinguishes in detail the tasks embedded in embodiment design. The division separated the management of conceptual information from transferring and integrating this conceptual data into a commercial CAD system tool. This gives the possibility to try other alternative solutions without generating the final design geometry.

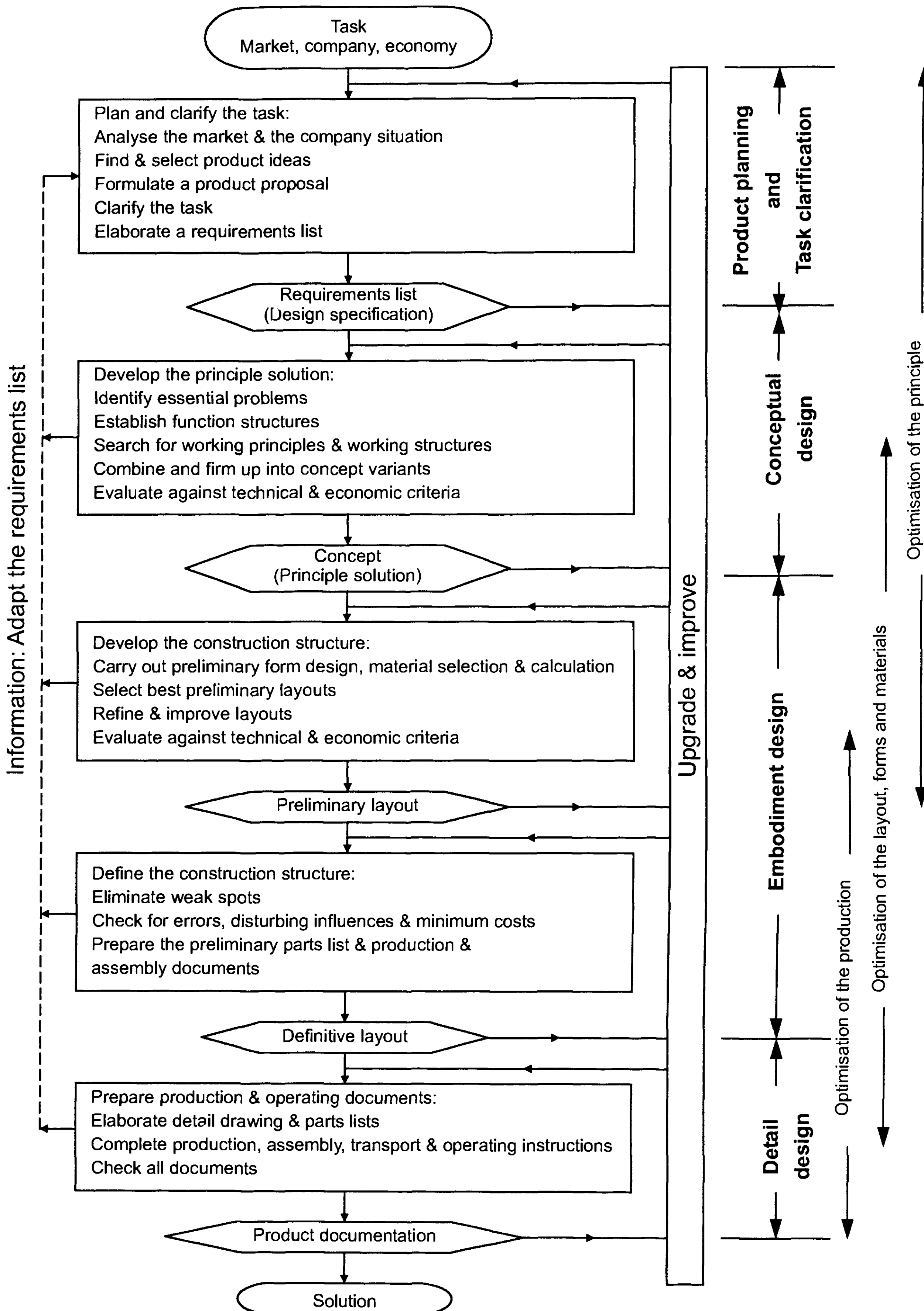


Figure 2.2: The Pahl and Beitz model of the design process (Pahl & Beitz 1996)

2.2.2 Conceptual design

Conceptual design refers to the early stages of design, when major decisions are still to be made. It takes the statement of the problem, brings engineering science, practical knowledge, production methods, and commercial aspects together, and generates broad solutions to it in a form referred to as "schemes" (French, 1985). It is the phase where the most important decisions are taken and where there is the most scope for striking improvements. Decisions made during conceptual design have significant influence on the cost, performance, reliability, safety, and environmental impact of a product. At this stage, information is very fuzzy and incomplete, which makes the design process quite difficult and challenging. The tasks involved in conceptual design are characterised by tentativeness, trial-and-error, exploration, ambiguity, and imprecision (Nakakoji et al., 2001).

In modern industry, with companies participating in global design chains, product design requires collaboration in a distributed environment. Extensive research has been carried out to develop prototype systems and methodologies for collaborative design (Kima et al., 2004; Sharma et al., 2006; Shyamsundar & Gadh, 2002). Work in infrastructure design, communication algorithms and geometric computing algorithms has been made to address the complexity of collaborative design activities and the specific requirements of CAD systems. Huang et al. (Huang et al., 2003) developed a system called **ProDefine** to support early product definition on the Internet. This

system supports collaboration, synchronously and/or asynchronously, through the Internet. Chen et al. (Chen et al., 2004) developed an Internet-enabled real-time collaborative assembly modelling system called **e-Assembly**. It allows geographically dispersed designers to jointly build an assembly model in real time over a distributed computing network such as the Internet. This e-Assembly system contributes to identifying and resolving assembly induced design conflicts arising from the outsourcing of design activities in the early stages of team design. Some of the previous work related to collaborative design has been reported in the literature (Fuh & Li, 2005).

2.2.3 Intelligent design

It has been estimated that design decisions made in conceptual design account for more than 75% of final product costs (Hsu & Liu, 2000). More importantly, a poorly conceived design concept might never be compensated for by a good detailed design. Researchers have focused their attention in developing tools and techniques that are able to support conceptual design activity. A standard for a good design tool (problem solver) has been defined as follows (Pahl & Beitz, 1996). The tool must:

- Have a sound and structured technical knowledge;
- Find an appropriate balance between concreteness and abstraction, depending on the situation;
- Be able to deal with uncertainty and fuzzy data;

- Continuously focus on the goals while adopting flexible decision making behaviour;
- Possess a further ability referred to as heuristic competence, which involves: activating goal-directed creativity, recognising importance and urgency, planning, guiding, and controlling their work.

Intelligent design is aimed at modelling and automating the cognitive processes and knowledge representations which human engineers apply to design problems. The cognitive processes include searching, reasoning or inference and optimisation. Collaboratively they operate on environment artefacts to carry out routine and creative design. The elements which make up an intelligent design theory include a process model, a set of appropriate knowledge representations and a research approach (Preston & Mehandjiev, 2004).

The process model contains several models which together explain different aspects of the process. These are: *Strategy* (a textual description of the process, in a manner similar to a general paradigm such as learning, searching, game playing, evolutionary, generative, or a stepwise approach), *Descriptive* (a textual and graphical description of the process, showing its main features, knowledge and control flows, and its general structure), *Formal* (these models provide mathematical rigor or make use of a logic to formalise the process and associated terms), and *Computable* (usually an algorithmic model, ready for translating it into supporting software).

Knowledge in design is used for manipulating design objects, for showing the next stage in given situations, for producing and interpreting design specifications and for controlling the design process. Representing knowledge relies on the developments carried out: for example, function-based modelling, grammar and ontology, geometry-based methods, a logical approach, graph-based modelling, generative representation, the design rationale, rule-based systems.

The research approaches used by design researchers fall into the categories of concept generation and concept selection

This section reviews the work in intelligent design from the following perspectives.

- Modelling the knowledge and complex interactions between various facets of a product.
- Generation and selection of feasible solutions.
- Decision making and trade-off for the feasible solutions.

2.2.3.1 Conceptual design modelling

Designers have limited time to build up experience in all relevant fields. Hence, there is a need to capture, store, and reuse knowledge. The goal of any knowledge representation is to enable and facilitate automated and semi-automated reasoning processes (Bo & Salustri, 1999). There are two important issues for modelling

(Vancza, 1999): the models should be re-usable and shareable, and the models should not only involve physical models, which cannot support commonsense reasoning and efficient design problem solving. A few design modelling methods are reviewed in this section.

Function-based modelling

Engineering design can be defined as mapping from a requirement specification at the functional level into a set of attribute values of concrete products. Functionality plays a crucial role in the conceptual design of engineering devices. 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. Function-based design modelling helps guide, constrain and solve the design problem by reasoning about the functions that the designs provide. Function is often integrated into a complete design approach.

The most well-known approach is "Function-Behaviour-Structure (FBS)". The FBS scheme has been applied to support design synthesis based on function (Qian & Gero, 1996). This scheme uses the relationships between the physical structure, behaviour, and functionality of designs to provide a basis for product development. The eight processes depicted in the FBS framework are claimed to be fundamental for all designs (Gero & Kannengiesser, 2004):

- *Formulation* transforms the design requirements, expressed in function, into behaviour that is expected to enable this function.
- *Synthesis* transforms the expected behaviour into a solution structure that is intended to exhibit this desired behaviour.
- *Analysis* derives the "actual" behaviour from the synthesised structure.
- *Evaluation* compares the behaviour derived from structure with the expected behaviour to prepare the decision if the design solution is to be accepted.
- *Documentation* produces the design description for constructing or manufacturing the product.
- *Reformulation type 1* addresses changes in the design state space in terms of structure variables or ranges of values for them if the actual behaviour is evaluated as unsatisfactory.
- *Reformulation type 2* addresses changes in the design state space in terms of behaviour variables or ranges of values for them if the actual behaviour is evaluated as unsatisfactory.
- *Reformulation type 3* addresses changes in the design state space in terms of function variables or ranges of values for them if the actual behaviour is evaluated as unsatisfactory.

Some researchers presented an extended model of FBS, called the Function-Environment-Behavior-Structure (FEBS) design model (Deng et al., 2000), in which the newly added "environment" stands for those environmental elements

contributing to the functions of the design. Bo and Salustri (Bo & Salustri, 1999) proposed a representation of product function, which has "function descriptor, input descriptor, output descriptor, how link, why link and value". O'Sullivan (O'Sullivan, 1999, 2002) applied a function-means map to model functional design knowledge, indicating how functions can be provided by physical means. Anthony et al. (Anthony et al., 2001) developed an approach, Conceptual Understanding and Prototyping (CUP), which integrates the description of formally represented engineering knowledge (function and behavior) with 3D graphical conceptual modeling. Kitamura and Mizoguchi (Kitamura & Mizoguchi, 2003) proposed an ontology-based method for capturing the knowledge of function decomposition. How functional reasoning has successfully established representation of function in design has been reported in the literature (Umeda & Tomiyama, 1997).

Geometry-based methods

In conceptual design various solutions are usually generated from a non-spatial perspective and lack detailed geometric structure. However, geometry is also important at the conceptual design stage. It is important to consider all critical geometric and spatial relationships that are relevant.

Geometry modelling focuses on representing the structural aspects of a product. Gero and Jupp (Gero & Jupp, 2003; Jupp & Gero, 2004) developed an approach to shape

and spatial representation for architectural design and the 2D building plan. The core idea is that design drawings can be uniquely characterised by the representation of embedded shape and spatial features. Each embedded shape and spatial feature is described by qualitative values and stored as a series of symbols in a 1D string and graphs.

STEP (STandard for the Exchange of Product model data) may also be considered as a tool to support knowledge representation (Denkena et al., 2005). STEP has been widely used for product data exchange and management. Its data models and methods provide a common basis for integrated collaboration processes in enterprises, allowing a holistic view that encompasses areas like design, engineering, testing, manufacturing, and quality assurance.

The realised product of engineering design is a 3D model. However, the traditional 2D modelling restrains the designer's creativity and imagination and hampers innovation. The development of 3D modelling and virtual manufacturing provides a good platform for conceptual design and innovation. Designers can start directly from a 3D concept to implement conceptual design, decide the framework of the product, then with the techniques of engineering analysis, simulation, Virtual Reality (VR), etc., analyse and evaluate the feasibility of the solution and the quality and reliability of the product. This design method makes full use of designer's intelligence and creativity, without the constraints of 2D modelling. With the newly available VR technology, it

is nowadays possible to build a design system that allows full three-dimensionality in all stages of the design process (Arangarasan et al., 2000). Spacedesign (Fiorentino et al., 2002) is an approach which uses task-specific configurations to support the design workflow from concept to mock-up evaluation and review. The first-phase conceptual design benefits from a workbench-like 3D display for free hand sketching, surfacing, and engineering visualisation.

Recently, aesthetic criteria have caught more attention in CAD (Fiorentino et al., 2002) (Juster et al., 2001). Aesthetic engineering and artistic shape optimisation (Sequin, 2005) needs more support from CAD tools. In a traditional CAD setting, a computer primarily serves as a precise drafting and visualisation tool, permitting the designer to view the emerging geometry from different angles and in different projections. Nowadays, a computer actively supports the creation of geometric shapes by procedural means and can even optimise a surface by maximising some beauty functionalities.

Grammar

Design grammar includes a vocabulary of engineering entities, a set of terminal symbols, a design start symbol, and knowledge about valid configurations of engineering entities (Andersson, 1993). There are two main categories of design grammar: graph grammars and shape grammars.

Andersson (Andersson, 1993) introduced a structure and components of a vocabulary for conceptual design of mechanical products. The components of this vocabulary consist of engineering concepts represented as engineering entities, using Conceptual Graphs and classified into taxonomies. Each engineering entity is defined by its position in the taxonomy and by a type description. This vocabulary can be utilised for generating the resulting design descriptions of the conceptual design phase and for representing both syntactic knowledge and interpretative knowledge. Semantics and syntax have also been used by some researchers. Ding and Gero (Ding & Gero, 2001) developed a syntax-semantics model to interpret style, with semantics to be the implicit properties of style and syntax to be the explicit representation of style. Deng (Deng, 2002) proposed a semantic and syntactic representation of mechanical function and behaviour.

Shape grammars derive designs in the language they specify by successive application of shape transformation rules to some evolving shape, starting with an initial shape. Shape grammars are essentially a rule set defining how shapes in a set can be modified. In addition, shape grammars allow labels to be associated with shapes to carry non-geometric information and guide the generation process. Finally, their parametric nature allows the same small and finite rule set to generate an infinite number of designs, allowing a generative system to explore a wide variety of designs (Agarwal & Cagan, 2000). Shape grammar has been used to represent engineering knowledge and to analyse designs (Agarwal & Cagan, 2000). McCormack et al.

(McCormack et al., 2004) applied shape grammars as a method for encoding the key elements of a brand into a repeatable language, which is used to generate products consistent with the brand.

Ontology

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 (Hsu & Woon, 1998). Noy and McGuinness (Noy & McGuinness, 2001) defines an ontology as a formal explicit description of concepts in a domain of discourse (classes or concepts), properties of each concept describing various features and attributes of the concept (slots or roles or properties), and restrictions on slots (facets or role restrictions). Ontology, together with a set of individual instances of classes, constitutes a knowledge base. A design ontology has an intentional semantic structure that defines and arranges all related notions (Horvath et al., 1998).

A certain amount of research on the use of ontology in design has been conducted (Setchi et al., 2005). Borst (Borst, 1997) developed an ontology collection called PHYSSYS that covers a wide, multidisciplinary range of physical systems and their engineering. This collection contains different types: highly generic ontologies (mereology, topology, and systems theory), base ontologies valid for a whole field

(e.g. technical components, physical processes, representing natural categories or viewpoints within a broad field), and domain ontologies (specialisations of base ontologies to a specific domain, e.g. thermodynamics). Horvath et al. (Horvath et al., 1998) introduced ontologies for formalising conceptual design concepts, which include structure and shape as well as functionality. Kitamura and Mizoguchi (Kitamura & Mizoguchi, 2003) developed functional ontologies including a device-centered ontology and a functional concept ontology, aiming at systematisation of functional knowledge for design. Ahmed (Ahmed, 2005) argued that the descriptions of designing a particular component or assembly could be classified in four ways: steps of the design process; components or assemblies; the function; the issues; or any combination of these. Based on this, he identified four taxonomies, which are design process, product, functions, and issues. These form the taxonomies for ontology for engineering design referred to as Engineering Design Integrated Taxonomy (EDIT).

Logical approach

Salustri (Salustri, 1996) attempted to use logic to describe the structure of design. He developed a framework named Artefact-Centered Modelling (ACM) to partition the problem of describing design into manageable components. ACM partitions the overall design endeavour by abstracting both by function and by structure. These abstractions form the axes of a two-dimensional matrix of design aspects. Based on

the ACM, the Axiomatic Information Model for Design (AIM-D) was developed. It provides formal bases for quantities, features, parts and assemblies, systems and sub-assemblies, which help designers to think about design problems in a more structured manner, and to form the logical foundations for tools to aid designers in their daily task. Salustri (Salustri, 2002) introduced the use of a logical theory, Design Mereotopology (DMT), in product modelling and spatial reasoning of designed products. Mereotopology (MT) is a branch of logic dealing with the qualitative formalisation of two fundamental relationships between entities: parthood (i.e. one entity being part of another) and connection. DMT provides a framework for improved understanding of product modelling knowledge.

Graph-based modelling

Graphs are popular representations in the conceptual design stage. They have been used to model all aspects of a product (Castano et al., 1998). Conceptual graphs can represent both functional and manufacturing-related information. Conceptual graphs have been applied for representing assemblies, components, features, low level geometric objects, and constraints (Salomons et al., 1995). Qian and Gero (Qian & Gero, 1996) used a graph, which consists of five finite sets for elements, attributes, relationships, operations, and processes, to describe a design structure. Al-Hakim et al. (Al-Hakim et al., 2000) applied graph theory to represent a product and the relationships between its components. Zha and Du (Zha & Du, 2001) utilised a

Knowledge Petri net graph with objects scheme to uniformly model a mechanical system or an assembly and its design process. They represented the hybrid design object model in terms of a four level hierarchy: function-behaviour, structure, geometry, and feature. The structure model is described as a place-transition based component-connector or part-joint multilevel hierarchical graph, while the functions, behaviours, geometries, features, and constraints are embedded as objects in such a hierarchy, and their causal relations are described by the corresponding Knowledge Petri net graphs. Salustri and Parmar (Salustri & Parmar, 2003) introduced Design Schematics (DS), which is a diagramming method intended to capture product information at early design stages. It is based on concept maps. Gero and Tsai (Gero & Tsai, 2004) used bond graphs as a foundation for the development of a representation of buildings and their uses, called Archi Bond Graphs (ABGs). Bond graph modelling has also been applied to air pump system design (Seo et al., 2005).

Generative representation

Generative representation, which is different from the traditionally parameterised representations, does not encode complete design concepts but rather rules on how to develop, or "grow" these designs. This representation method has been mostly applied to evolutionary design. This is because in evolutionary design, parameterised representations are inadequate to seek novel designs, and they have some scaling-up problems as the design application problems increase in size and complexity. These

generative representations (Kicinger et al., 2005b) improve the scalability of evolutionary design systems and produce novel designs exhibiting interesting and qualitatively different patterns from known designs (Kicinger et al., 2005c).

Design rationale

Design rationale encompasses a broad context surrounding product development processes, including information about decisions, why they have been made, as well as relationships or dependencies that may link decisions either to part of the product representation (a function, artefact, etc.), or to other decisions (Szykman et al., 2001). Design rationale is considered to play an important role in design modelling. A survey on the research on design rationale has been reported (Hu et al., 2000).

There are two fundamental and complementary representations of design rationale. First is the notion of design rationale as the recording of the design intent of an artefact. For example, in traditional mechanical design, rationale might include a functional description, geometric or assembly constraints, and performance criteria. Second is design rationale as a record of the design process, the communications among agents, the decision-making that occurs as well as the decision-making process.

A generic structure of design rationale systems consisting of three main layers has

been identified as follows (Lee, 1997):

- *Decision layer* characterising the decision process. The decision layer contains five sub-layers: argument, alternative, evaluation, criteria, and issue.
- *Design artefact layer* containing information relating the components of an artefact and linking these to the decision layer.
- *Design intent layer* representing information about the design decisions, e.g. requirements, strategies, and goals.

Model process history has also been integrated in the design modelling. Hayes and Regli (Hayes & Regli, 2001) attempted to unite traditional CAD and solid model data structures with a representation of the temporal design process. They presented a representational formalism called Model Process Graphs (MPGs). MPGs integrate a model's description with a model of temporal changes that occur during the design process. They argued that model process graphs can be used as a substrate on which design history, intent, and rationale can be captured.

Other methods

A number of conceptual design modelling methods have been reported in the literature (Salustri, 2001, 2005; Seebohm & Wallace, 1998; Zavbi & Duhovnik, 2000). Amongst the approaches taken are the use of natural language, physical laws, and rule-based systems. These methods address different aspects of the design modelling

of conceptual design and general engineering design.

2.2.3.2 Concept generation

An "ideal" approach for concept generation should be a process of repeated divergence and convergence (Liu et al., 2003). Liu et al. presented an approach consisting of a series of generation and evaluation rather than a single step of generation and evaluation. Their approach consists of three levels of solution abstraction, namely topological solution, spatial configuration, and generic physical embodiment level. Expansion of solutions consists of three synthesis processes. The processes of narrowing down solutions involve applying sets of heuristics to each level. They argued that such an approach should increase the effectiveness of the exploration of concepts with minimum compromise to the richness of the solution space explored.

Design can be divided into two groups: routine design and non-routine design. Routine design is a design process based only on selection or on modification. In both cases, no changes in the representation space occur. Non-routine or creative design is a conceptual design process which is based on innovation, invention, or discovery. In all these cases, changes in a representation space occur. Thus, there are two major differences between the routine and creative design: the number of changes of the representation space and the nature of inference. There are no changes in the

representation space for routine design, and at least one change for non-routine or creative design. Routine design typically employs deductive inference (selection and modification), while creative design employs inductive inference (innovation, invention and/or discovery) (Arciszewski et al., 1995).

2.2.3.2.1 Creative design

Innovation plays an important role in conceptual design. The essence of innovation in conceptual design is to discover new ideas, especially when the current products cannot satisfy the user requirements. Creative design involves not just a search within a defined space but also the introduction of either new variables or new schemas - a process called exploration (Gero, 1996). In other words, a design process is creative when it explores not only the values of attributes (decision variables) within individual design spaces but also develops the number of these attributes, i.e. when changes in the representation space occur.

According to the innovation levels, conceptual design can be distinguished by five major paradigms (Arciszewski et al., 1995). This classification is based on the taxonomy proposed by Altschuller (Altshuller, 1969) and modified and adapted by Arciszewski et al.

- *Selection*: the design concept is produced by selecting it from a class of known concepts in a given engineering domain.

- *Modification*: the design concept is produced as a combination and/or modification of known design concepts from a given domain. The modification process is based on a deterministic or random generation process.
- *Innovation*: the design concept is produced as a combination of known concepts from a given domain and other domains.
- *Invention*: the design concept is produced as a combination of known concepts from a given domain and from new concepts based on a new technology, which have been recently introduced.
- *Discovery*: the design concept is produced as a combination of known concepts from a given domain and new concepts based on new scientific principles.

Redistribution of functions is also considered a creative technique (French, 1985), because redistribution of functions among parts can often make improvements in schemes.

TRIZ, first developed by Altshuller (Altshuller, 1984), is a human-oriented knowledge-base systematic methodology of inventive problem solving. TRIZ uses a relatively small number of heuristics for solving inventive technical problems. These main heuristics and instruments include Preliminary Analyses, Contradiction Matrix, Separations Principles, Substance-Field Analysis, Standard Approaches to Inventive Problems, Algorithm for Inventive Problem Solving (ARIZ), Agents Method, etc. The detail of these has been reported (Mann, 2002; Savransky, 2000). Interest in the

principle of TRIZ has result in the development of a number of approaches to innovative design (Pham & Liu, 2006; Pham et al., 2006).

2.2.3.3 Concept selection

Concept selection is a decision making process in nature. A critical analysis and evaluation of current engineering design methodologies from a decision making perspective has been reported in the literature (Ng, 2006). The selection procedure usually involves two steps, namely elimination and preference.

A language-based framework (White, 1995) for the evaluation of product design has been developed. Deng et al. (Deng et al., 2000) proposed a generic functional design verification model for conceptual design. In their work, design verification is achieved by identifying input and output design variables, developing a variable dependency graph, propagating constraints over the variable dependency graph, and checking the values of the design variables against these constraints. Dezfuli (Dezfuli, 2001) proposed a value-based approach to conceptual design decision making. The approach introduces the notion of design values and design objectives plus their importance in guiding the design decision making process. Designers develop their value structures through a design objective structuring process and use them to identify, expand, and search through the design context space. The objective structuring process provides valuable insights into the design process and points out

those aspects of the requirements which are important for the designer. It also helps the designer to avoid unnecessary searches among alternative concepts which do not provide value to the design process. A framework was also developed for an engineering conceptual design process based on some important properties of design values and design objective structures. This framework provides the means for designers to incorporate the uncertainties of design alternative concepts into the process in a formal way and provides an evaluation method for designers to compare different design concepts with each other in a more consistent way.

2.2.4 Artificial Intelligence techniques in design

Engineering design needs to be formulated and supported by specific design methods. Design methods may help design in the following ways (French, 1985):

- By increasing insight into problems, and increasing the speed of acquiring insight;
- By diversifying the approach to problems;
- By reducing the size of the mental steps required in the design process;
- By prompting inventive steps, and reducing the chances of overlooking them;
- By generating design philosophies (synthesising principles, design rationales) for the particular problem in question.

Artificial Intelligence (AI) has been playing an important role in the field of

engineering design. The scientific and practical aims of AI (Doyle & Dean, 1996) are: constructing intelligent machines; formalising knowledge and mechanising reasoning; using computational models to understand the psychology and behaviour of people, animals and artificial agents; making working with computers as easy and as helpful as working with skilled cooperative, and possibly expert people. AI can learn new concepts, reason, and draw useful conclusions about a design problem; understand the natural languages of designers; perceive and comprehend a visual scene (Wang et al., 2002).

This section reviews the use of a few frequently used AI techniques in engineering design.

Constraint satisfaction problem

Constraint processing is concerned with the development of techniques for solving the Constraint Satisfaction Problem (CSP). CSPs involve finding values for variables subject to restrictions on which combinations of values are acceptable. A large number of problems in engineering design can be formulated as CSPs. For design problems, starting from a list of design requirements, design objectives and important factors in a successful design are collected in an unstructured manner. Each requirement can be formulated as a testable constraint rule. The advantage of CSP is that it is a reasoning model that both provides modelling and solves a problem within

the same framework (Sqalli et al., 1999). O'Sullivan proposed a constraint-based approach to providing support to a designer during conceptual design (O'Sullivan, 1999, 2002).

There has also been a focus on research and applications that integrate CSP with CBR (Sqalli et al., 1999). Purvis and Pu (Purvis & Pu, 1995) investigated a methodology which formalises the adaptation process using constraint satisfaction techniques. They represented each case as a primitive CSP with additional knowledge that facilitates retrieving and applied an existing repair-based CSP algorithm to combine these primitive CSPs into a globally consistent solution for the new problem.

Agent-based approach

Decomposition and parallel execution in collaborative design naturally lend themselves to an agent-based approach. A design process can be considered as a discrete-event system occurring as the result of multiple "agents" acting towards a common general goal, with each agent having its own priorities, context, and domain knowledge (Salustri, 2000). Agents have incomplete information and limited reasoning capabilities and resources. In a community of agents, there is no global control and centralised data and the computations are asynchronous (Sycara, 1998). The motivations for applying multi-agent systems are: to solve problems that are too large for a centralised agent to solve, to provide solutions to problems that can

naturally be regarded as a society of autonomous interacting components-agents and to provide solutions in situations where expertise is distributed. The concept and technology of agents has given considerable support to distributed design.

Campell et al. (Campbell et al., 1999) introduced a new design generation theory known as A-Design, which is an agent-based and adaptive strategy for performing conceptual engineering design. The methodology has four distinct subsystems: an agent architecture, a multi-objective design selection scheme, a functional representation for electro-mechanical systems and an iterative-based algorithm for evolving optimally directed design states. Cvetkovic and Parmee (Cvetkovic & Parmee, 2002) presented the use of software agents within an interactive evolutionary conceptual design system. Several different agent classes are introduced, including search agents, interface agents, and information agents.

Evolutionary algorithms

Genetic Algorithms (GAs) model natural selection and the process of evolution. Conceptually, genetic algorithms use the mechanisms of inheritance, genetic crossover, and natural selection in evolving individuals that, over time, adapt to their environment. They also can be considered as a search process, searching for better individuals in the space of all possible individuals. Also, genetic algorithms have increasingly been applied in engineering design. Basically, genetic algorithms have

been considered as tools for optimisation and parameter tuning in engineering design. Genetic algorithms have been used to generate candidate solutions to particular types of design problems. Several advanced genetic algorithms have been introduced, which have proved to be efficient in solving difficult design problems (Renner & Ekart, 2003). A tool called Emergent Designer has been developed, which involves evolutionary algorithms to represent engineering systems and their related design processes (Kicinger, 2004; Kicinger et al., 2005a).

Machine learning

Both learning and conceptual design processes are based on performing various forms of inference. All experience, therefore, from machine learning research that studies learning as an inferential process is relevant to design and can be used for developing a formal model of design processes (Arciszewski et al., 1995).

Model-based reasoning

Model-based reasoning can guide the design process by evaluating partial designs, or alternatively, can constrain the space of feasible design solutions (Vancza, 1999).

Rough set theory

Many approaches to the handling of incomplete information have been developed. These include fuzzy set theory, rough set theory, and Dempster-Shafer theory of evidence (Alisantoso et al., 2005). Alisantoso et al. (Alisantoso et al., 2005) proposed a rough set-based approach to early design analysis.

2.2.5 Discussion

Conceptual design is a very important phase of engineering design process. It has been shown in the previous sections that a number of tools and techniques have been developed to support conceptual design activity. However, most of these tools and techniques offer few possibilities for the reuse of existing designs. Case-Based Reasoning (CBR) has the potential to support design by reminding designers of previous solutions that could help in new situations (Maher et al., 1995). An advantage of CBR is that it starts from once satisfactory solutions and most of the design knowledge is available after a design case has been retrieved. There is considerable scope for research into using CBR techniques to support the conceptual phase of design. Case-based approaches to supporting engineering design have been reported in the literature and will be reviewed in the next section.

2.3 A review of Case-Based Reasoning for design

The reliance on past experience has motivated the use of Case-Based Reasoning techniques. A CBR system stores past problem solving episodes as cases which can be retrieved to help solve a new problem. CBR is based on two observations about the nature of the world: (1) the world is regular and similar problems have similar solutions; and (2) the types of problems encountered tend to recur.

A number of researchers have applied CBR to engineering design problems. Bilgic and Fox (Bilgic & Fox, 1996) discussed similarity based retrieval in engineering design. They focused on how requirements, i.e. goals and constraints, can be used to dynamically retrieve relevant cases from a case library, and how cases in the library should be represented to support this style of dynamic indexing. Leake et al. (Leake et al., 1999) argued that CBR fits naturally into a new mode of knowledge management that not only tracks where documents are but tracks how they are used and where they are needed to access multiple information sources to provide the right information at the right time. They demonstrated their approach in automotive body design. CBR has also been widely applied to various specific engineering problems. Qin and Regli (Qin & Regli, 2000, 2003) presented a case study of how to apply CBR to a specific engineering problem, mechanical bearing design. Waheed and Adeli (Waheed & Adeli, 2005) presented the use of CBR in steel bridge engineering. A few efforts have been made to build a fully automated gripper design system based on CBR (Gourashi, 2003;

Pham et al., 2005).

Traditionally, CBR systems draw their cases from a single local case base tailored to their task. However, when a system's own set of cases is limited, it may be beneficial to supplement the local case base with cases drawn from external case bases for related tasks. The effective use of external case bases requires strategies for multi-case-base reasoning (MCBR): (1) for deciding when to dispatch problems to an external case base, and (2) for performing cross-case-base adaptation to compensate for differences in the tasks and environments that each case base reflects (Leake & Sooriamurthi, 2002, 2003). Al-Shihabi & Zeid (Al-Shihabi & Zeid, 1998) used multi-case adaptation and case built-in adaptation knowledge to produce a design plan for a new design problem.

CBR has also been used in combination with other AI techniques. Rosenman (Rosenman, 2000) developed a case-based model of design, using an evolutionary approach, for the adaptation in spatial layout design. Saridakis et al. (Saridakis et al., 2006) developed a system that retrieves existing design solutions, by using a fuzzy case representations and neural-network-based retrieval mechanism. The system has been used in structural design.

Representing a case, measuring the similarity of the cases and adapting a case are key issues in CBR. These will be discussed in detail in the remaining of this section.

2.3.1 Represent a design case

One of the main difficulties in supporting conceptual design is the complexity of modelling the different aspects of a product. The common representation of a design case includes feature-based representation, graph-based representation and geometric representation.

Feature-based representation

Feature-based representation is the most common form of representation of a design case. Each case is described by a set of attributes and each attribute takes a value. A feature is an information unit describing a region of interest of some characteristics of a product. It can be an individual attribute, a set of attributes or one or more derived attributes. Attributes define the vocabulary for explaining a design; values identify the specific information for each design. In most feature-based representation cases, design is formalised as classes. A generalisation of this representation is shown in Figure 2.3.

Bilgic and Fox (Bilgic & Fox, 1996) defined individual cases with a finite number of attribute-value pairs and a retrieval context with a finite number of constraints on the attributes.

Figure 2.4 shows a design of a bolt which helps illustrate how this representation paradigm is used. In the example, the list of features determines how to describe a bolt design. As shown in the figure, a bolt can be expressed by its nominal diameter, material and so on. A specific bolt is determined by the specific values for each of the attributes.

Graph-based representation

Graph-based representation focuses on the relationships between the elements of a design. Usually, in a graph-based representation, nodes represent distinct features and edges represent associations among features.

An example of a graph-based representation of a bolt is shown in Figure 2.5. The function and structure of a bolt is represented as labels of the nodes in the graph. The links in the graph represent dependencies among function and structure. The attribute "connecting components" is embodied by the values of head, shank, thread and end. By representing a design case in this way, nodes and relationships between nodes determine a bolt design.

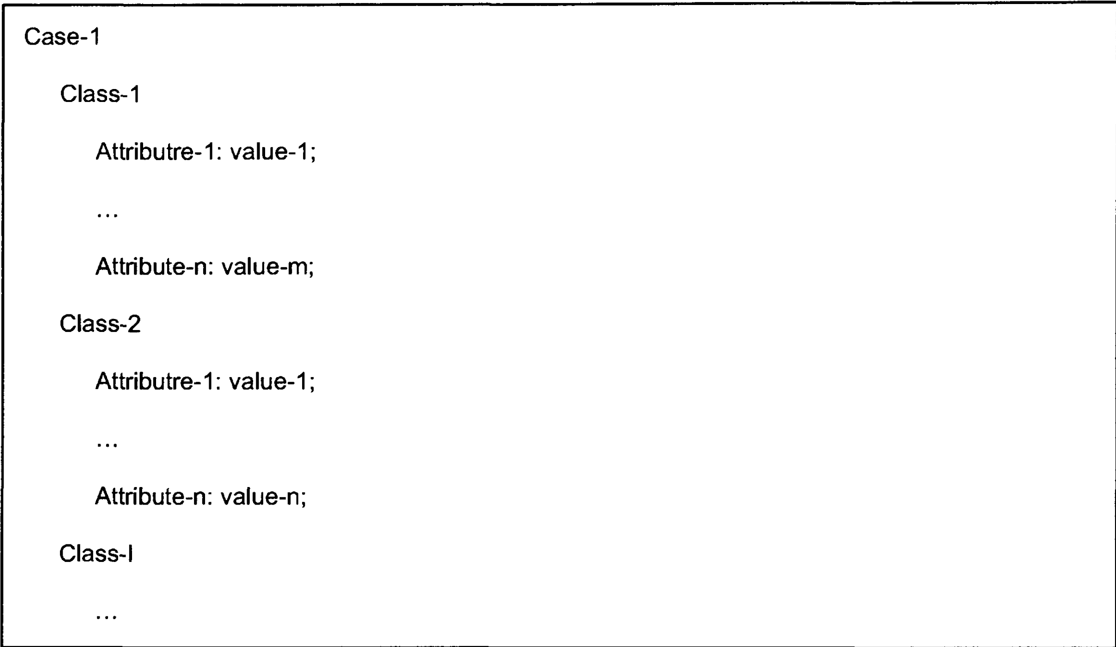


Figure 2.3: A generalisation of feature-based representation

Case: Bolt 001

FUNCTION:

Function-1: Connecting components

Function-2: Tightening components

BEHAVIOUR

Maximum working load: 70550N

STRUCTURE

Nominal diameter: 33mm

Bolt head shape: cylindrical

Bolt head thickness: 25mm

Diameter of bolt head: 50mm

Diameter of shank: 27mm

Length of shank: 20mm

Length of thread: 32mm

Pitch of thread: 2mm

Material: steel

Figure 2.4: A feature-based representation of a bolt design

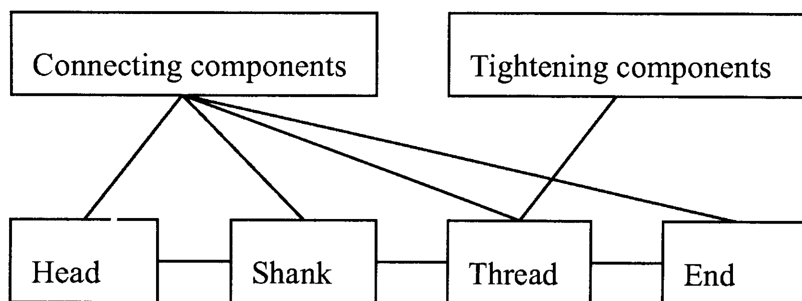


Figure 2.5: A graph-based representation of a bolt design

Geometric representation

Geometric representation is a direct way of carrying design information. It enables a design to be described as a geometric model which includes 2D or 3D geometric shapes. General representations of geometric shapes include: B-rep (Boundary representation), CSG (Constructive Solid Geometry), variational geometry and feature representations (Hsu & Woon, 1998). Geometric representation focuses on representing the structural aspects of a product. It requires intuitive experience for the user to understand and recognise the design. Francois and Medioni (Francois & Medioni, 1996) presented a symbolic and structured shape description model which can be used for the efficient indexing and retrieval of 2D or 3D generic object shapes.

Every representation technique has its own focus. In order to build a comprehensive design model, it is necessary to integrate various aspects of design information.

2.3.2 Existing approaches to measuring similarity

Methods of similarity assessment have been reported in the literature (Smyth & Keane, 1998). Bridge (Bridge, 1998) classified the approaches to measuring the similarity of object representation as geometric, structural, and feature-based.

Geometric approach

With a geometric approach, a set of features is extracted from the structural representation, which is used as an *n*-dimensional vector to which distance measures can be applied. Assume there are two objects with *n* features each: *O1(x1, x2...xn)* and *O2(y1, y2...yn)*. Similarity between these two objects can be measured by calculating the Euclidean distance: $\sqrt{(x1-y1)^2+(x2-y2)^2+...+(xn-yn)^2}$. Relatively simple distance measures include the Euclidean distance, Manhattan distance, and Hausdorff distance (Ohbuchi et al., 2002). Ohbuchi et al. employed the Euclidean distance and the elastic-matching distance as the measures of distance between pairs of feature vectors in their work. They investigated this method to compare the shape similarity of 3D models, which was to compute a set of shape features from a given model, as well as the distance between those pairs of shape features.

Structural approach

With a structural approach, similarity is measured by graph matching (Bespalov et al., 2003; El-Mehalawi & Miller, 2003; Hilaga et al., 2001; Iyer et al., 2003; Le et al., 2004). Cost-based Distance Measurement is frequently adopted for this purpose (Francois & Medioni, 1996; Papadopoulos & Manolopoulos, 1999). This method involves modifying the graph for an object to transform it into the graph for the object with which it is to be compared. The number of the required modifications is then

taken as the similarity measure. An example of comparing graphs using this method is illustrated in Figure 2.6. Three graphs *G1*, *G2*, and *G3* are shown in the figure. It is obvious that *G1* is more similar to *G2* than to *G3*. This is because only one edge needs to be added to *G1* in order to obtain *G2*, whereas one edge and one node are needed in order to obtain *G3*.

Feature-based approach

In a feature-based approach, objects are represented by sets of features and measuring similarity is based on feature commonality and differences. The similarity of the features is usually measured by numerical weighting methods (Castano et al., 1998). Weights are assigned to each feature and the similarity is the sum of a weighted number of equal features. Bilgic and Fox (Bilgic & Fox, 1996) counted the occurrences where the two cases satisfy the same constraints and normalised it using the weights. Tversky's theory of similarity, as described in (Keane et al., 2001), characterised the similarity between two entities, *a* and *b*, as being a weighted sum of a function of the identical features of *a* and *b* and a function of the distinctive features in each of the entities. The model is characterised as:

$$s(a, b) = \theta f(A \leftrightarrow B) - \alpha f\,(A - B) - \beta f\,(B - A),$$

where A and B represent the set of attributes that respectively make up the entities a and b; $(A \leftrightarrow B)$ represents the set of attributes that are common to A and B; (A - B) represents the distinctive features in A while (B - A) represents the distinctive features

in B; and θ, α, and β are factors of importance.

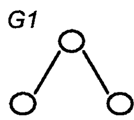
The existing methods compare different aspects design. In order to have a comprehensive similarity measure, it is helpful to involve various design knowledge in the process of similarity assessment.

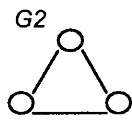
2.3.3 Existing methods of adaptation

In general, the methods relevant to case adaptation vary in different tasks or problems to be solved. Adaptation can substitute some components of a previous solution, or modify the overall structure of an old solution. Existing methods of adaptation can be classified as follows.

Human intervention

The simplest method of adaptation is human intervention, in which the designers are responsible for modifying the design according to their knowledge.





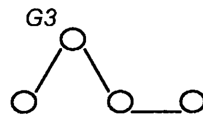


Figure 2.6: Similarity among graphs

Substitution method

As the name implies, this method relies on the substitution of values or components relevant to the new problem in the retrieved case. The structure of the new solution remains unchanged. Some researchers believe that substitution adaptation is the best means for automatic knowledge acquisition (Jarmulak et al., 2001). Substitution adaptation is applied for both nominal and numerical values and is suitable for decomposable design problems, in particular formulation and configuration.

Transformation method

Transformation method deals with the structure of the solution. Rules or procedures are used to transform a selected case into a new solution. It supports the reorganisation of solution elements and permits the addition and deletion of such elements under certain conditions. Generally, transformational adaptation systems employ a mixed set of adaptation operators and transformation rules. Bergmann and Wilke (Bergmann & Wilke, 1998) developed a formal model of transformational adaptation. The model is based on the "quality" of a solution to a problem, where quality signifies a more general sense and denotes some kind of appropriateness, utility, or degree of correctness.

Derivational replay

Derivational replay assumes that the retrieved case includes the method or procedure used to generate the solution in the case description and that the same method is

reused for the specifications of the new problem. Derivational replay has been applied in some researchers' work (Rivard & Fenves, 2000).

Multiple case combination

Recent research has demonstrated the power of delivering solutions through retrieval, adaptation and subsequent composition of multiple cases. This leads to multiple case combination, where design cases or components from multiple cases are combined to provide new design solutions. Newer approaches indicate that it is helpful to compose a solution from parts of several old cases. This is possible if the solution consists of different parts which can be adapted more or less independently. It will be effective if few conflicts exist between these components so that a change in one component will not have many side-effects on the other components.

Hierarchical adaptation

Hierarchical adaptation is another development of adaptation (Bergmann & Wilke, 1995; Smyth & Cunningham, 1992). Cases are stored at several levels of abstraction and the adaptation is performed in a top-down fashion. At first, the solution is adapted at the highest level of abstraction. The solution is then refined in a stepwise manner and the required details are added. Hierarchical adaptation reuses either a single case or different cases for different levels of abstraction or refines different details of the solution.

Instead of using these methods on their own, a combination of these approaches is often adopted.

Different strategies have been applied to guide the process of adaptation. The strategies used for the modification and evaluation of a design case include constraint satisfaction, model-based reasoning, rule-based reasoning, heuristic reasoning and qualitative reasoning. Some researchers argue that the more radical the adaptation, the greater the danger of losing the quality of the original design, as adaptation changes a once satisfactory design. Whatever adaptation strategies are used, extensive domain specific knowledge is required to guide the process of adaptation.

2.3.4 Discussion

The application of CBR to many aspects of engineering design has been presented in this section. It can be seen that representing a design, measuring similarity of designs and adapting a design all hinge on various aspects of design information. A comprehensive design model is needed to integrate various aspects of design information, and design knowledge needs to be effectively used to measure the similarity of designs and to guide the process of adaptation. Fractal theory, which has been adopted in the field of manufacturing systems design and analysis (Warnecke, 1993), promises to help design modelling and the reuse of design knowledge. An introduction of fractal based thinking is presented in the next section.

2.4 Fractal based thinking

The "fractal factory" idea was introduced by Warnecke (Warnecke, 1993). A fractal is "an independently acting corporate entity whose goal and performance can be precisely described", and it has following characteristics:

- Self-similarity

  Fractals are self-similar; each one performs services.

- Self-organisation

  Operatively, procedures are optimally organised by applying suitable methods. Tactically, fractals determine and formulate their goals in a dynamic process and decide upon internal and external contacts. Fractals restructure, regenerate and dissolve themselves.

- Self-optimisation

  The system of goals, which arises from the goals of the individual fractals, is free from contradictions and must serve the objective of achieving corporate goals.

- Goal-orientaıion

  Fractals are networked via an efficient information and communication system. They themselves determine the nature and extent of their access to data.

- Dynamics and vitality

  The performance of a fractal is subject to constant assessment and evaluation.

In recent years, the fractal factory idea has been applied in manufacturing systems. A

Fractal Manufacturing System (FrMS) (Ryu & Jung, 2003; Ryu et al., 2003) is based on the concept of autonomous cooperating agents referred to as fractals. A conceptual structure of FrMS is shown in Figure 2.7. The major component of an FrMS is a Basic Fractal Unit (BFU), which consists of five functional modules: observer, analyser, resolver, organizer and reporter. A fractal architecture is a hierarchical structure built on BFUs. The design of a basic unit incorporates a set of pertinent attributes that can fully represent any level in the hierarchy. Theses attributes serve to describe a specific structure and a functionality of a particular represented level, as well as its coordination with adjacent levels (Tirpak et al., 1992). In other words, the term "fractal" can represent an entire manufacturing factory at the top level or a machine at the bottom level. Each BFU provides services with an individual goal and acts independently. To function as a coherent whole, goal consistency is maintained by a goal-formation process. BFUs resolve conflicts through cooperation and negotiation.

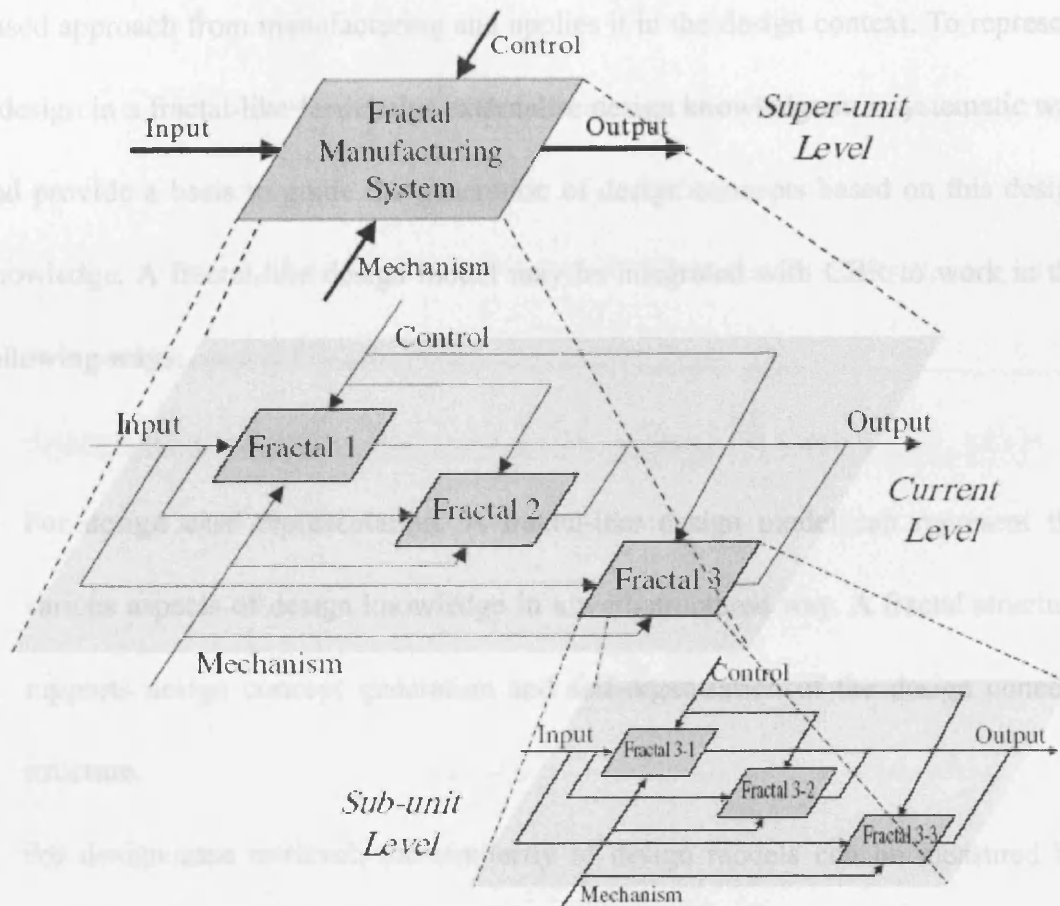


Figure 2.7: Conceptual structure of FrMS (Ryu & Jung, 2003)

Engineering design is also fractal in nature because a design problem can be considered as a consequence of "sub-designs" acting towards common general goals, with each sub-design having its own context and knowledge. A design therefore has the potential to be represented in a fractal-like form. This research takes the fractal based approach from manufacturing and applies it in the design context. To represent a design in a fractal-like form helps externalise design knowledge in a systematic way and provide a basis to guide the generation of design concepts based on this design knowledge. A fractal-like design model may be integrated with CBR to work in the following ways.

- For design case representation. A fractal-like design model can represent the various aspects of design knowledge in a well-structured way. A fractal structure supports design concept generation and self-organisation of the design concept structure.
- For design case retrieval, the similarity of design models can be measured by incorporating design knowledge based on the fractal-like design model.
- For design case adaptation, a fractal-based adaptation approach can use design knowledge such as performance and goals to guide the process of adaptation. The fractal specific characteristics such as self-similarity, self-organisation, goal-orientation can also play an important role in the automation of the adaptation process.

As noted before, successful conceptual design has a powerful impact on product quality. It is this need which is a primary motivation for the research presented in this thesis. The details of the proposed fractal-based approach will be explained in the subsequent chapters. The approach involves three elements:

- **Design case representation**. This research proposes a new representation technique, Fractal-like Design Modelling (FDM), which integrates design knowledge in a graph-based form, and has fractal-specific characteristics. This will be discussed in Chapter 3.
- **Design case retrieval**. Based on FDM, a novel method of the similarity assessment between a new design and the existing designs is developed. This will be explained in Chapter 4.
- **Design case adaptation**. With the help of fractal characteristics, a new approach to adaptive design is developed, which is called fractal-based adaptation. This will be addressed in Chapter 5.

These three parts work together to achieve an automated, case-based, conceptual design method: Fractal-Based Re-design.

2.5 Summary

In this chapter the research relevant to the work presented in this thesis has been

reviewed. The background of conceptual engineering design was reviewed from four perspectives. Firstly, the literature on engineering design research was reviewed. Two widely accepted models of engineering design process were considered. Secondly, conceptual design out of the engineering design process was highlighted. Thirdly, intelligent design was reviewed from three perspectives: design modelling, concept generation, and concept selection. Then some important AI techniques applied in design were discussed. This chapter also reviews the literature on the application of CBR techniques to engineering design. Finally, this chapter gives an introduction to fractal theory and its relevance to this research.

Chapter 3

Fractal-like design modelling using attributed graphs

3.1 Preliminaries

This chapter presents a design modelling technique developed to represent design knowledge. A new representation technique, Fractal-like Design Modelling (FDM) is proposed, which integrates design knowledge in a graph-based form, and has fractal specific characteristics. FDM is then used as the basis for assessing the similarity between a new design and existing designs, and for adapting a retrieved design to suit a new situation, which is presented later in this thesis.

The rest of the chapter is organised as follows: Section 3.2 explains fractal-like design modelling in detail; Section 3.3 presents the characteristics of a fractal-like design model which can help the process of design.

3.2 Fractal-like design modelling

Design modelling is a basis for case indexing, case retrieval and case adaptation.

Modelling the different aspects of a product is useful to support conceptual design. As mentioned in Chapter 2, the common representations of a design case, e.g. feature-based representation, graph-based representation and geometric representation, have their own focuses. In order to build a comprehensive design model, it is necessary to integrate various aspects of design knowledge.

Usually, design knowledge exists in a variety of forms. It can be related to design objects or to a design process. The knowledge is generally used for manipulating design objects, showing the next stage in given situations, producing and interpreting design specifications and controlling the design process (Preston & Mehandjiev, 2004). In order to effectively support the process of conceptual design, different types of design knowledge need to be defined and modelled in a formal way. This section discusses how to model design knowledge. Section 3.2.1 describes how to represent a design case. Representing the knowledge related to design objects is discussed in Section 3.2.2, and representing the knowledge related to the design process is discussed in Section 3.2.3.

3.2.1 Representing a design case in attributed graphs

Usually, a representation of an engineering design is a description of an engineering system expressed in terms of attributes (Arciszewski et al., 1995). Likewise, the proposed fractal-like design model applies attributed graphs as a carrier to represent

design cases. Design knowledge is represented by nodes and relations of the attributed graphs. The function, feature and structure models consist of primitive units comprising Elements (E), Attributes (A) and Relationships (R), which make up a graph. This is denoted as G = [E, A, R].

Element

An element is a basic unit. It is represented by a node of a graph. An element can be either an abstract entity or a physical entity. For example, in a function model, an element is a function, which is an abstract entity; whereas in a structure model, an element is a physical part, which is a physical entity. An element is identified by its label (name).

Attribute

Most elements have attributes, e.g., colour, shape, and material. Each attribute has a value. The value may be numerical (e.g., the width of a chair), or nominal (e.g., the material from which a chair is made can be wood or plastic).

Relationship

In addition to attributes, relations of elements are also important to represent a design. Relationships between elements are represented by edges in a graph and are described with relation variables included in the data structure. If the elements are abstract entities, the relationships will be abstract. For example, a seat's function has two abstract entities: to support the legs and to support the back. The relationship between these two entities "and" is an abstract relationship. If the elements are real objects, the relationships can be abstract or they can be positional relationships involving numerical or nominal data. Some examples of the configuration of mechanical components are shown as follows:

- A chair *is composed of* a seat, a back, and four legs (abstract relationship);
- The legs *are parallel to* each other (nominal relationship);
- The back and the seat *lie at an angle of 60 degrees* (numerical relationship).

Using these three primitive units, the models of a design case can next be explained.

3.2.2 Representing knowledge related to design objects

Generally, design knowledge is represented categorically. Coyne et al. (Coyne et al., 1990) characterised design as concerned with design descriptions, a vocabulary of elements, interpretations, and design knowledge in their symbolic model of a design

system. Maher et al. (Maher et al., 1995) considered the content of a design case to be design drawings, requirements and solutions, or function-behaviour-structure. Bilgic and Fox (Bilgic & Fox, 1996) defined three crucial elements which must be explicitly represented in a case base, namely concepts (fit, form, function, behaviour, working principle), issues, and requirements. It is believed that design needs a multiplicity of representations (Dym, 1994). This research, therefore, proposes to represent different aspects of design knowledge by three models: a "*Function*" *model*, a "*Feature*" *model* and a "*Structure*" *model*, which originate from the well-known FBS model.

Function model

The function model refers to the purpose of the design and explicitly describes the way in which the design is to be used. For instance, the purpose of a bolt is to connect and tighten the components. In a function model, an element is a function and a relationship is the abstract interrelation between two functions. A function model can be denoted as $G_{function} = [E, R]$.

An example of a function model of a car body is illustrated in Figure 3.1(a). In the figure, the function model of a car body is composed of three elements (support chassis, protect passengers, and stylise) and three relationships (and, and, and). The representation of the function model in case base is shown in Figure 3.1(b).

Feature model

A feature model describes the characteristics of a design. It is an extended model compared to "behaviour" in an FBS model, which describes the effect of a design after it is used. A feature model not only includes the behaviour of a design, but also the important functional and structural characteristics. A feature model represents the design in terms of features. Each element corresponds to a feature, which has an individual attribute, a set of attributes or derived attributes taken from elements of other models. A feature model can be denoted as $G_{feature} = [E, A]$.

An example of a feature model of a car body is shown in Figure 3.2(a). The feature model of a car body is composed of three elements (behaviour features, function features, structure features), and each element has its own attributes. The representation of the feature model in the case base is shown in Figure 3.2(b).

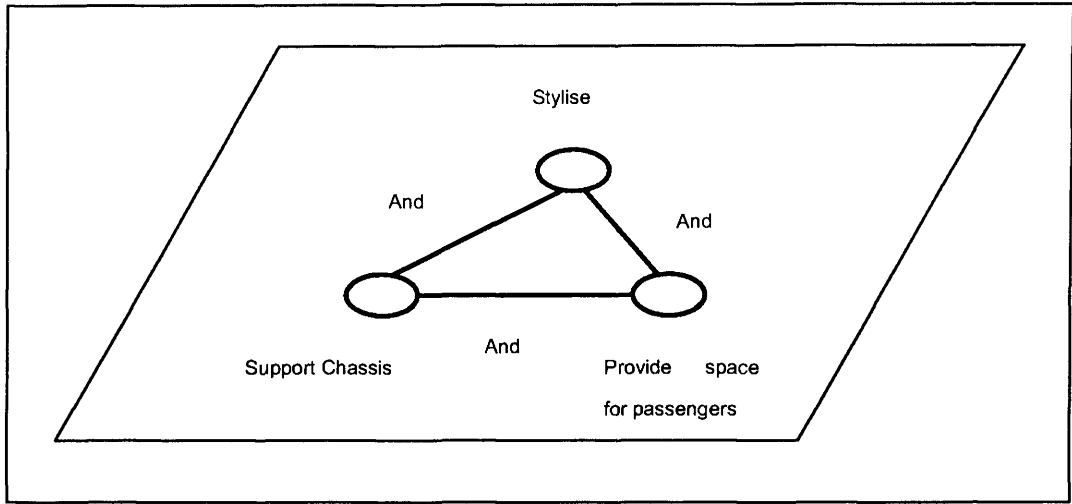
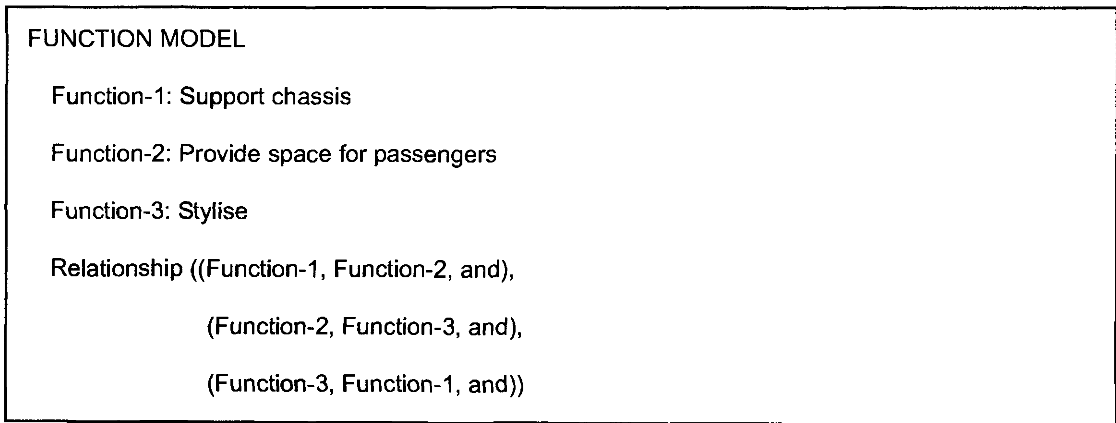


Figure 3.1(a): A function model of a car body



FUNCTION MODEL

Function-1: Support chassis

Function-2: Provide space for passengers

Function-3: Stylise

Relationship ((Function-1, Function-2, and),
(Function-2, Function-3, and),
(Function-3, Function-1, and))

Figure 3.1(b): Representation of a function model of a car body in case base

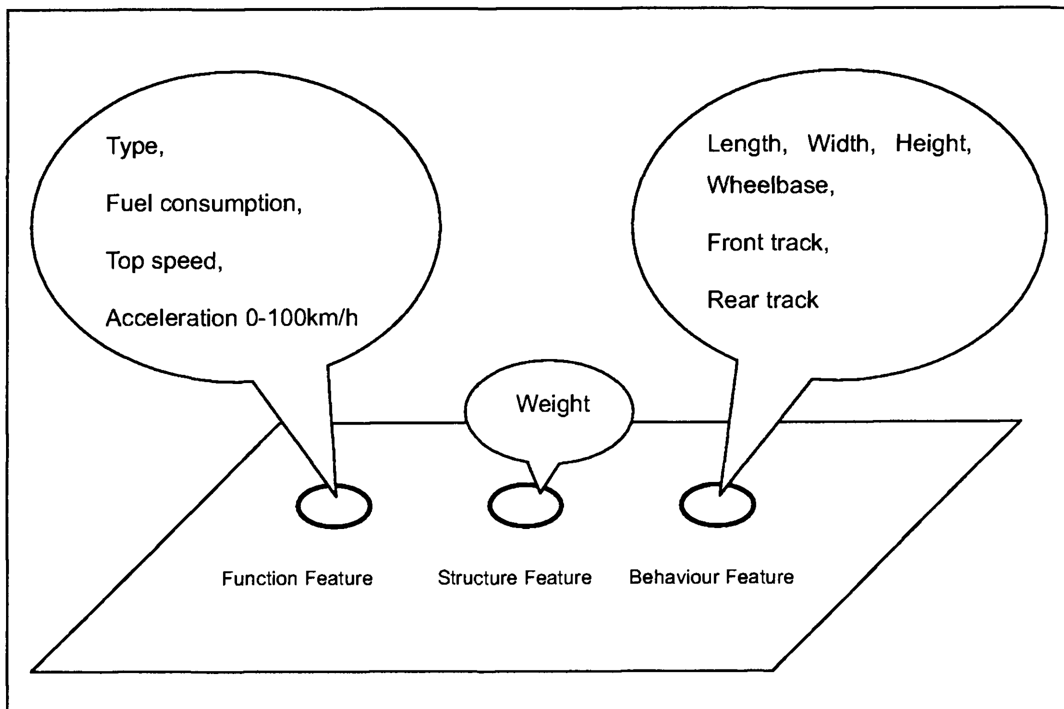


Figure 3.2(a): A feature model of a car body

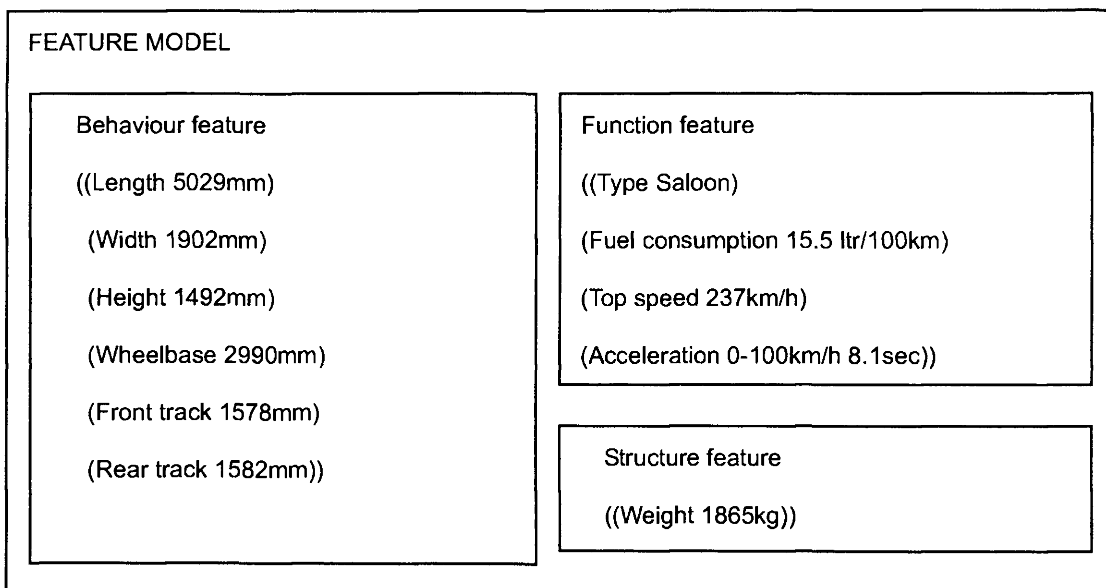


Figure 3.2(b): Representation of a feature model of a car body in case base

Structure model

The structure model concerns the physical organisation of a design. It exists at three levels: *assembly model, part model*, and *geometric model.* An assembly model consists of elements representing assemblies and relationships signifying the connections between the assemblies. An assembly contains attributes describing its characteristics and relationships between elements, including nominal and numerical relationships. A part model is made up of parts, which are the basic units of a design model. A complex design can be considered to be composed of sets of parts. A part is represented by a node in a graph. A node contains all the attributes of a part and a relationship can be either nominal or numerical. A geometric model provides a direct way to illustrate a design. Each node represents a geometric entity. A geometric entity can be any geometric model, e.g. line, curve, and box. There are positional relationships between the nodes, which can be nominal or numerical. Among these three levels, there are relationships between the assembly model and part model, as well as between the part model and geometric model. These relationships are subordinate relationships and are abstract. There are two types of relationships, therefore, in a structure model: positional relationships (numerical and nominal) and subordinate relationships (abstract). A structure model can be denoted as $G_{Structure} = [E, A, R]$.

Figure 3.3(a) illustrates an example of a structure model of a car body. At the first

level, the assembly model comprises three elements: engine compartment, passenger cabin, and luggage compartment. Every element has attributes such as length, width, and height. The relationships between elements is positional. Similarly, at the other levels, the models contain attributed elements with positional relationships among them. There are subordinate relationships between the levels, such as the one that denotes that the engine compartment is composed of bonnet, front panel, left front wing, and right front wing. For simplicity, only the representation of the assembly model in the case base is shown in Figure 3.3(b).

The use of graph-oriented representation to describe a design is intended to enable the integration of different kinds of knowledge expressed in different forms. The notion of integration here is, therefore, more in terms of types of knowledge and concepts than just of representation.

3.2.3 Representing design knowledge related to the design process in a fractal model

Design knowledge related to the design process, e.g. adaptation knowledge, is represented through generalised schemes and stored within individual design cases in this research. This research proposes to use fractals to describe these schemes.

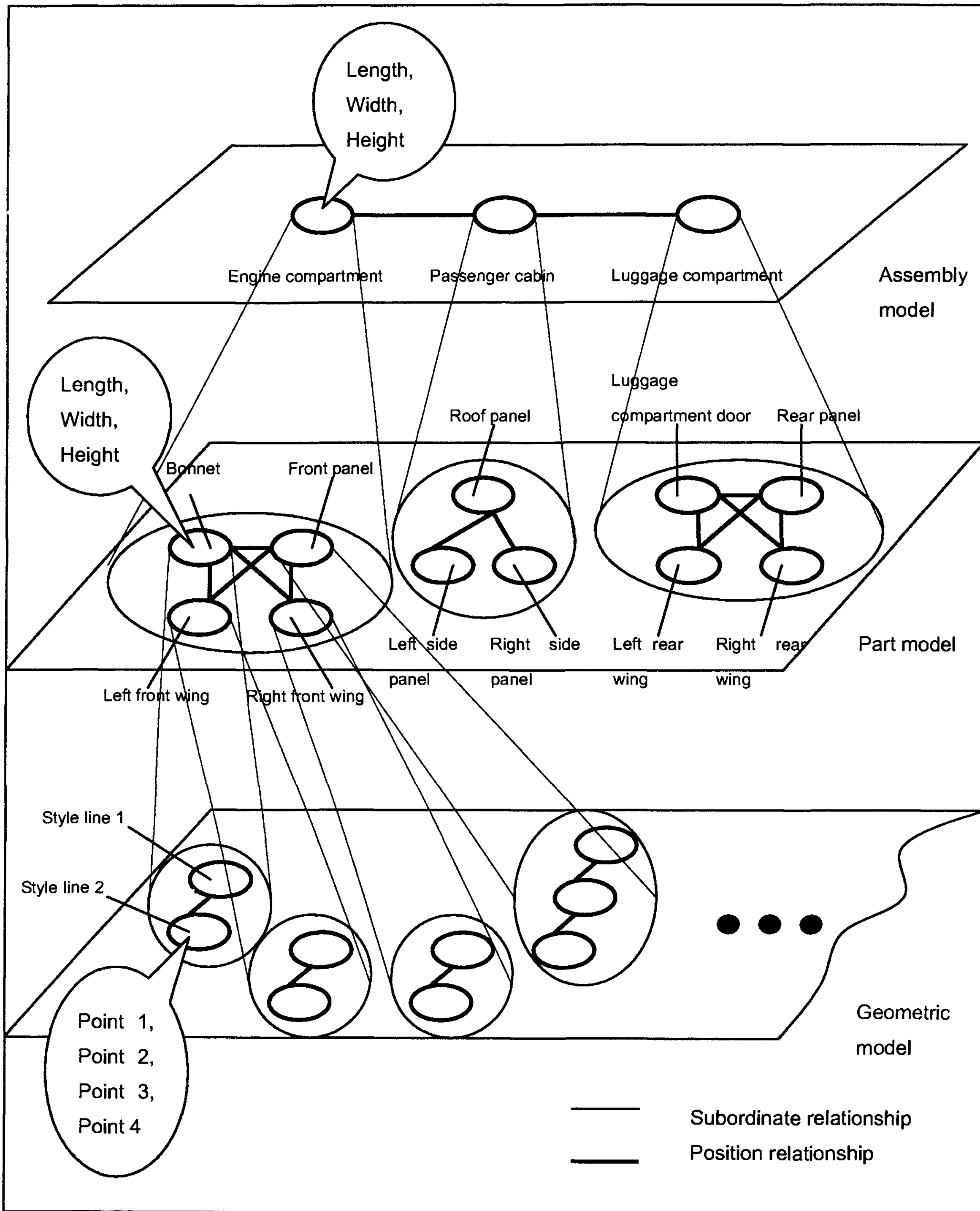


Figure 3.3(a): A structure model of a car body

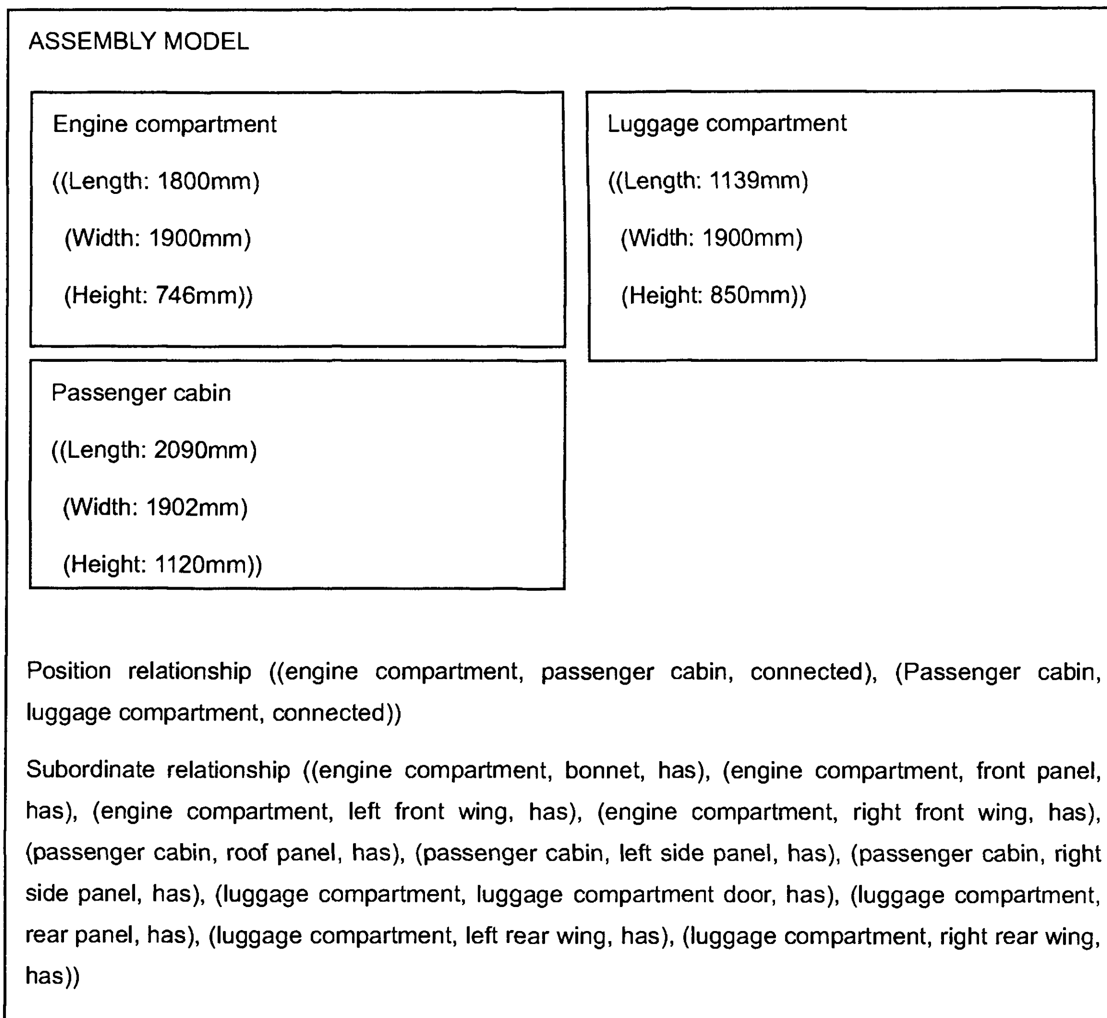


Figure 3.3(b): Representation of an assembly model of a car body in case base

In a structure model, a part (or a set of parts) that has particular goals and can provide a certain performance to a design is referred to as a ***fractal***. Basic fractal units form a fractal model. A fractal model is an extended representation of a part model and an assembly model. In a basic fractal unit, as well as geometric attributes, design knowledge is also incorporated to guide the process of adaptation. This research proposes to represent design knowledge by *Performance* and *Goal*.

Performance describes the task of a fractal. For example, in car body design the performance of an engine compartment is "to provide space for the engine".

Goal specifies the objective(s) of a fractal. Each fractal has individual goals which cooperate with those of other fractals. Knowledge of how fractals achieve the ultimate goals is integrated into fractals. In this research, it is proposed that a goal consists of four parts:

- An attribute which has effect on the goal.
- An operation on the attribute, which can be "maximise" or "minimise".
- Type of goal, which can be individual goal (:i) (the goals that cannot propagate to another level) or corporate goal (:c) (the goals that are able to propagate to another level).
- The corporate goal to which this goal contributes.

A goal can be described by a 4-tuple. For example, :G1-G1 (:minimise :volume :c :G1)

indicates that the goal :G1-G1 is to minimise the volume of the fractal. It is a corporate goal and its parent goal is :G1.

An example of a basic fractal unit is shown in Figure 3.4, and an example of representing an engine compartment as a fractal is shown in Figure 3.5.

A design can be decomposed into a hierarchy of fractals, with each fractal providing a specific problem description and a corresponding solution. Each fractal performs its own role in a structure, while all the fractals work together to produce the desired functions of a design. The term "fractal" can represent an entire fractal model at the assembly level or a basic fractal unit at the part level. Such a hierarchical structure allows the use of both specific design knowledge in each fractal and overall design knowledge in an entire fractal model. Figure 3.6 shows a fractal structure. The structure can be reconfigured according to a change in customer requirements. In addition, under this architecture, a series of knowledge architecture can be generated. Each category of components in the fractals can form their own system. For example, goals of fractals constitute a structure that describes the context and relationship of the goals. This information will be used for design adaptation. The detail will be discussed in Chapter 5.

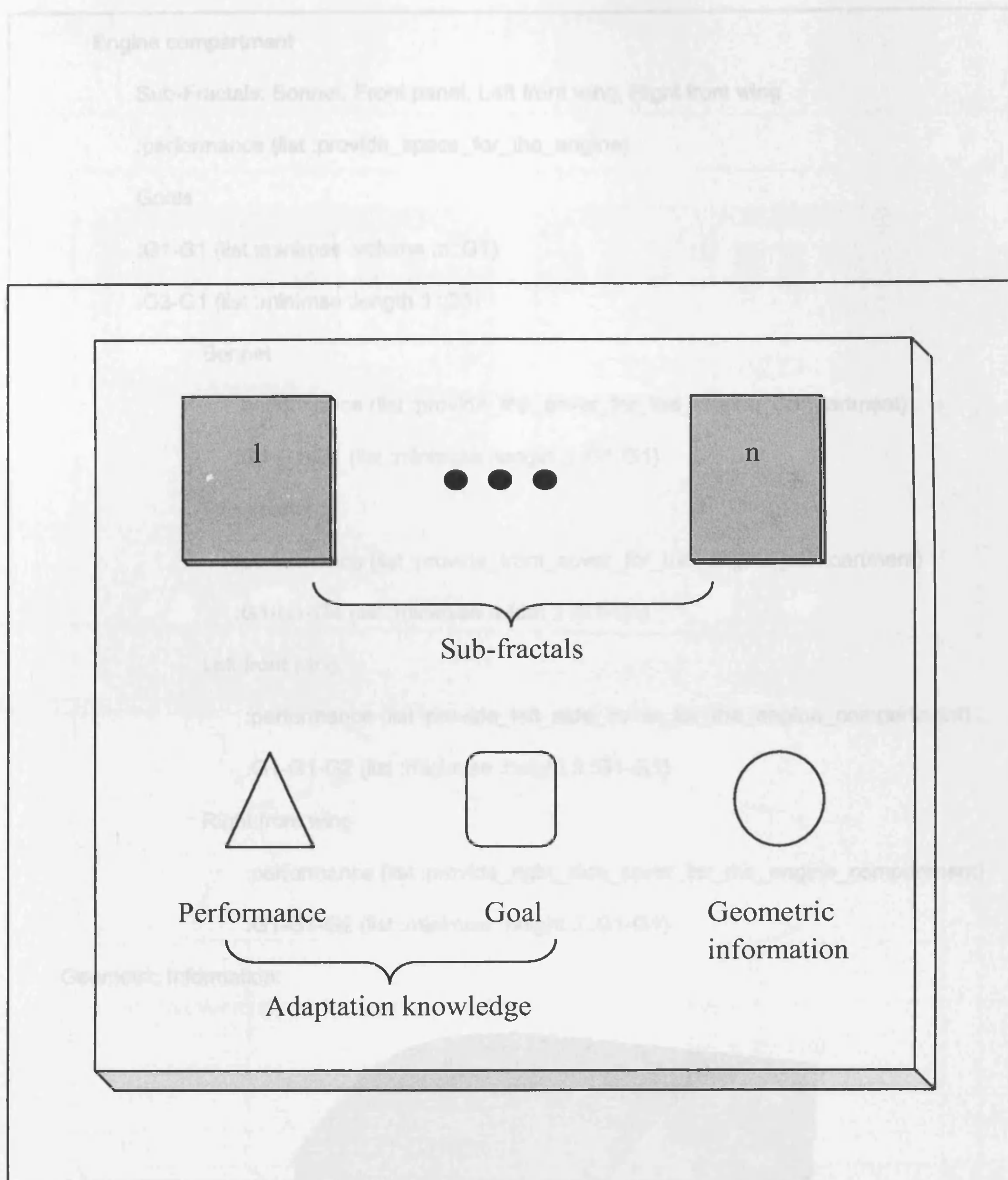


Figure 3.4: An illustration of a basic fractal unit

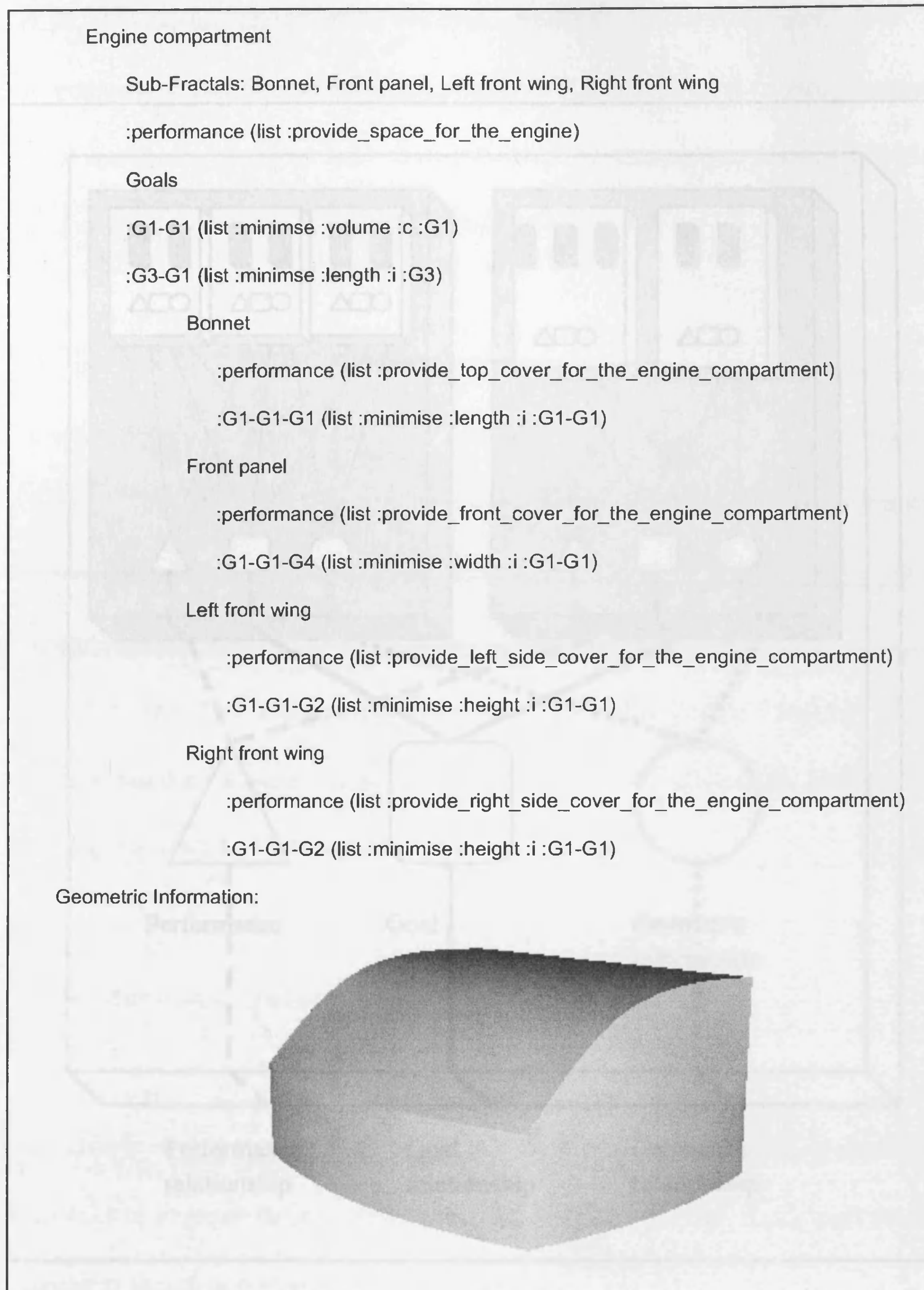


Figure 3.5: An example of representing an engine compartment as a fractal

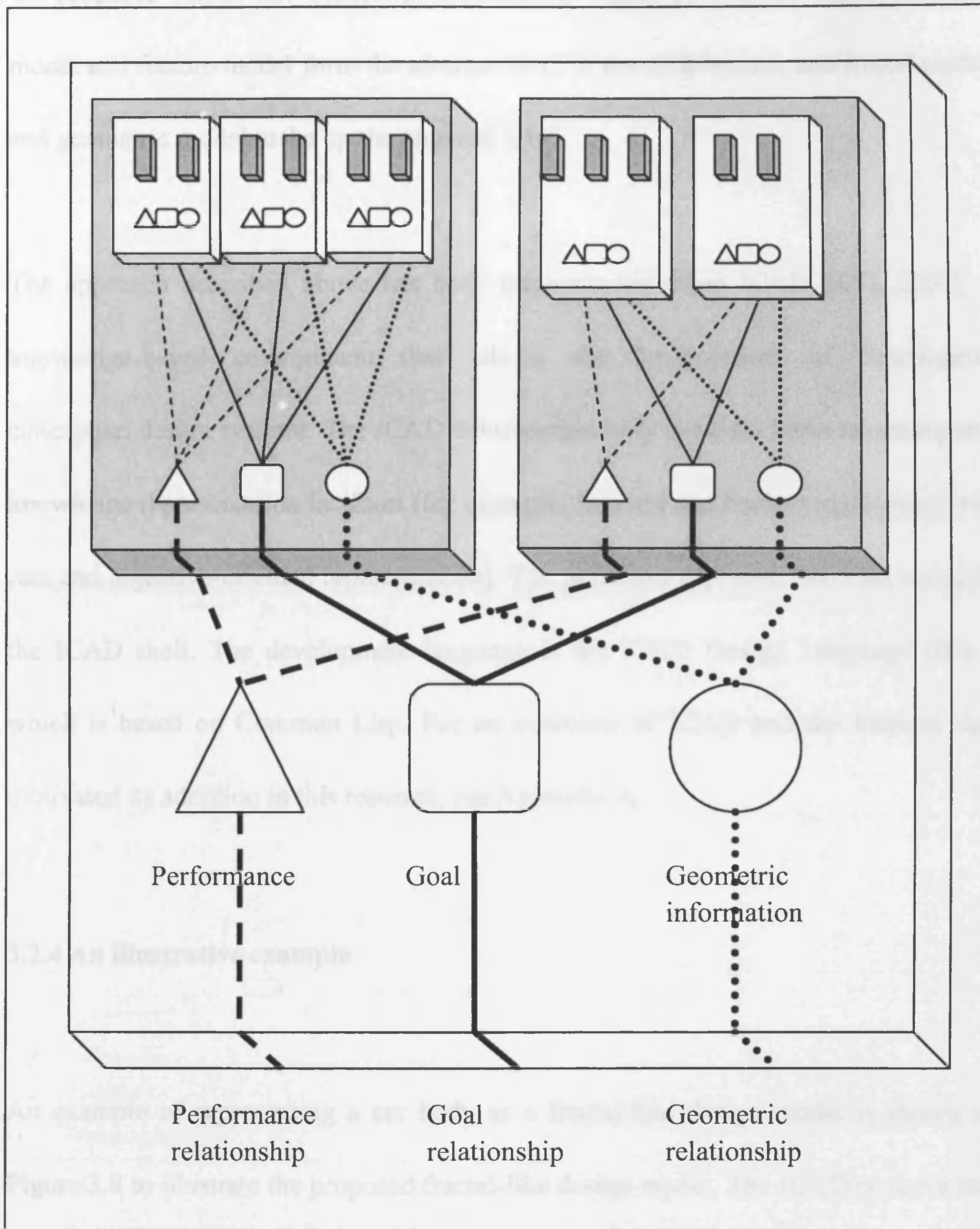


Figure 3.6: An overview of the fractal structure

So far, a fractal-like design model has been introduced. The complete architecture of the proposed fractal-like model is illustrated in Figure 3.7. In the figure, function model and feature model form the abstract level of the architecture, and fractal model and geometric model make up the physical level.

The approach described above has been implemented using ICAD (KTI, 2001), a knowledge-based environment that allows the development of "intelligent" conceptual design systems. The ICAD environment only provides basic reasoning and knowledge representation facilities (for example, forward and backward chaining, and rule and objective-oriented representation). The proposed approach has been coded in the ICAD shell. The development language is the ICAD Design Language (IDL), which is based on Common Lisp. For an overview of ICAD and the features that motivated its adoption in this research, see Appendix A.

3.2.4 An illustrative example

An example of representing a car body as a fractal-like design model is shown in Figure 3.8 to illustrate the proposed fractal-like design model. The ICAD code for the example is shown in Appendix B.

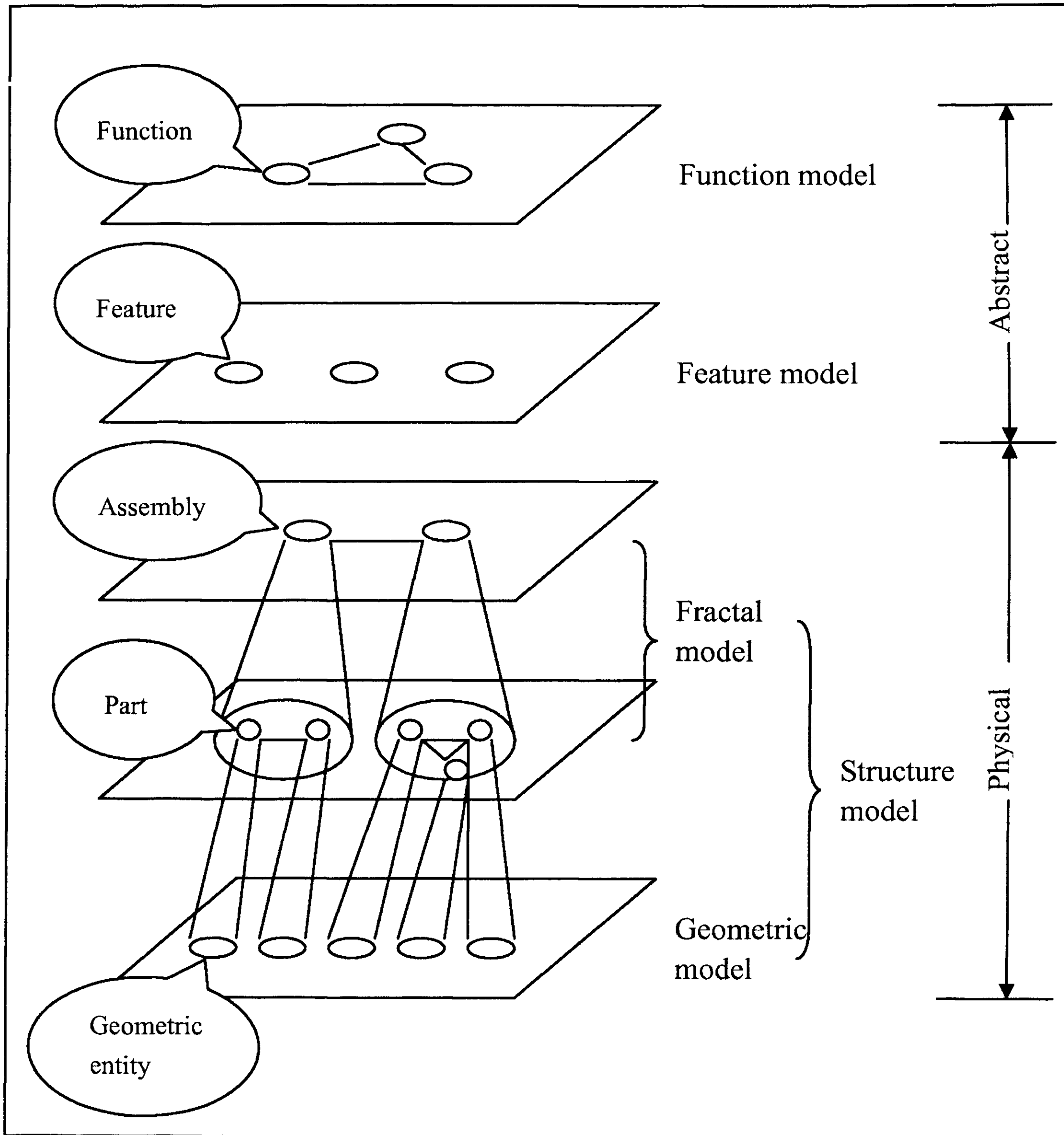
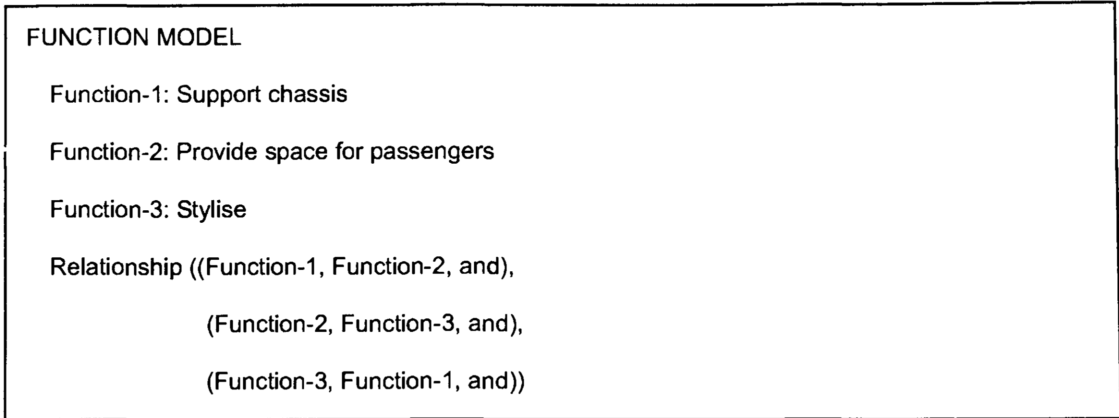


Figure 3.7: A structure of a fractal-like design model



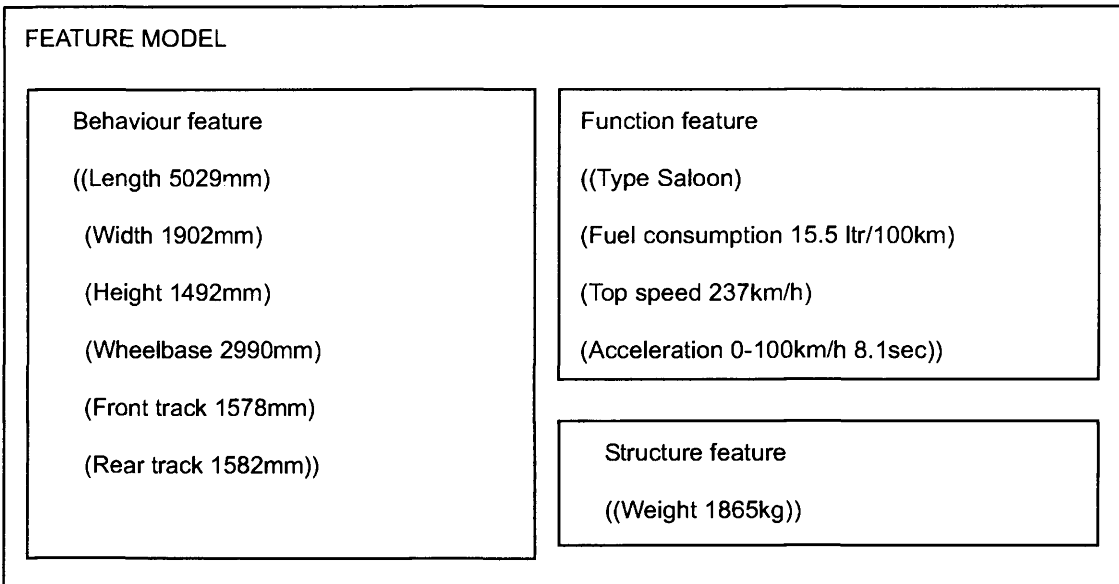


Figure 3.8: An example of representing a car body as a fractal-like design model (to be continued)

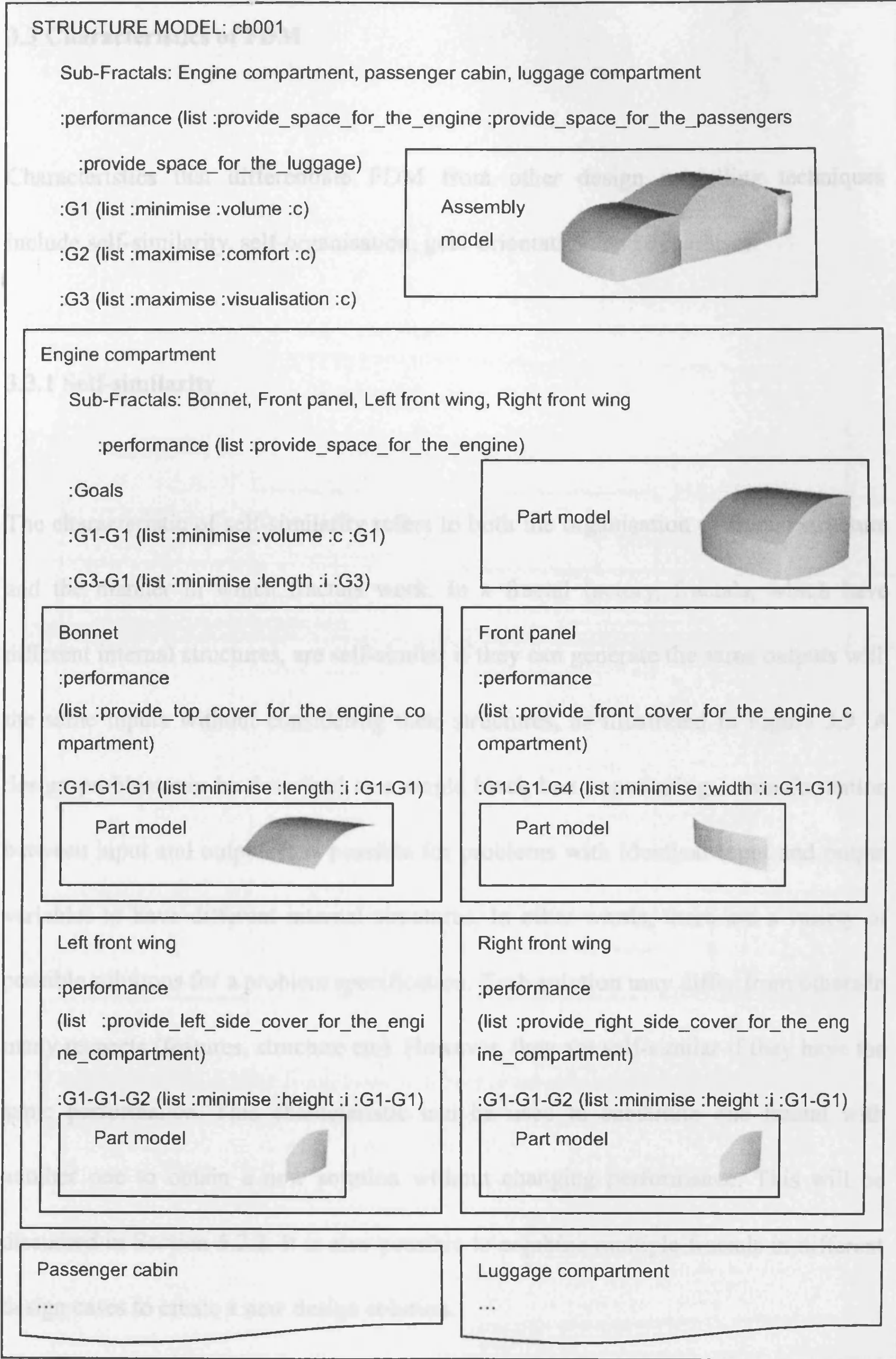


Figure 3.8: An example of representing a car body as a fractal-like design model (continued)

3.3 Characteristics of FDM

Characteristics that differentiate FDM from other design modelling techniques include self-similarity, self-organisation, goal-orientation, and dynamism.

3.3.1 Self-similarity

The characteristic of self-similarity refers to both the organisation of fractal structure and the manner in which fractals work. In a fractal factory, fractals, which have different internal structures, are self-similar if they can generate the same outputs with the same inputs without considering their structures, as illustrated in Figure 3.9. A design problem can be described as a single black box constituting a transformation between input and output. It is possible for problems with identical input and output variables to have different internal structures. In other words, there are a variety of possible solutions for a problem specification. Each solution may differ from others in many respects (features, structure etc). However, they are self-similar if they have the same performance. This characteristic can be used to substitute one fractal with another one to obtain a new solution without changing performance. This will be discussed in Section 5.2.2. It is also possible to combine multiple fractals in different design cases to create a new design solution.

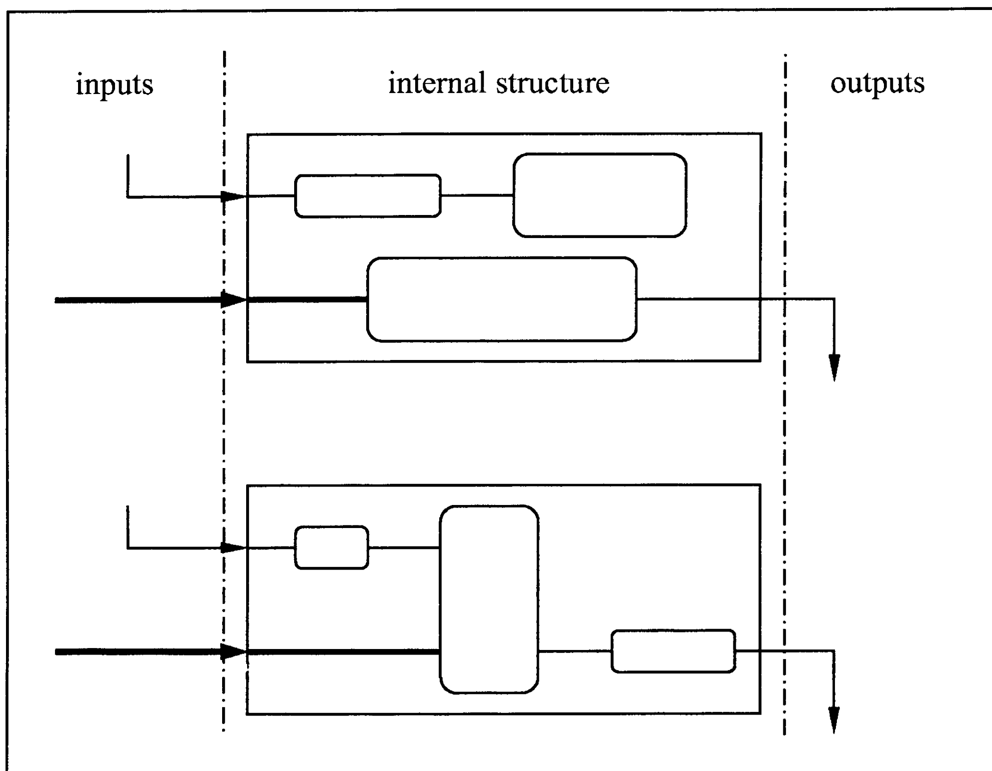


Figure 3.9: Self-similar fractals with different internal structures (adapted from Warnecke 1993)

3.3.2 Self-organisation

When confronted with new customer requirements, fractals restructure themselves to suit the new situation. The relationships between fractals are reconfigured and the fractals are reorganised. This reorganisation of fractals is normally supported by an appropriate design modelling language. The structure of the whole design can be changed depending on the goals of the fractals and the customer requirements. This will be discussed in Section 5.2.2.

3.3.3 Goal-orientation

Each fractal has individual goals. These are networked together from the bottom level to the top level to contribute towards corporate goals. The goals of the fractals form a structure, a Goal Dependency Graph (GDG), which describes the context of the goals and the relationships between them. A GDG starts from corporate goals and ends at individual goals. A GDG propagates the corporate goals to individual goals, thus facilitating the propagation of goals from the top-level fractals to the bottom-level fractals. If there are contradicting or duplicated goals in the GDG, a simplified GDG can be built by removing them, and adaptation can be executed according to that. These will be further discussed in Section 5.2.2.

3.3.4 Dynamism

Dynamism, or vitality (Warnecke, 1993), which originated from the field of biology or medicine, denotes the "power to sustain life". In this research, the term dynamism is used to describe the adaptability of a fractal in the process of case-based design. During its lifetime, a fractal serves a design by modifying its performance and goals and cooperating with other fractals. The dynamism of a fractal is determined by the "enduring" and "non-enduring" attributes inside it. The idea of "enduring" and "non-enduring" attributes was introduced by Gourashi (Gourashi, 2003). An attribute is enduring if it remains unchanged during the life of a design, and if the very concept of the design will change should the attribute be modified. A non-enduring attribute, on the other hand, is an attribute that, if changed, does not alter the design concept, but only creates another instance of it. The more non-enduring attributes a fractal has, the more dynamism it gains.

3.4 Discussion

This chapter has proposed a design modelling method named FDM for case-based reasoning in conceptual design, which is the first effort to apply fractal theory to design modelling. In this method, a design is represented by function model and feature model at the abstract level, and by fractal model (including assembly model and part model) and geometric model at the physical level. All these representations

are in the form of attributed graphs. A fractal model is designed for representing the knowledge related to the design process, while the other models are designed for representing the knowledge related to the design objects. Knowledge related to the design objects includes product function, feature and structure, while knowledge related to the design process includes performance and goal. A summary of a fractal-like design model is shown in Figure 3.10.

Moreover, a fractal-like design model possesses important fractal characteristics, which can greatly benefit the process of case-based design in the following aspects:

- Self-similarity enables the substitution of fractals and the combination of multiple fractals in different design cases.
- Self-organisation facilitates the reorganisation of fractals.
- Goal-orientation helps guide the process of case adaptation.
- Dynamism helps realise the adaptability of a fractal.

In addition, FDM has several advantages:

- It is sufficiently comprehensive to represent engineering design problems in different fields.
- It helps the recognition of designs at both an abstract level and a practical level.
- FDM enables the integration of different kinds of knowledge expressed in different forms.

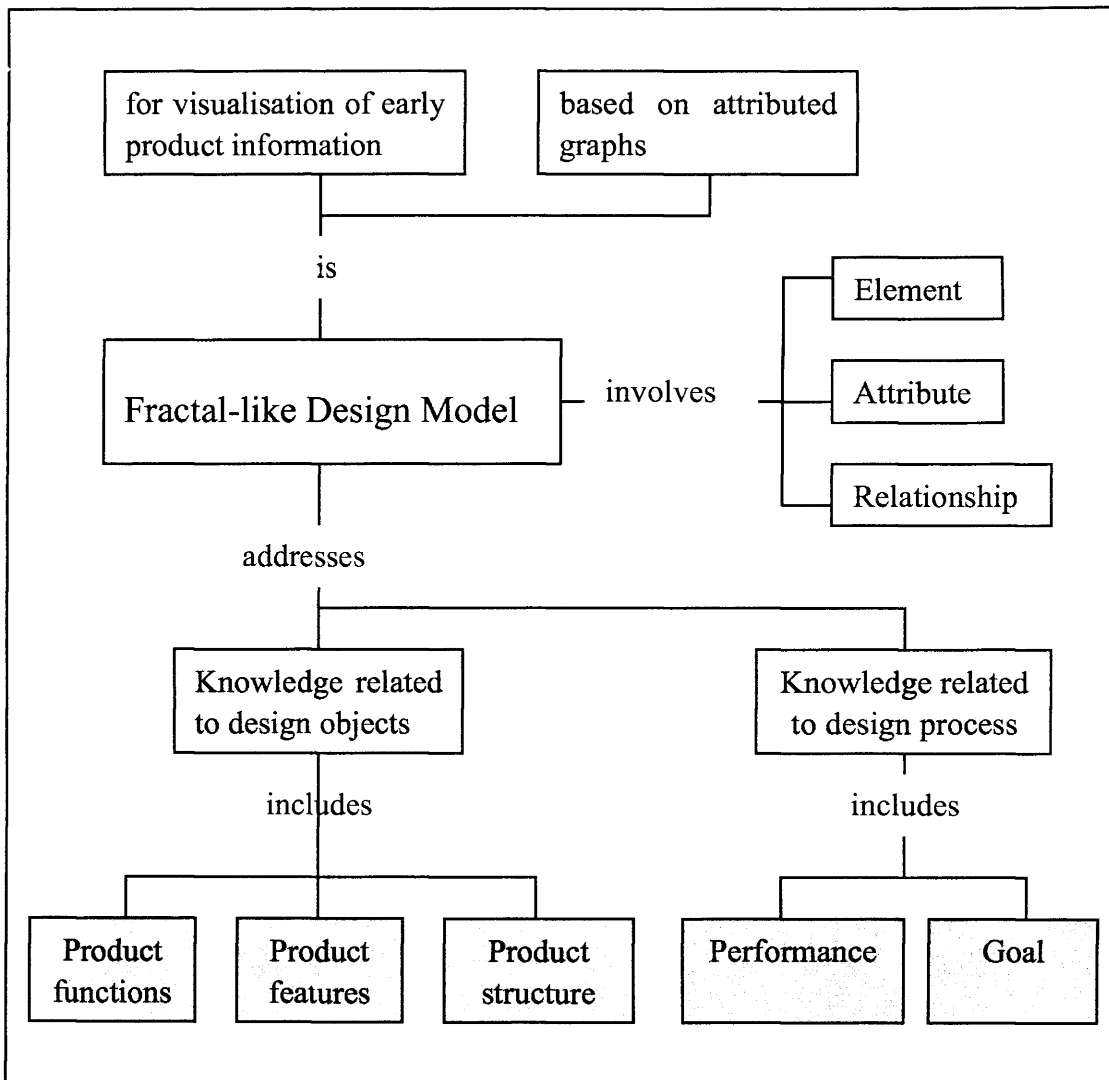


Figure 3.10: A summary of fractal-like design model

3.5 Summary

The research reported in this chapter is aimed at developing a systematic approach for intelligent case-based design. A design model is at the core of the system. The model will be employed to assess the similarity between a new design and the existing designs, and to adapt a retrieved design to suit a new situation. The model is in the form of attributed graphs containing knowledge about function, feature, structure, performance, and goal, and has the fractal characteristics of self-similarity, self-organisation, goal-orientation, and dynamism.

The chapter studied the approach of fractal-like design modelling and discovered the key properties of fractals that make it work.

Chapter 4

Similarity Assessment on Attributed Graphs in Design Case Retrieval

4.1 Preliminaries

In many CBR systems, case retrieval relies on the similarity between the new problem context and cases in the case base. Methods of grading similarity have been developed. However, some methods do not incorporate sufficient product information to allow detailed comparisons of similarity among complex designs, while some involve complicated geometric comparisons that do not measure similarity according to criteria for domain specific technical knowledge (Elinson & Nau, 1997).

This chapter introduces an approach for assessing the similarity of design models presented in Chapter 3. Methods of similarity assessment have been reported in the literature (Smyth & Keane, 1998). As mentioned in Chapter 2, the existing methods compare either features or structures separately. In order to have a more comprehensive similarity measure, this research considers both structure and feature factor. The aim is to develop a method to query a case base of design models, which are in the form of attributed graphs containing design knowledge about function,

feature and structure and to identify existing designs with graphs similar to the target problem. Similarity of design models is measured by concurrently applying feature-based similarity measures and structure-based similarity measures.

The rest of the chapter is organised as follows: section 4.2 introduces the generation of the compared model; section 4.3 describes the proposed approach to measure the similarity of designs; section 4.4 gives an example to explain the method.

4.2 Compared model generation

The first step towards case retrieval is to build a compared model. A compared model is used to query a case base of design models. Some researchers have constructed a Model Dependency Graph as a query, which is a representation of the design features and interdependencies of those design features of a CAD model (Cicirello & Regli, 2001). The nodes of this graph correspond to individual design features and an edge between two nodes corresponds to some spatial dependence between the features. As discussed in Chapter 3, cases in the case base are represented in three models: function model, feature model, and structure model. Accordingly, a compared model which is in the form of attributed graphs containing design knowledge about function, feature, and structure, needs to be specified to identify existing designs with graphs similar to the target problem. The user should be able to select these models by preference.

To begin case retrieval, a user is asked to give a description of function, feature, and structure to query the case base. For structures, structure models in the case base can be selected as a query. Constructing a compared model that integrates new requirements and existing models is the usual method for re-design.

An example of a compared model is shown in Figure 4.1. Used as the query are: functions and their relationships, features (including length, width, height, car type, wheelbase, front track, rear track, weight, fuel consumption, and acceleration), and input structure (which is "CAB" in the case-base).

4.3 Similarity measure

It is proposed that design similarity is evaluated using a combination of two methods: *Structure-based similarity evaluation* and *Feature-based similarity evaluation*. The structure-based similarity measure is based on the minimum number of the primitive operations on the structure to equalise the target model and the case. It makes a premise for easy adaptation, as most retrieval approaches expect that a case, which is similar to the target problem, should also be easy to adapt. The feature-based similarity measure is a traditional method. It measures similarity by counting the number of common features between the case and the target problem (often with some priority weighting of features).

Function

(Function-1: Support chassis)

(Function-2: Provide space for passengers)

(Function-3: Stylise)

Function Relationship ((Function-1, Function-2, and), (Function-2, Function-3, and), (Function-3, Function-1, and))

Input Features

((Length 5029mm)

(Width 1902mm)

(Height 1492mm)

(Wheelbase 2990mm)

(Front track 1578mm)

(Rear track 1582mm)

(Type Saloon)

(Fuel consumption 15.5 ltr/100km)

(Top speed 237km/h)

(Acceleration 0-100km/h 8.1sec)

(Weight 1865kg))

Input structure

(THE :CASE-BASE :CAB)

Figure 4.1: A compared model

4.3.1 Structure-based similarity measure

To assess similarity of structure, the equivalence of two graphs needs to be defined first. Assume there are two graphs $G1$ and $G2$. $E_i(G)= [e_1(G), e_2(G), \dots e_n(G)]$ is used to denote the elements at the ith level, in which $e_n(G)$ is the nth element of graph G in level i. $R_i(G)= [r_1(G), r_2(G), \dots r_n(G)]$ denotes the relationships between the elements in level i and $P_i(G)= [p_1(G), p_2(G), \dots p_n(G)]$ denotes the relationships between level i and level $i\text{-}1$. There is equivalence between graphs $G1$ and $G2$ at the ith level, i.e. $Q_i(G1, G2)$, as a Boolean variable, is true if and only if:

a. the numbers of elements at the ith level in $G1$ and $G2$ are equal;

b. the numbers of relationships at the ith level in $G1$ and $G2$ are equal;

c. for each $e_j(G1)$, there is a corresponding $e_k(G2)$ such that $e_j(G1)= e_k(G2)$;

d. for each $r_j(G1)$, there is a corresponding $r_k(G2)$ such that $r_j(G1)= r_k(G2)$;

e. for each $p_j(G1)$, there is a corresponding $p_k(G2)$ such that $p_j(G1)= p_k(G2)$.

$e_j(G1)= e_k(G2)$ if the labels of the elements are the same. Attributes are not considered in assessing structural equivalence, because only structure is of interest here; $r_j(G1)= r_k(G2)$, $p_j(G1)= p_k(G2)$ if the types of the relationships are the same. Let $T_i(G1, G2)$ be a Boolean variable and $T_i(G1, G2) = Q_1 \wedge \dots \wedge Qi$. $T_i(G1, G2)$ is true if $Q_1(G1, G2), Q_2(G1, G2), \dots, Q_i(G1, G2)$ are all true. Then two graphs $G1$ and $G2$ are equal if and only if for all i, $T_i(G1, G2)$ is true.

For ease of adaptation, a cost-based distance calculation method is applied to grade

similarity. The degree of similarity between *G1* and *G2* can be determined by assessing the minimum number of primitive operations (structural modifications) that need to be applied to *G1*, in order to make *G1* and *G2* equivalent. Primitive operations that can be performed on a graph are adding, removing, and replacing an element or a relationship. Adding/removing an element or a relationship is to append/delete an element or a relationship to/from a graph. Replacing an element or a relationship is the operation of removing an existing element or a relationship and adding a new one. The dissimilarity D_s is given by the sum of the number of primitive operations as shown in Eq. (1).

$$D_s = \Sigma n_1 + 2\Sigma n_2 \quad (1)$$

where n_1 is the number of added/removed elements and relationships; n_2 is the number of replaced elements and relationships.

In Eq. (1), n_2 is multiplied by 2 because the replacement operation involves two steps, removal and addition.

The structure-based similarity S_s is defined in Eq. (2), so that the lower the degree of dissimilarity is, the more similar the graphs are.

$$S_s = 1/\, D_s \quad (2)$$

This method of accessing similarity is applied to the function model and the structure model. Operating on the geometric level of the structure model is different from operating on the other levels. The first difference is that entity types instead of labels are compared. As geometric entities may have different names in different designs (for example, the same curve can be named curve001 and curve002 in different cases *d1* and *d2* respectively), elements cannot be identified by their labels. The second difference is that only subordinate relationships are compared. There are many positional relationships in a geometric model and they can be easily modified and generated. It is difficult and unnecessary to compare all the positional relationships in a complex design. The collection of subordinate relationships is defined in Figure 4.2. The similarity measure for the function model and structure model are defined in Figure 4.3 and Figure 4.4.

This method calculates the similarity of the graph structure, which reflects the difficulty of modifying the structure of a given design model into a target structure. Following the similarity measure, all the compared designs are arranged in an ascending order of similarity. Two lists are obtained, which respectively represent function similarity and structure similarity between a particular design and a given set of design cases.

```
Repeat
    Search the objects in the case base
    if the object is :sub-relations
        then write the contents of the object in a list A
    end-if
until the complete case base is searched.
```

Figure 4.2: Obtaining the subordinate relationships

```
Repeat
    Select a case in the case base
    Repeat
        Read an input function
        if the input function is in the function model of the case
            then count equal functions
        end-if
    until all the input functions are searched
    Repeat
        Read an input function relationship
        if the input function relationship is in the function model of the case
            then count equal function relationships
        end-if
    until all the input function relationships are searched
    Count the sum of equal functions and function relationships
    Calculate the dissimilarity
    Sort the cases in an ascending order of similarity
until the complete case is searched
```

Figure 4.3: Structure-based similarity measure on function model

```
Repeat
    Select a case in the case base
    Repeat
        Read an element of the input structure
        if the element is in the structure model of the case
            then count equal elements
        end-if
    until all the elements in input structure are searched
    Repeat
        Read a subordinate relationship in the input structure
        if the subordinate relationship is in the structure model of the case
            then count equal relationships
        end-if
    until all the subordinate relationships in the input structure are searched
    Count the sum of equal elements and relationships
    Calulate the dissimilarity
    Sort the cases in an ascending order of similarity
until the complete case is searched
```

Figure 4.4: Structure-based similarity measure on structure model

4.3.2 Feature-based similarity measure

This research proposes that features used for comparison are of two types, as defined in Figure 4.5. One is features from the feature model, which are usually attributes describing function, structure and behaviour. The other type is optional features, which are attributes or elements extracted from other models by the user.

The equivalence of attributes is defined as follows:

- For nominal attributes, they are equal if the strings are equal.
- For numerical attributes, they are equal if they are in the range of tolerance band. The tolerance bands are given by the user, as shown in Figure 4.6. Taking the first element in the list of the tolerance band for example, 0.1 indicates that the input value Vi and the compared value Vc are equal if $0.9Vc < Vi < 1.1Vc$.

In order to find design cases that are really needed by the designer, features are subjectively graded by the user, as shown in Figure 4.7. Let N_e denote the number of pairs of features that are equal to each other within a specified tolerance band and N_m the number of features in the presented case that do not exist in the given design. Feature-based similarity S_f can be calculated using the following equation:

```
:optional-features (list (list :passenger-cabin :length 3000)(list :passenger-cabin :width 1902) (list :passenger-cabin :height 1490) (list :luggage-compartment :length 529) )

:input-features
 (list (the :length) (the :width) (the :height) (the :wheelbase) (the :fronttrack) (the :reartrack)
    (the :car-type)(the :fuel-consumption) (the :top-speed) (the :acceleration)
    (the :weight))
```

Figure 4.5: Features for comparison

```
:tolerance-band (list 0.1 0.08 0.05 0.08 0.05 0.05 1 0.1 0.1 0.1 0.3)
```

Figure 4.6: Tolerance band

```
:weights (list 0.5 0.5 0.2 0.2 0.2 0.2 0.5 0.3 0.3 0.3 0.3)
```

Figure 4.7: Grades for features

$$S_f = \Sigma k_i N_e - \Sigma k_i N_m \quad (i=1,\ldots n, \quad 0 \leqslant k_i \leqslant 1) \qquad (3)$$

where

k_i –grades of features,

N_e - number of equal features,

N_m - number of missing features.

Feature-based similarity measures on a feature model and optional features are defined in Figure 4.8 and Figure 4.9 below. The sum-up of these two similarity measures are defined in Figure 4.10.

```
Repeat
        Read a case in the case base
        Repeat
            Read an input feature
            Find the according feature in the case base
            Read Tolerance-band
            Read Weight
            if the features equal in a tolerance-band
                then add Weight to Feature-equal
            end-if
        until all the input features are read
        Sort the case by a descending order of Feature-equal
until all the cases in the case base are read
```

Figure 4.8: Feature-based similarity measure on feature model

```
Repeat
    Read a case in the case base
    Repeat
        Read an optional feature
        Find the according optional feature in the case base
        Read Tolerance-band
        Read Weight
        if the optional features equal in a tolerance-band
            then add Weight to Optional-feature-equal
        end-if
    until all the optional features are read
    Sort the case by a descending order of Optional-feature-equal
until all the cases in the case base are read
```

Figure 4.9: Feature-based similarity measure on optional features

```
Repeat
    Read an item in the ranking list of feature-based similarity measure on
optional features as A
    Rank A
    Repeat
        Read an item in the ranking list of feature-based similarity measure on
        feature model as B
        if A and B are the same case
            then rank B
        end-if
    until A and B are the same case
    Calculate the weighed sum of the ranking of A and B
    Sort the cases according to descending sum
until all the items in the feature-based similarity measure ranking list are read
```

Figure 4.10: Feature-based similarity measure

4.3.3 Similarity assessment

The previous sections have described two methods of measuring similarity. Compared to feature similarity, structure similarity is usually recognised as the first factor to be considered, because the aim of similarity evaluation is to use an existing design model as the basis for further design. The more similar a structure is to a desired specification, the easier the modification will be. On the other hand, feature similarity is also important due to the ease with which features can be compared. As for the difference between function and structure, function is the basis of a design and determines its purpose, while structure reflects the organisation of the design. With regard to structure, an assembly model and a part model represent the general organisation of a design; on the other hand, a geometric model explicitly expresses a design. As previously mentioned, geometric models are easily modified so that they are less important during design reuse. Due to the complex nature of a design, users may encounter different situations and may require different similarity measures.

This research proposes a numerical weighting method to address this problem. The user may define a parameter $w(0 \leqslant w \leqslant 1)$ to signify the importance of each measure. The overall similarity measure is the sum of the weighted similarity values. Thus, based on the similarity values for the feature model, function model, and specified levels of structure model, the overall similarity measure S can be obtained as follows:

$$S = w_1 N_{feature} + w_2 N_{function} + w_3 N_{structure} \quad (0 \leqslant w_1, w_2, w_3 \leqslant 1) \tag{4}$$

where $N_{feature}$, $N_{function}$ and $N_{structure}$ denote the positions of a design on the lists ranking its feature, function, and structure similarity with a given design. w_1, w_2 and w_3 denote the weights for feature, function and structure similarity measures. The design with the largest S is the most similar to the specified design. The method of similarity assessment is defined in Figure 4.11.

By adopting this weighting method, the user can freely choose from the measures to be employed and decide their importance in a particular situation. For example, if someone wants to compare the similarity of a function model only, they can set 0 as the weights for all the other measurements.

The whole procedure is illustrated in Figure 4.12. The ICAD code for the proposed similarity measure approach is shown in Appendix C.

```
Repeat
    Read an item in the feature-based similarity measure ranking list as A
    Rank A
    Repeat
        Read an item in the ranking list of structure-based similarity measure
        on function model as B
        if A and B are the same case
            then rank B
        end-if
    until A and B are the same case
    Repeat
        Read an item in the ranking list of structure-based similarity measure
        on structure model as C
        if A and C are the same case
            then rank C
        end-if
    until A and C are the same case
    Calculate the weighed sum of the ranking of A, B and C
    Sort the cases according to descending sum
until all the items in the feature-based similarity measure ranking list are read
```

Figure 4.11: Similarity assessment

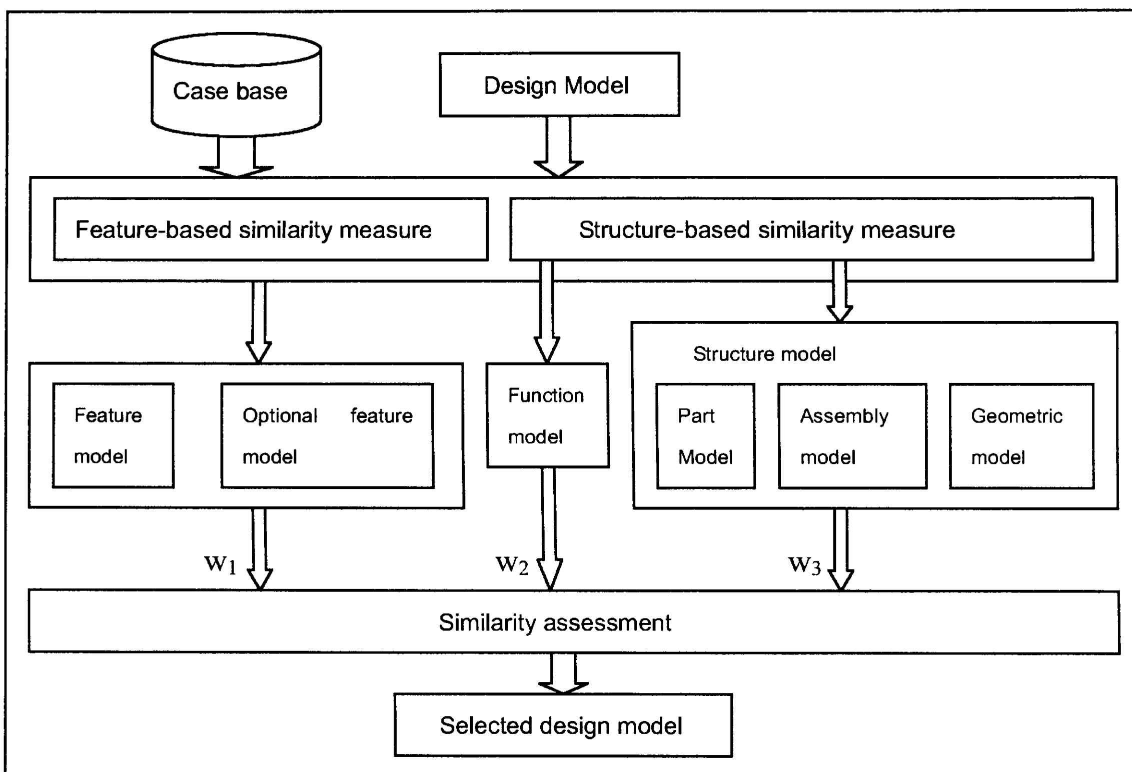


Figure 4.12: An illustration of similarity assessment method

4.4 An illustrative example

Design models of car bodies are used to exemplify the method. A representation of a car body using style lines is illustrated in Figure 4.13. A car body can be classified into three assemblies: engine compartment, passenger cabin, and luggage compartment. Each assembled part is composed of a few parts. For example, a passenger cabin is composed of roof panel, left side, and right side.

Assume there are three models *d0*, *d1*, and *d2*, where *d0* is the input model while *d1*and *d2* are the models in the case base. *d1* and *d2* are to be compared to *d0* in order to determine which of them is more similar to *d0*.

Feature-based similarity measure

Table 4.1 shows comparisons between the feature models of *d0* and *d1* using the feature-based similarity measure (S_f). Various features are compared, including type, length, width, height, wheelbase, front track, rear track, weight, fuel consumption, top speed, and acceleration 0-100km/h. The parametric features require offsets to constrain the range of the equal features, which is given in Figure 4.14. Weights are assigned to each feature as shown in Figure 4.15.

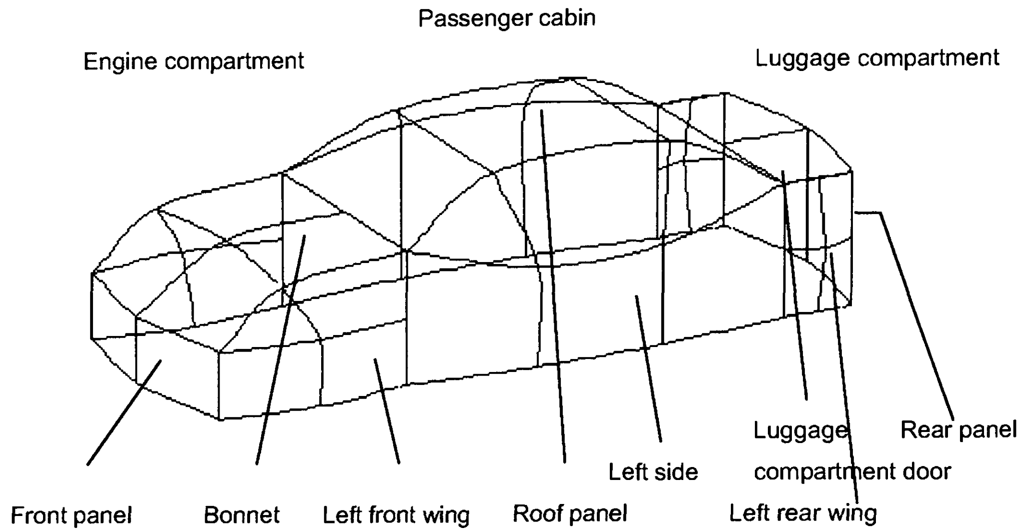


Figure 4.13: Car body represented by style lines

	k_i	Tolerance band	*d0*	*d1*	Comparison with *d1*	*d2*	Comparison with *d2*
Type	0.5	-	Saloon	Saloon	E	Compact	NE
Length	0.5	±0.1	5029	4775	E	4262	NE
Width	0.5	±0.08	1902	1800	E	1751	E
Height	0.2	±0.05	1492	1435	E	1408	NE
Wheelbase	0.2	±0.08	2990	2830	E	2725	NE
Front track	0.2	±0.05	1578	1512	E	1484	NE
Rear track	0.2	±0.05	1582	1526	E	1493	NE
Weight	0.3	±0.2	1865	1570	E	1375	E
Fuel consumption	0.3	±1	15.5	12.2	E	9.7	E
Top speed	0.3	±0.1	237	226	E	201	NE
Acceleration 0-100km/h	0.3	±0.1	8.1	9.1	NE	11.1	NE

E: equal; NE: not equal

Table 4.1: Comparison of the feature models of *d0* and *d1* and of *d0* and *d2* using S_f

Accept! Default! Cancel!

Cbr

Enter tolerance-band for length,width, height, wheelbase, fronttrack, reartrack, car-type, fuel-consumption, top-speed, acceleration and weight (a lisp expression)

Select Accept! or Default! to use default (0.1 0.08 0.05 0.08 0.05 0.05 1 0.1 0.1 0.1 0.3)

(LIST 0.1 0.08 0.05 0.08 0.05 0.05 1 1 0.1 0.1 0.2)

Figure 4.14: Input tolerance band

Choice Attribute

Accept! Default! Cancel!

Cbr

Enter weights for length,width, height, wheelbase, fronttrack, reartrack, car-type, fuel-consumption, top-speed, acceleration and weight (a lisp expression)

Select Accept! or Default! to use default (0.5 0.5 0.2 0.2 0.2 0.2 0.5 0.3 0.3 0.3 0.3)

(LIST 0.5 0.5 0.2 0.2 0.2 0.2 0.5 0.3 0.3 0.3 0.3)

Figure 4.15: Input weights for features

Table 4.2 shows a comparison between *d0* and *d1* on the optional features using S_f. Attributes including length, width, and height of passenger cabin and the length of luggage compartment are extracted from the structure model and used as optional features for comparison. As for the feature model, the parametric feature is given a tolerance band and every feature is assigned a weight as shown in the table.

According to Table 4.1 and Table 4.2, the result of comparing *d0* and *d1* and *d0* and *d2* is listed in Table 4.3.

The feature-based similarity measure S_f can be calculated according to Eq.(3):

$S_f(d0, d1) = \Sigma k_i N_e - \Sigma k_i N_m = 0.5 \times 5 + 0.3 \times 5 + 0.2 \times 4 = 4.8$

$S_f(d0, d2) = \Sigma k_i N_e - \Sigma k_i N_m = 0.5 \times 2 + 0.3 \times 2 - 0.3 \times 1 = 1.3$

$S_f(d0, d1) > S_f(d0, d2)$, which means that from the perspective of features, *d1* is more similar to *d0*.

	k_i	Tolerance band	*d0*	*d1*	Comparison with *d1*	*d2*	Comparison with *d2*
Passenger cabin\length	0.5	±0.1	2090	1967	E	2725	NE
Passenger cabin\width	0.5	±0.1	1902	1800	E	1751	E
Passenger cabin\height	0.3	±0.06	1120	1100	E	1000	NE
Luggage compartment\length	0.3	±0.1	1139	1080	E	-	M

E: equal; M: missing feature; NE: not equal

Table 4.2: Comparison of the optional features of *d0* and *d1*and of *d0* and *d2* using S_f

	d0 and *d1*	*d0* and *d2*
Number of equal features	*k*=0.5: 2+3=5, *k*=0.3: 3+2=5, *k*=0.2: 4+0=4.	*k*=0.5: 1+1=2, *k*=0.3: 2+0=2, *k*=0.2: 1+0=1.
Number of missing features	0	*k*=0.3: 1

Table 4.3: Summary of comparison results

Structure-based similarity measure

Since the function models of *d0, d1* and *d2* are all the same,

$D_s(fun)\{d0, d1\} = 0$,

$D_s(fun)\{d0, d2\} = 0$,

and $S_s(fun)\{d0, d1\} = S_s(fun)\{d0, d2\}$.

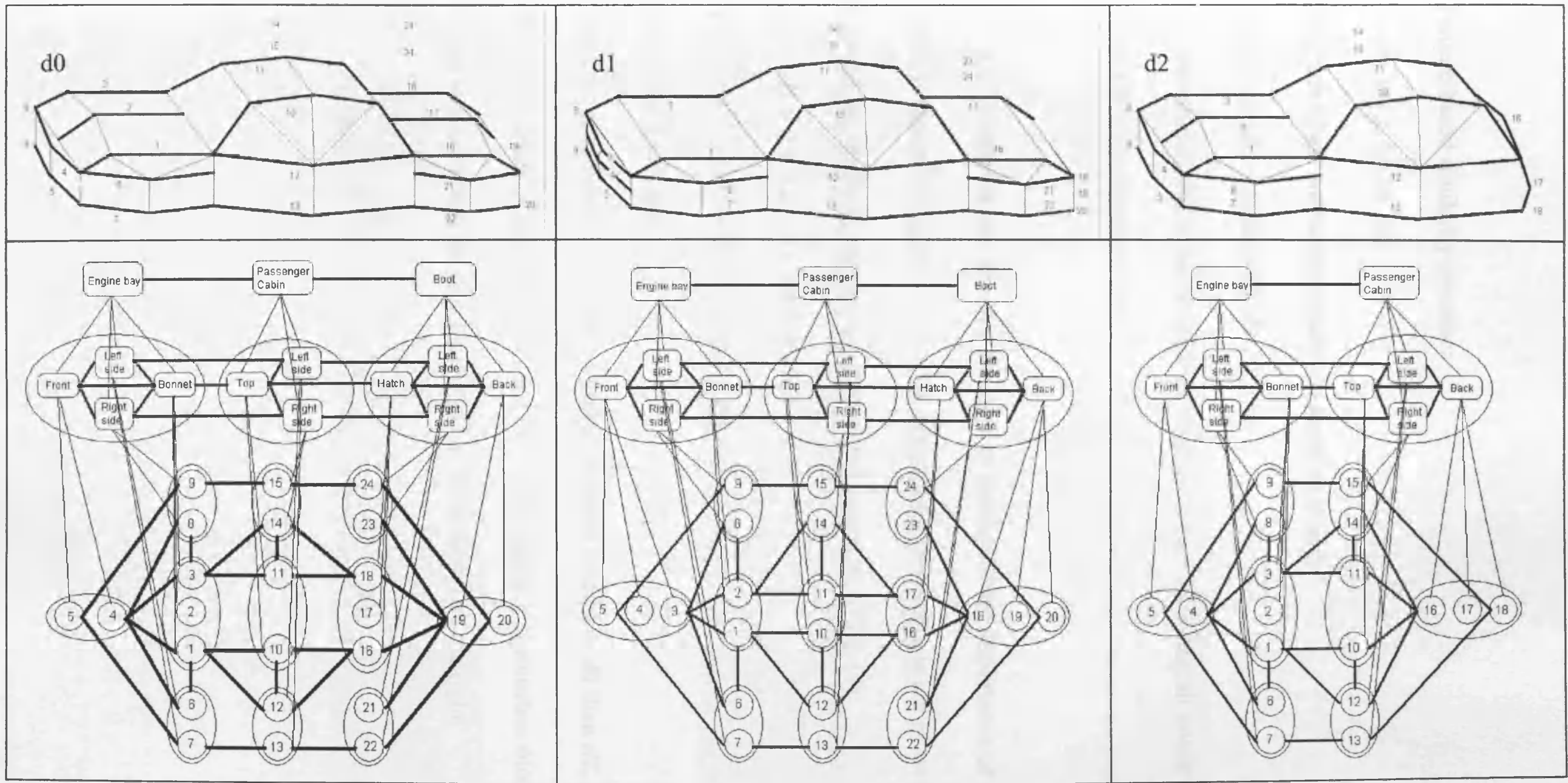
In this example, the structure-based similarity measure S_s is applied at all the levels of the structure model. The illustration of the structure models of *d0, d1* and *d2* are listed in Figure 4.16. Different styles of lines are used to distinguish the relationships: The thick line represents the positional relationship between the parts, which includes non-parametric relationship and parametric relationship; the thin line identifies the subordinate relationship between parts or between parts and assemblies. As described in Section 4.3.1, from Eq. (1) the result of comparison is obtained as:

$D_s(str)\{d0, d1\} = 8$,

$D_s(str)\{d0, d2\} = 25$.

Therefore, $S_s(str)\{d0, d1\} > S_s(str)\{d0, d2\}$, which means that from the perspective of structures, *d1* is more similar to *d0*.

So far, the following results have been obtained:



(a) (b) (c)

Figure 4.16: Structure models of *d0*, *d1*, *d2* (a), (b) and (c)

Feature-based similarity measure:

$S_f(d0, d1) > S_f(d0, d2)$.

Structure-based similarity measure on function model:

$S_s(fun)\{d0, d1\} = S_s(fun)\{d0, d2\}$.

Structure-based similarity measure on structure model including all levels:

$S_s(str)\{d0, d1\} > S_s(str)\{d0, d2\}$.

A set of weights is provided by the user to determine the importance of each result, which is shown in Figures 4.17 - 4.19. Weights for this example are given as (0.4, 0.3, 0.3) and the similarity assessment is executed according to Eq. (4):

$S(d0, d1) = 0.4 \times 2 + 0.3 \times 2 + 0.3 \times 2 = 2$

$S(d0, d2) = 0.4 \times 1 + 0.3 \times 2 + 0.3 \times 1 = 1.3$

$S(d0, d1) > S(d0, d2)$.

Thus, the conclusion can be drawn that *d1* is more similar to *d0* than *d2*. Figure 4.20 shows the result of the retrieved case *d1*. In the figure, a geometric model of *d1* is shown on the left and the structure model of *d1* is shown on the right.

Choice Attribute

Accept! Default! Cancel!

Cbr

Enter weight for feature-based similarity measure (a number greater than or equal to 0 and less than or equal to 1)

Select Accept! or Default! to use default 0.4

0.4

Figure 4.17: Weights for feature-based similarity measure

Choice Attribute

Accept! Default! Cancel!

Cbr

Enter weight for structure-based similarity measure on function (a number greater than or equal to 0 and less than or equal to 1)

Select Accept! or Default! to use default 0.3

0.3

Figure 4.18: Weights for structure-based similarity measure on function model

Choice Attribute

Accept! Default! Cancel!

Cbr

Enter weight for structure-based similarity measure on structure (a number greater than or equal to 0 and less than or equal to 1)

Select Accept! or Default! to use default 0.3

0.3

Figure 4.19: Weights for structure-based similarity measure on structure model

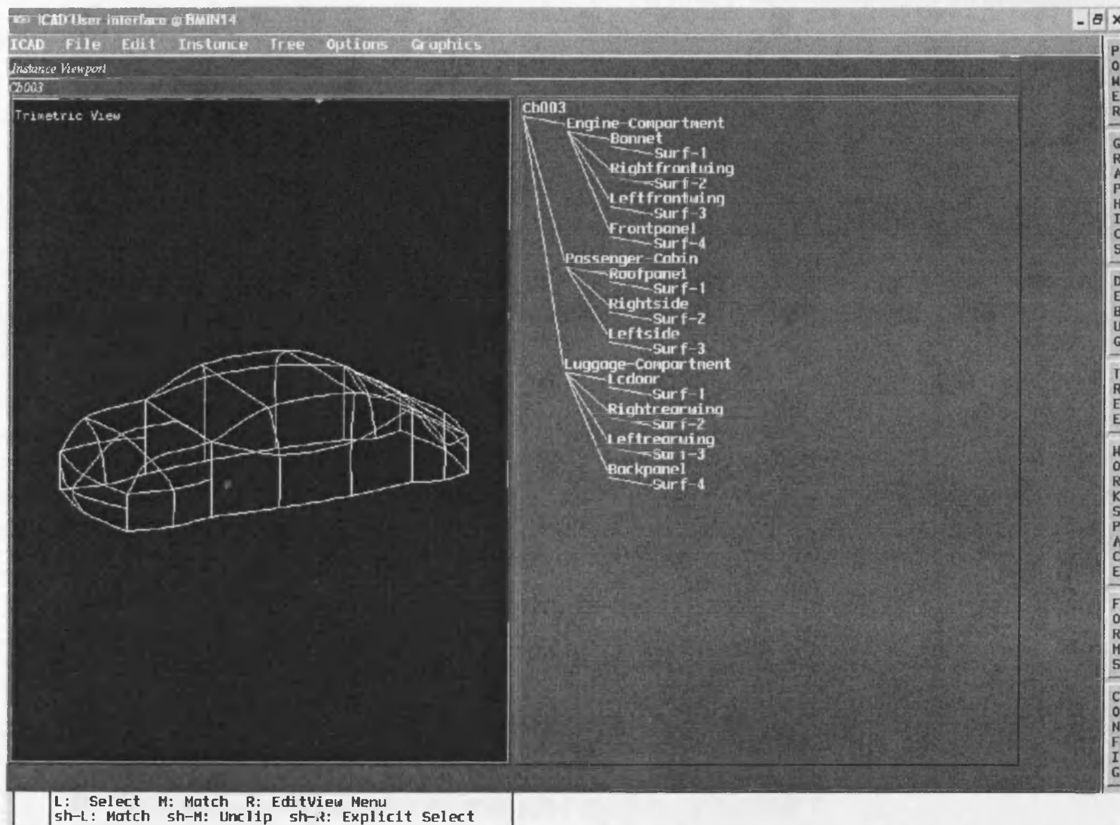


Figure 4.20: The retrieved case *d1*

4.5 Discussion

This chapter has proposed a new approach to measuring the similarity of designs using graph-based representations. It enables the user, when given a new design, to query a case base of design models, which are in the form of attributed graphs containing design knowledge about function, feature, and structure and to identify the existing designs with graphs similar to the new design. A graph consists of elements, attributes, and relationships: G = [E, A, R]. Two similarity measure methods (structure-based similarity measure and feature-based similarity measure) are applied on this form concurrently, in order to perform a complete comparison.

The approach of similarity measure can be outlined as follows:

- ➢ Specify a compared design model.
- ➢ Assess feature-based similarity and structure-based similarity.
  - ◆ Feature-based similarity measure on feature model and structure model;
  - ◆ Structure-based similarity measure on function model;
  - ◆ Structure-based similarity measure on structure model;
- ➢ Assign weights to each measure.

Compared to the previous methods, the proposed approach has the following advantages:

- It incorporates design knowledge to assess design model similarity. Previous research had the problems that product information was insufficiently incorporated to compare similarity of complex designs and that complicated geometric comparisons were conducted without taking account of the criteria of domain specific technical knowledge. Design is a stage in which human knowledge is involved. It is the knowledge rather than the shape model that should be compared. In the proposed method, design knowledge such as functions, features and structures is extracted and used for the comparison of designs.

- It assesses similarity by considering both the features and the structure of the product. In previous research, either feature or structure is compared individually, which will cause an incomplete comparison. The proposed research develops an approach to measure the similarity of the designs considering all these factors.

- It achieves flexibility of measuring similarity. User preference is applied to fulfil the designer's requirements. At different stages of the similarity assessment operation, users are involved by giving their preferences to guide the operation, for example, assigning weights, specifying optional features, and selecting the levels of structure model to be compared.

4.6 Summary

This chapter has presented a novel approach for measuring similarity of design. At first, the generation of compared model was discussed. A method for measuring the similarity between this model and existing models in the case base was then addressed. The method consists of structure-based similarity measure, feature-based similarity measure, and similarity assessment. Finally, an example of the comparison of car body was given to exemplify the proposed method.

Chapter 5

Fractal-based Adaptation and Fractal-Based Re-design

5.1 Preliminaries

Design is a complex open-ended task and it is unreasonable to expect a case base to contain representatives of all possible designs. Adaptation is, therefore, a desirable capability for case-based design systems. Research in adaptive design considers how to adapt existing design cases in order to tackle new design problems. However, case adaptation is often considered to be the most difficult part of a case-based reasoning system (Purvis & Pu, 1995). The difficulties arise from the fact that adaptation often does not converge, especially if it is not done in a systematic way. Furthermore, in the design domain, multiple cases must be considered in conjunction in order to solve the new problem, resulting in the difficulty of how to efficiently combine the cases into a global solution for the new problem.

The proposed work develops a systematic approach for adaptation in the CBR process. The approach adapts the fractal-like design model, which was discussed in Chapter 3, with the help of fractal characteristics. The research presented in this thesis

investigates an efficient way to develop a systematic method for automating adaptation, to investigate the use of fractal characteristics in adaptation and to use the performance and goals of a design problem to guide the process of adaptation. A description of the fractal-based adaptation method is presented with examples in Section 5.2. Section 5.3 reports the whole process of Fractal-Based Re-design.

5.2 Fractal-based adaptation

Design case adaptation is a complicated problem solving process. Design case adaptation involves identifying the differences between the new and the old design contexts and modifying the previous design by taking those differences into account. The issues to be addressed in developing an adaptation method include the representation of domain knowledge to be used in adaptation and the strategy for adaptation.

For strictly engineering optimisation problems, representations should be direct (i.e. they should encode possible solutions) and parameterised (allowing only for slight variations). They usually incorporate domain knowledge in order to make the search more efficient (Kicinger et al., 2005b). Case adaptation can be considered as an optimisation problem. Traditionally in case-based reasoning, adaptation knowledge has taken the form of solution transformation rules. Adaptation knowledge structures and processes can take many forms, from the use of declarative rules for substitutional

adaptation to the use of more complex operator-based or derivational knowledge in first-principles approaches. In this research, the adaptation knowledge is represented through generalised schemes, i.e. fractals, which are stored within the individual design cases as discussed in Chapter 3.

In adaptive design, depending on the demands of the requirements list, the fractal structure can be modified by the variation, addition, or omission of individual sub-fractals or by changes in their combination. An advantage of setting up a fractal structure is that it allows a clear definition of sub-designs. If an assembly can be substituted directly as a fractal, the subdivision of the fractal structure can be discontinued at a fairly high level of complexity. In those cases requiring further development, the division into sub-fractals of decreasing complexity can be continued until the search for a solution seems promising.

The purpose of re-design can be divided into two. These are modifying the performances and improving the goals of the existing designs. In this research, these two issues are resolved with the help of the fractal characteristics. This research proposes performance revision and goal-oriented substitution as the main manipulations of adaptation.

5.2.1 Performance revision

As discussed in Chapter 3, every design case can be represented as a group of fractals working together to deliver a particular overall performance. A description of a new design problem is represented as a group of sub-performances that may be provided by a number of fractals. These sub-performances are indexed to identify the fractals. The new sub-performances are compared to the sub-performances of the retrieved design case as shown in Figure 5.1. The differences between the two sets of sub-performances reveal the discrepancies between the two cases. In this way, it can be seen where the transformation to the old design case should be applied so as to fulfil the new requirements.

Once the discrepancies have been identified, the adaptation process starts. If there is performance discrepancy, the fractals are restructured to adapt to the new design problem. The system automatically selects the operation according to the performance discrepancies under different situations as outlined below.

```
Repeat
      Read the performance in the retrieved case
      Add the performance in a list Retrieved-case-performance
until all the performance is added
Repeat
      Read an item in the list Retrieved-case-performance
      if the item is in the input performance
            then add it in Kept-performance
      end-if
until all items in the list Retrieved-case-performance are read
Removed-performance<- Retrieved-case-performance - Kept-performance
Repeat
      Read an item in the list Kept-performance
      if the item is in the input performance
            then remove it from Input-performance
      end-if
until all items in the list Retrieved-case-performance are read
Added-performance<-Input-performance
```

Figure 5.1: Comparison of performance

- Situation 1: The performances are identical in the new problem and in the old case.
- Situation 2: Some performances in the old case are not in the new problem, while all performances in the new problem are in the old case.
- Situation 3: Some performances in the new problem are not in the old case while all performances in the old case are in the new problem.
- Situation 4: Some performances in the new problem are not found in the old case and some performances in the old case are not found in the new problem.

This research proposes the following rules for adapting the design case:

Operation 1: Add component.

Operation 2: Remove component.

- Rule 1: if it belongs to situation 1, no manipulation is needed.
- Rule 2: if it belongs to situation 2, those fractals whose performance is not in the new problem will be "removed".
- Rule 3: if it belongs to situation 3, fractals from other cases that contain those distinguished performances will be "added";
- Rule 4: if it belongs to situation 4, which is a combined situation of situation 2 and situation 3, the solution is to "remove" the redundant fractals and to "add" the required fractals.

The process of performance revision is defined in Figure 5.2. The ICAD code for performance revision is shown in Appendix D.

5.2.1.1 An illustrative example

Presented here is an example of performance revision for a car body. It is supposed that the aim of re-design in this example is to re-design a saloon car body to a compact one. At first, the system asks the user for the performances, as shown in Figure 5.3. In this example, the performances for a compact car are listed below:

- To provide space for the engine.
- To provide space for the passengers and the luggage.

Then the system asks the user to select the case for re-design as shown in Figure 5.4. A saloon car body "cb003" is selected as the re-design object. The performances of "cb003" (the saloon car) are listed below:

- To provide space for the engine.
- To provide space for the passengers.
- To provide space for the luggage.

As can be seen, the first performance exists in the previous case but the second performance does not. According to the rules, operations "add" and "remove" are used to revise the existing design. A fractal providing the second performance is

selected from the case base to help construct the new design solution. This selection is conducted randomly in the case base. The whole process and the result are shown in Figure 5.5. As shown in the figure, the passenger cabin in cb002 is used to replace the cabin in cb003, and the luggage compartment of cb003 is removed to obtain the result.

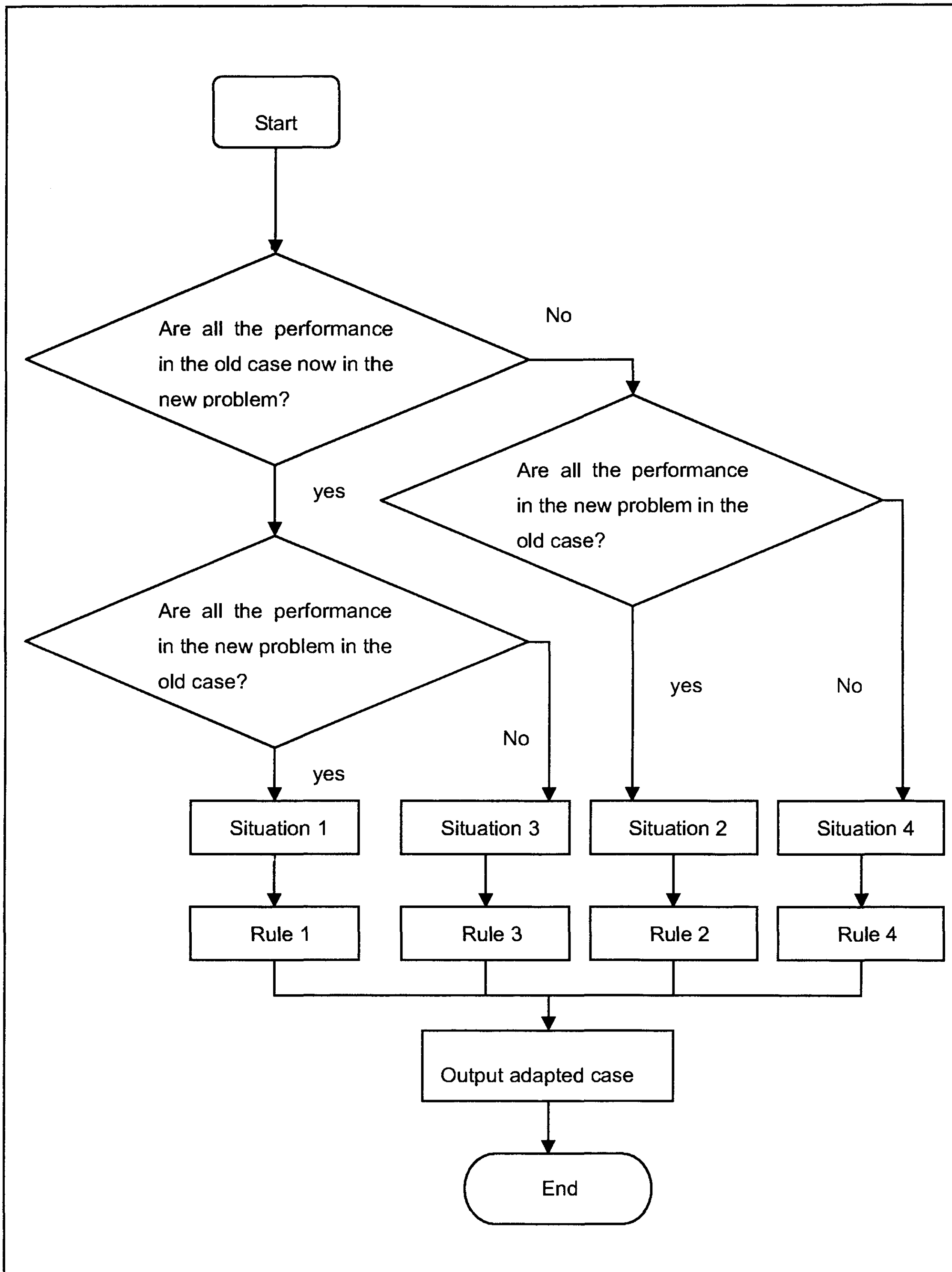


Figure 5.2: The process of performance revision

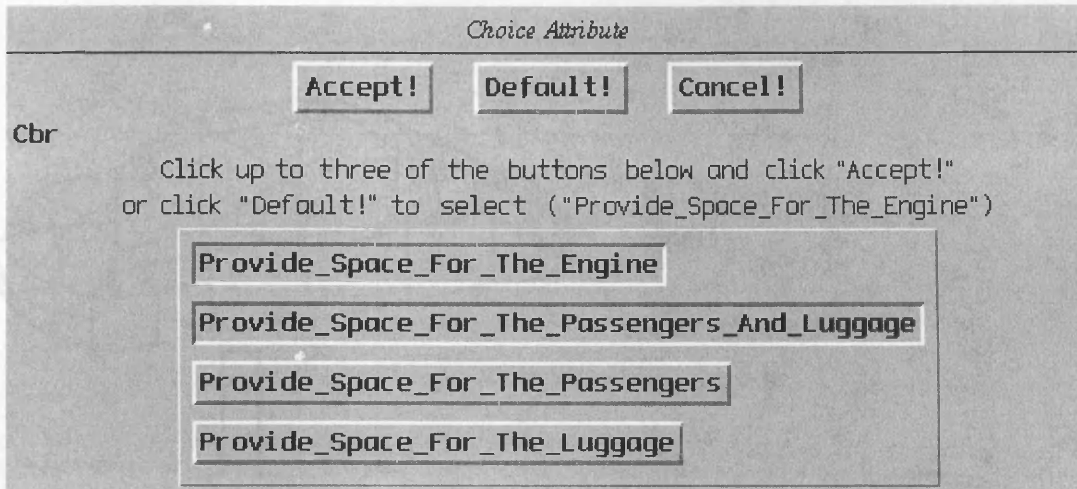


Figure 5.3: Performance selection

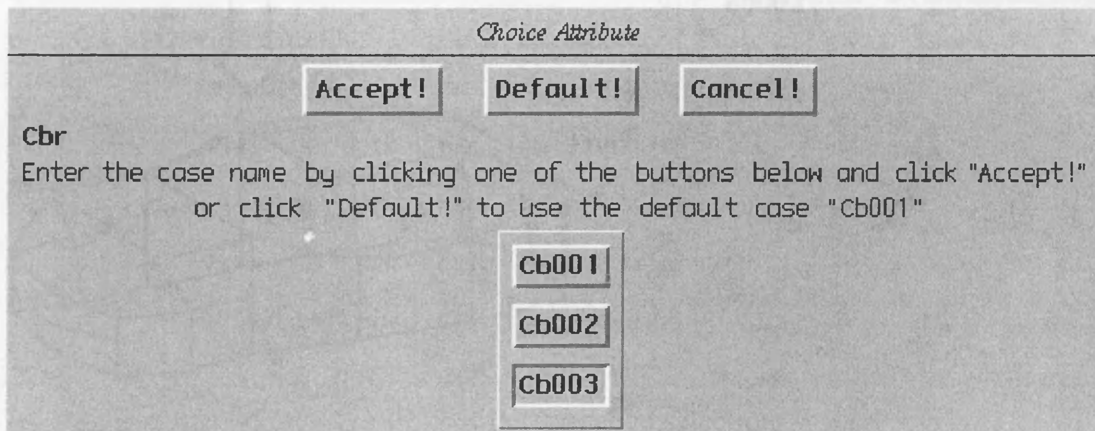


Figure 5.4: Re-design case selection

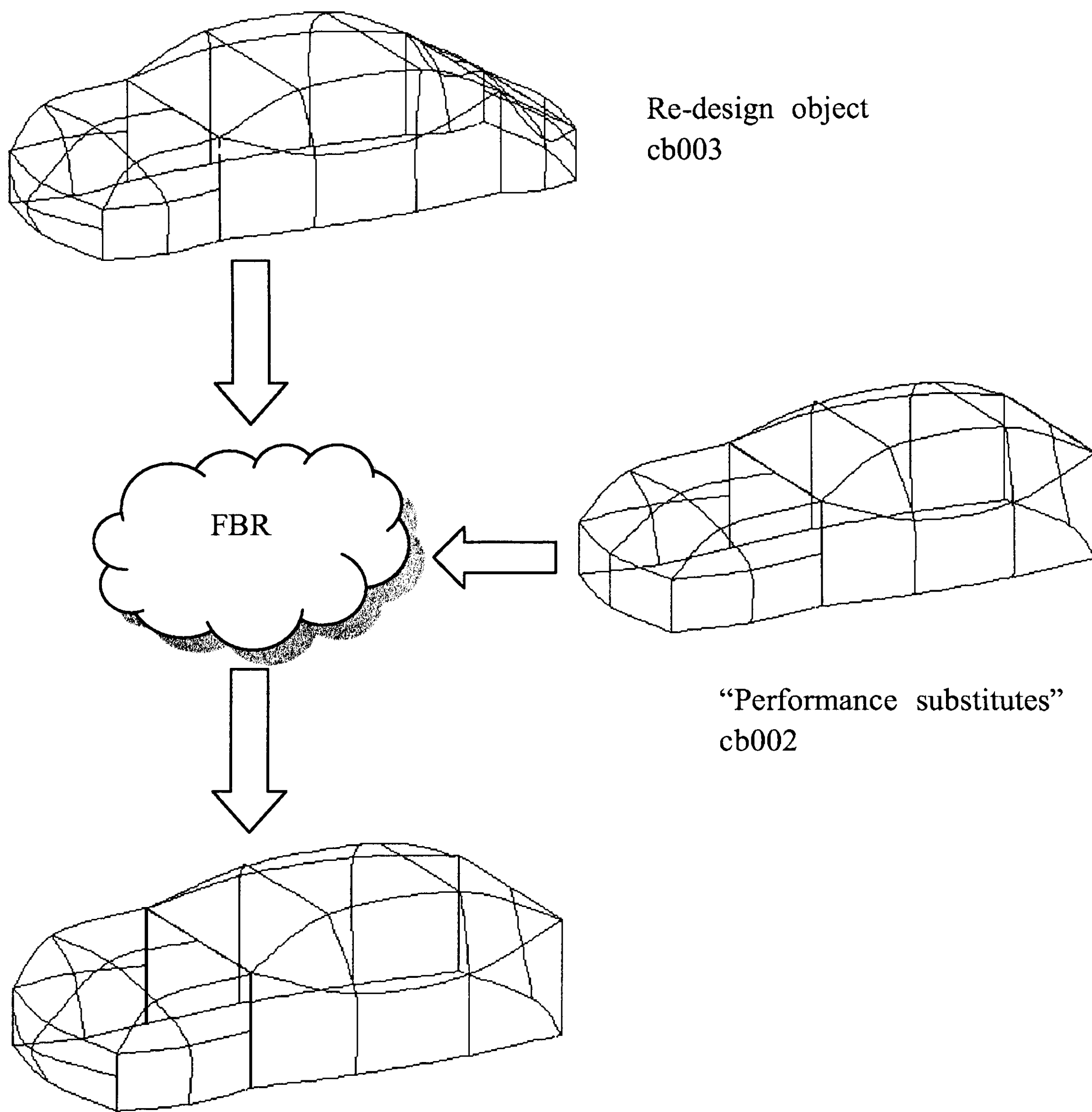


Figure 5.5: An illustration of the process of re-design

5.2.2 Goal-oriented substitution

Designers often encounter such situations where the existing designs no longer satisfy the particular requirements in some specific aspects. In this situation, the aim of re-design is to improve the existing design towards particular goals. Goal-oriented substitution is developed to solve this kind of problem.

Every fractal has individual goals. As defined in Section 3.3.3, a goal consists of four parts:

- An attribute which has effect on the goal.
- An operation on the attribute, which can be "maximise" or "minimise".
- Type of goal, which can be individual goal (:i) (the goals that cannot propagate to another level) or corporate goal (:c) (the goals that are able to propagate to another level).
- The corporate goal to which this goal contributes.

The goals of each fractal are somewhat different from those of the others. To achieve these goals coherently, goal consistency should be maintained. A goal formation process is proposed here. Under the architecture of the fractals, the goals of the fractals form a structure that describes the context and relationship of the goals. A Goal Dependency Graph (GDG) can be generated. As a graph representing the dependencies between the goals, a GDG propagates the corporate goals to individual

goals. The process of the input goals propagation is defined in Figure 5.6.

Inside the individual goals, the attributes and the desired operations can be obtained. This is the knowledge which guides the design on how to obtain the goals. As some operations on the same attributes might be contradicted, these goals will be removed. Then a simplified GDG is obtained. The identification of contradicted goals and the generation of a simplified GDG are defined in Figure 5.7. These attributes and operations can be used as constraints to query the case base. Fractals containing more suitable attributes to achieve the goals are selected to substitute the fractals in the previous design. The fractal structures after substitution are self-organised by changing the specific substituted fractals' attributes. The characteristic of self-similarity guarantees that the performance of the fractals is not affected. The generation of substitution is defined in Figure 5.8. The ICAD code for goal-orientation substitution is shown in Appendix E.

```
Find Individual-goals in the case base and make a list
Repeat
    Read an input goal in the list of Input-goals
    Repeat
        Search in the list of Individual-goals
        if the corporate goal that individual goal contributes equals the input goal
            then add the individual goal to a list A
        end-if
    until all the Individual-goals are searched
    list B<-Add the input goal to list A
    GDG<-Add list B
until all the input goals are read
```

Figure 5.6: The process of input goal propagation

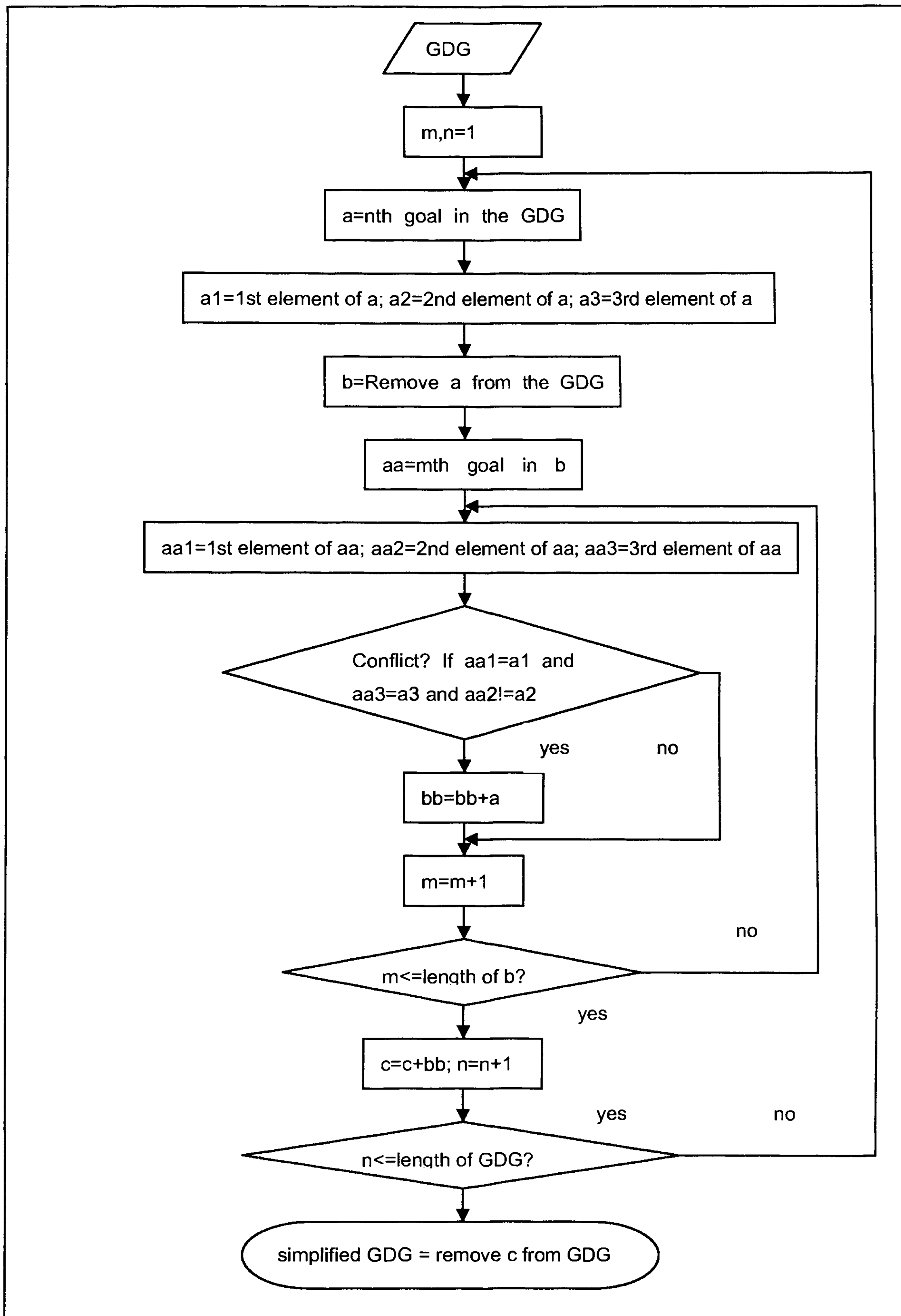


Figure 5.7: The process of generation of simplified GDG

```
Repeat
    Read a goal in SGDG as A
    Repeat
        Search in the case base
        Check the name of object B
        If object B is the fractal where A is from
            Then if the attribute of B > the attribute of A
                    and Operation on attribute of A is maximise
                or if the attribute of B < the attribute of A
                    and Operation on attribute of A is minimise
                        then substitute B to replace A
                end-if
        end-if
    until a substitution is found or all the case base are searched
until all the items in SGDG is read
```

Figure 5.8: Definition of generation of substitution

5.2.2.1 An illustrative example

The following is an example of automated adaptive design for goal-oriented substitution for a car body. The structure of the goals is illustrated in Figure 5.9. As can be seen in the figure, G1, G2 and G3 are affected by the sub-goals in the lower layered fractals. For example, G1 is affected by G1-G1, G1-G2 and G1-G3 in engine compartment, passenger cabin and luggage compartment, and G1-G1 is affected by another lower layered fractals. Figure 5.10 shows the GDG of the goals, which illustrate the dependencies between the goals, and shows how the corporate goals G1, G2 and G3 propagate to individual goals.

Suppose the aim of re-design is to improve all three goals. As can be seen, :G1-G2 (list :minimise :volume :c :G1) and :G2-G1 (list :maximise :volume :c :G2), :G1-G2-G1 (list :minimise :height :i :G1-G2) and :G2-G1-G1 (list :maximise :height :i :G2-G1), :G1-G2-G4 (list :minimise :length :i :G1-G2) and :G2-G1-G3 (list :maximise :length :i :G2-G1), :G1-G2-G3 (list :minimise :length :i :G1-G2) and :G2-G1-G2 (list :maximise :length :i :G2-G1) are contradicted. So these goals are removed, and a simplified GDG is:

:G1->(:G1-G1-G1 :G1-G1-G4 :G1-G1-G2 :G1-G1-G3 :G1-G2-G2 :G1-G3-G1 :G1-G 3-G4 :G1-G3-G3 :G1-G3-G2) :G3-> (:G3-G1)

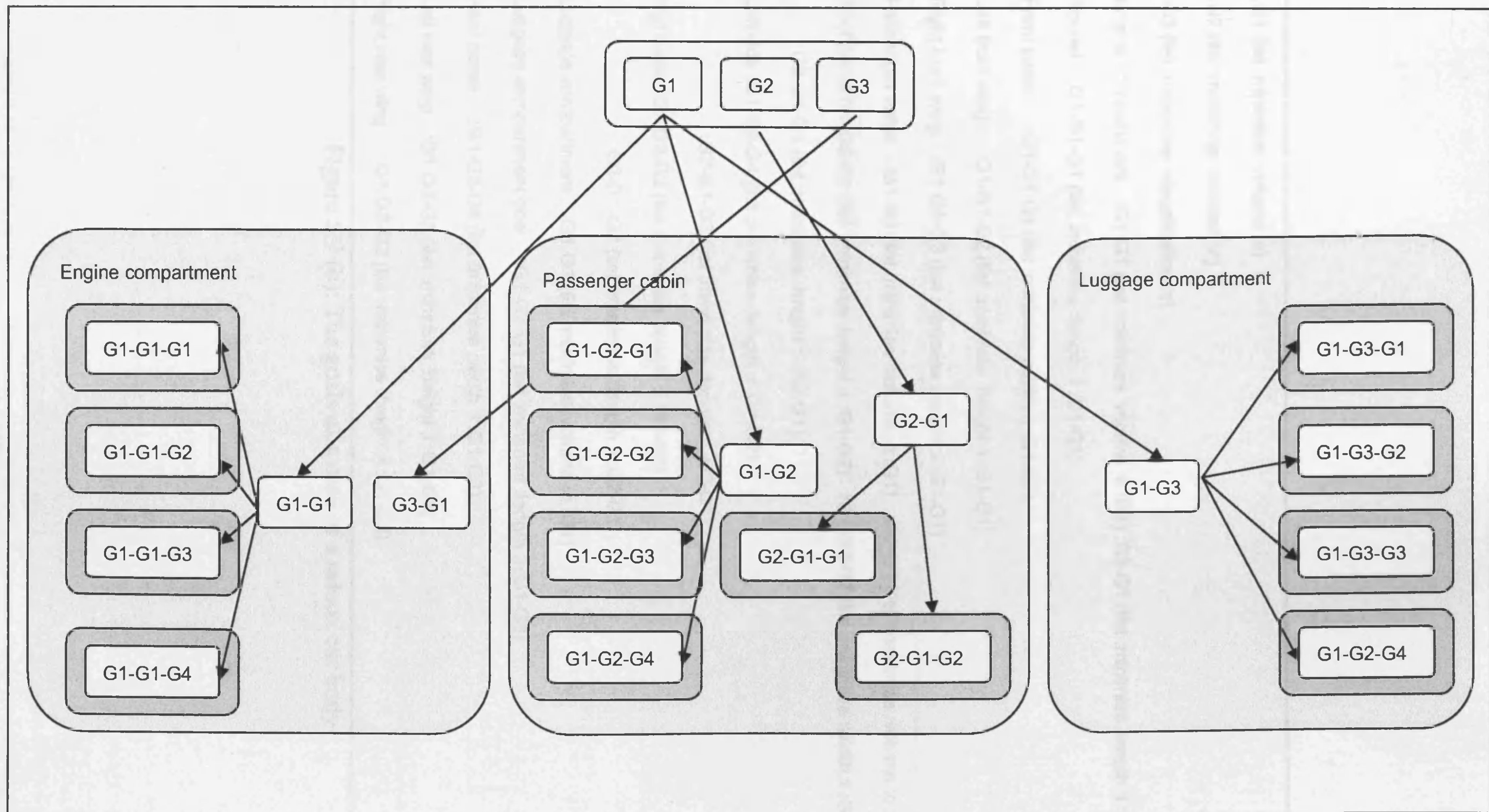


Figure 5.9 (a): Structure of the goals of a case of a saloon car body

:G1 (list :minimise :volume :c)

:G2 (list :maximise :comfort :c)

:G3 (list :maximise :visualisation :c)

Engine compartment :G1-G1 (list :minimise :volume :c :G1) :G3-G1 (list :minimise :length :i :G3)

Bonnet :G1-G1-G1 (list :minimise :length :i :G1-G1)

Front panel :G1-G1-G4 (list :minimise :width :i :G1-G1)

Left front wing :G1-G1-G2 (list :minimise :height :i :G1-G1)

Right front wing :G1-G1-G3 (list :minimise :height :i :G1-G1)

Passenger cabin :G1-G2 (list :minimise :volume :c :G1) :G2-G1 (list :maximise :volume :c :G2)

Roof panel:G1-G2-G1 (list :minimise :height :i :G1-G2) :G1-G2-G2 (list :minimise :width :i :G1-G2)

:G2-G1-G1 (list :maximise :height :i :G2-G1)

Left side :G1-G2-G4 (list :minimise :length :i :G1-G2)

:G2-G1-G3 (list :maximise :length :i :G2-G1)

Right side :G1-G2-G3 (list :minimise :length :i :G1-G2)

:G2-G1-G2 (list :maximise :length :i :G2-G1)

Luggage compartment :G1-G3 (list :minimise :volume :c :G1)

Luggage compartment door :G1-G3-G1 (list :minimise :length :i :G1-G3)

Rear panel :G1-G3-G4 (list :minimise :width :i :G1-G3)

Left rear wing :G1-G3-G3 (list :minimise :height :i :G1-G3)

Right rear wing :G1-G3-G2 (list :minimise :height :i :G1-G3)

Figure 5.9 (b): The goals of a case of a saloon car body

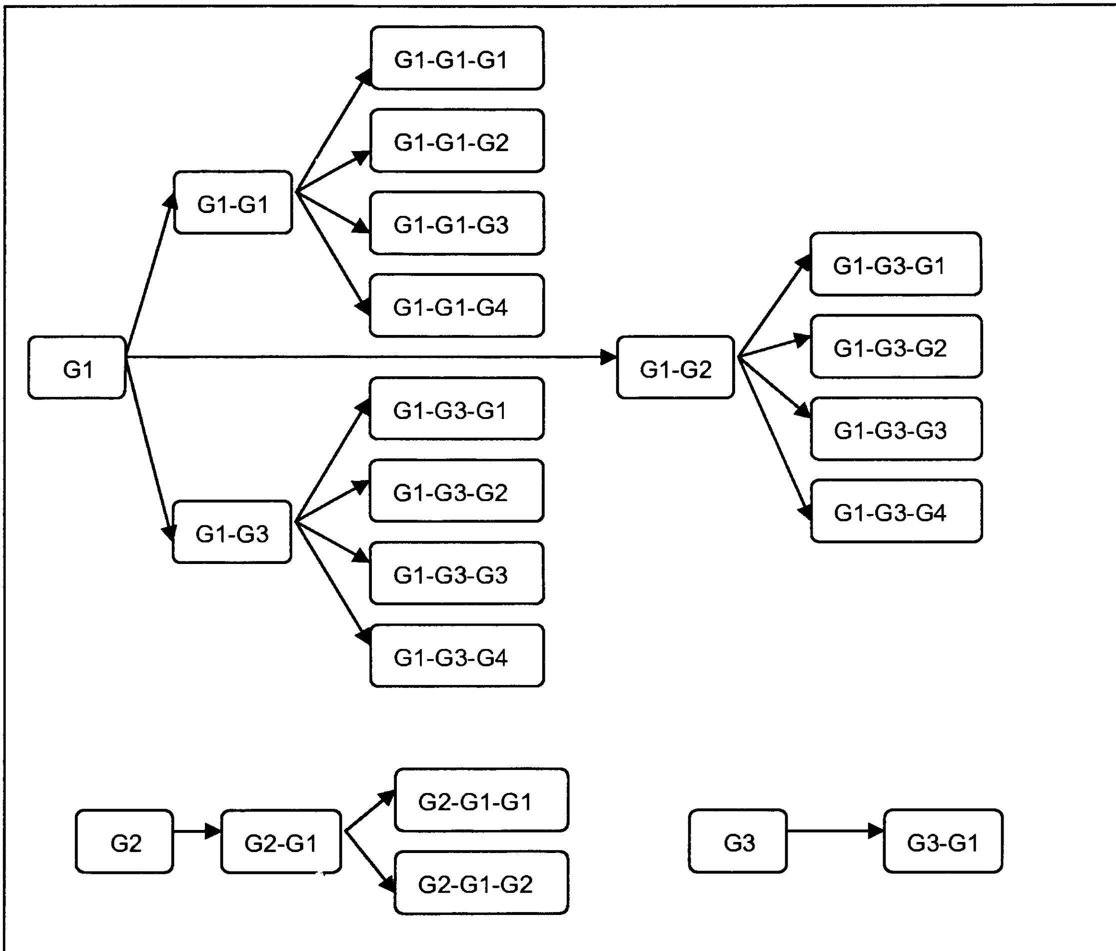


Figure 5.10: The GDG of a case of a saloon car body

These goals are used as constraints to query the case base for the fractals. In this example, bonnet with minimised length, front panel with minimised width, left front wing with minimised height, right front wing with minimised height, roof panel with minimised height, luggage compartment door with minimised length, rear panel with minimised width, left rear wing with minimised height, right rear wing with minimised height and engine compartment with minimised length are searched in the case base. If they are found, they will be used to substitute the according fractals of the re-design object.

5.3 Fractal-Based Re-design (FBR)

So far, with the adaptation strategies discussed in the previous section, together with case representation and case retrieval discussed in Chapter 3 and Chapter 4, a complete case-based design system has been established. These three parts are integrated as a systematic approach which can help the designer query a case base of design models, identify similar existing designs and adapt the retrieved design to suit the new situation. The framework of Fractal-Based Re-design (FBR) is illustrated in Figure 5.11. As shown in the figure, FBR starts with a design model describing the design requirements. Then the query model is compared to the models in the case base by a feature-based similarity measure and by a structure-based similarity measure. Next, these two similarity measures are assessed to determine the most similar cases in the case base compared to the query model. This case is then used as the base for

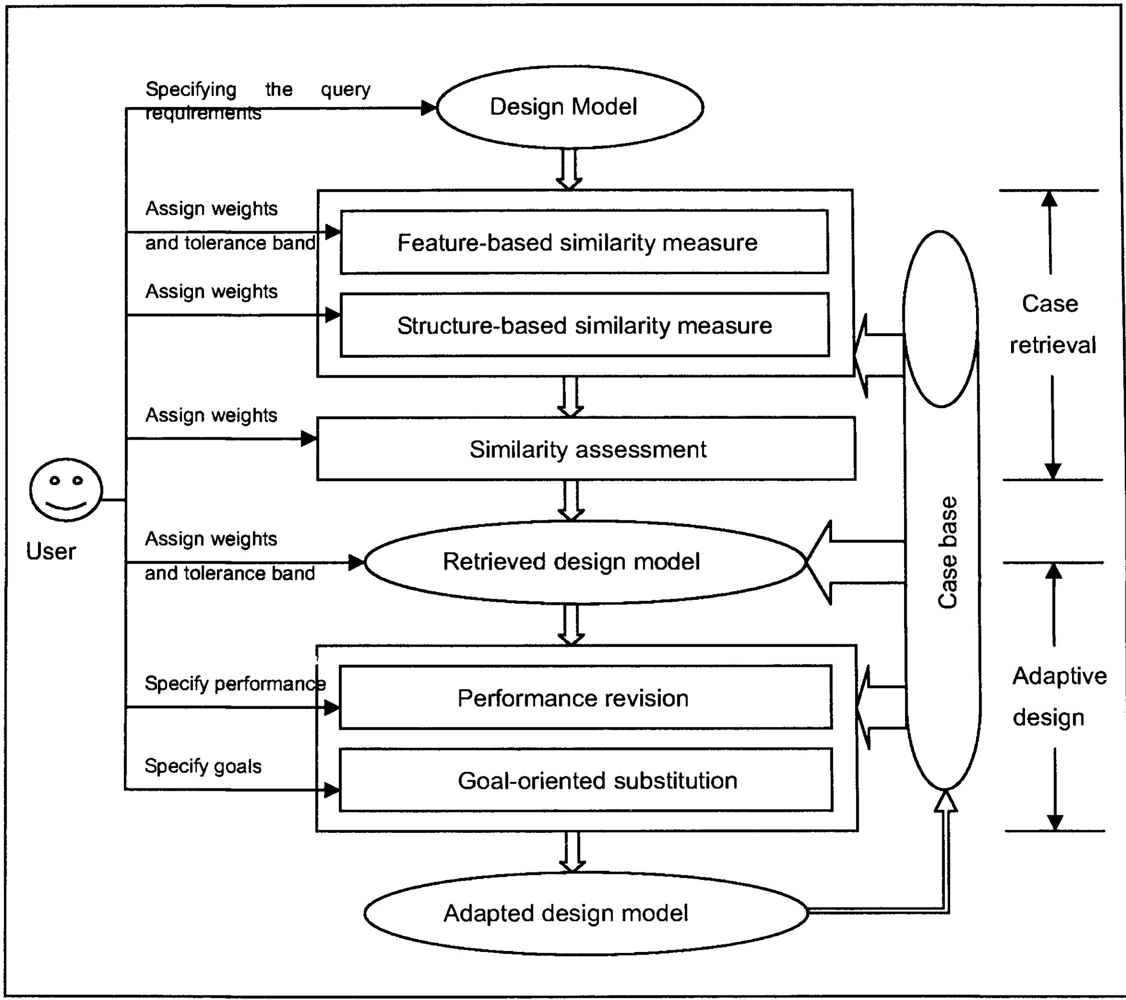


Figure 5.11: The framework of FBR

adaptive design. The user can choose to re-design according to performance or goals. Finally, the adapted design model is achieved. The designer is involved in the whole process of FBR by specifying inputs at each step. The designer can also assign weights to manipulations to control the process. Furthermore, the process can start not only with design requirements, but also start from the middle of the process, i.e. the designer can specify an existing model and begin adaptive design straight away. The ICAD code for FBR is shown in Appendix F.

5.4 Discussion

Adaptive design is a method used for conceptual design. With this method, when the design team is confronted with a new design problem, an old design case, which has some similarity to the requirements for the new design problem, can be retrieved and then modified to suit those new requirements. An advantage of this method is that most of the design knowledge is available once a design case has been retrieved. Only minor changes are required so as to modify the old design case.

Two main issues are involved in adaptive design. The first is the representation of the domain knowledge for the adaptation. In FBR, adaptation knowledge is distributed over fractals. The second is the strategy for adaptation. This chapter has addressed the strategy for adaptation. Design knowledge, such as performance and goals, is employed to develop a systematic method for automating adaptation. Performance

revision adapts a design by substituting fractals according to the identified performance; goal-oriented adaptation provides an efficient way for adaptation by using the goals of a design problem to guide the process of adaptation. The fractal specific characteristics such as self-similarity, self-organisation, goal-orientation play an important role in the automation of the adaptation process.

This chapter has also addressed Fractal-Based Re-design as a whole CBR process. FBR is intended to be a formal framework for tasks that include reasoning about design, formulating new design solutions, and the development of computer-based design aids. FBR aids in thinking about design at both the abstract and the practical levels and realises the automation of design processes.

5.5 Summary

In this chapter, the fractal-based adaptation strategy was presented with examples in Section 5.2. This included performance revision and goal-oriented substitution. Then Section 5.3 presented the whole process of FBR.

Chapter 6

A Case Study of Fractal-Based Re-design in Automotive Body Design

6.1 Preliminaries

Through the use of a case study, this chapter describes how the approach to conceptual design proposed in this thesis can be applied to support the design of an engineering product. The chapter starts with a short review of automotive body design in Section 6.2. A case study in automotive body design is presented in Section 6.3 and the results are discussed in Section 6.4.

6.2 Automotive body design

Automotive body design is a crucial task in automotive development. Body design has a profound impact on the vehicle's appeal and function. The body is the most expensive component of the vehicle to manufacture. It is designed under constraints arising from aesthetic considerations, structural and functional requirements, cost concerns and the availability of manufacturing resources. Engineers base their

judgments on specific experiences with prior designs. However, new engineers begin their work without these experiences and even experienced engineers may not have had experience with the most relevant designs for a particular problem. Multiple information resources exist to aid the design task, such as records of experiences with prior designs, stored in paper and electronic forms. However, it may be difficult or excessively time-consuming for engineers to find the information needed. Key questions for improving this process are how to provide better access to experiences and other engineering knowledge and how to improve the usefulness of the information when it is re-applied (Leake et al., 1999).

6.3 A case study of Fractal-Based Re-design in automotive body design

A case base has been built using the FDM technique proposed in Chapter 3. The models of automotive bodies in the case base are illustrated in Figure 6.1. Design knowledge such as functions, features, structures, performance and goals is represented as shown in the figure.

Cb001

FUNCTION MODEL

Function-1: Support chassis

Function-2: Provide space for passengers

Function-3: Stylise

Relationship ((Function-1, Function-2, and), (Function-2, Function-3, and), (Function-3, Function-1, and))

FEATURE MODEL

Behaviour feature

((Length 5029mm)

(Width 1902mm)

(Height 1492mm)

(Wheelbase 2990mm)

(Front track 1578mm)

(Rear track 1582mm))

Function feature

((Type Saloon)

(Fuel consumption 15.5 ltr/100km)

(Top speed 237km/h)

(Acceleration 0-100km/h 8.1sec))

Structure feature

((Weight 1865kg))

STRUCTURE MODEL

Sub-Fractals: Engine compartment, passenger cabin, luggage compartment

:performance (list :provide_space_for_the_engine :provide_space_for_the_passengers :provide_space_for_the_luggage)

:G1 (list :minimise :volume :c)

:G2 (list :maximise :comfort :c)

:G3 (list :maximise :visualisation :c)

Assembly model

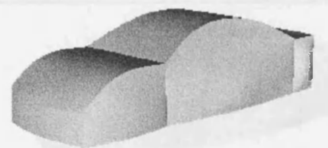


Figure 6.1 (a): The model of automotive body in the case base :Cb001

Cb002

FUNCTION MODEL

Function-1: Support chassis

Function-2: Provide space for passengers

Function-3: Stylise

Relationship ((Function-1, Function-2, and), (Function-2, Function-3, and), (Function-3, Function-1, and))

FEATURE MODEL

Behaviour feature

((Length 4262mm)

(Width 1751mm)

(Height 1408mm)

(Wheelbase 2725mm)

(Front track 1484mm)

(Rear track 1493mm))

Function feature

((Type Compact)

(Fuel consumption 9.7 ltr/100km)

(Top speed 201km/h)

(Acceleration 0-100km/h 11.1sec))

Structure feature

((Weight 1375kg))

STRUCTURE MODEL

Sub-Fractals: Engine compartment, passenger cabin, luggage compartment

:performance

(list :provide_space_for_the_engine :provide_space_for_the_passengers_and_luggage)

:G1 (list :minimise :volume :c)

:G2 (list :maximise :comfort :c)

:G3 (list :maximise :visualisation :c)

Assembly model

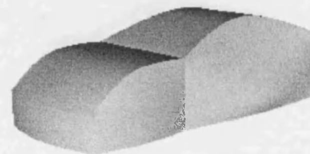


Figure 6.1 (b): The model of automotive body in the case base: Cb002

Cb003

FUNCTION MODEL

Function-1: Support chassis

Function-2: Provide space for passengers

Function-3: Stylise

Relationship ((Function-1, Function-2, and), (Function-2, Function-3, and), (Function-3, Function-1, and))

FEATURE MODEL

Behaviour feature

((Length 4775mm)

(Width 1800mm)

(Height 1435mm)

(Wheelbase 2830mm)

(Front track 1512mm)

(Rear track 1526mm))

Function feature

((Type Saloon)

(Fuel consumption 12.2 ltr/100km)

(Top speed 226km/h)

(Acceleration 0-100km/h 9.1sec))

Structure feature

((Weight 1570kg))

STRUCTURE MODEL

Sub-Fractals: Engine compartment, passenger cabin, luggage compartment

:performance (list :provide_space_for_the_engine :provide_space_for_the_passengers :provide_space_for_the_luggage)

:G1 (list :minimise :volume :c)

:G2 (list :maximise :comfort :c)

:G3 (list :maximise :visualisation :c)

Assembly model

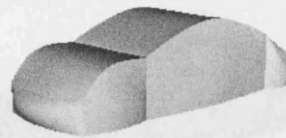


Figure 6.1 (c): The model of automotive body in the case base: Cb003

Cb004

FUNCTION MODEL

Function-1: Support chassis

Function-2: Provide space for passengers

Function-3: Stylise

Relationship ((Function-1, Function-2, and), (Function-2, Function-3, and), (Function-3, Function-1, and))

FEATURE MODEL

Behaviour feature

((Length 4100mm)

(Width 1700mm)

(Height 1400mm)

(Wheelbase 2750mm)

(Front track 1484mm)

(Rear track 1493mm))

Function feature

((Type Compact)

(Fuel consumption 13.7 ltr/100km)

(Top speed 210km/h)

(Acceleration 0-100km/h 11.5sec))

Structure feature

((Weight 1500kg))

STRUCTURE MODEL

Sub-Fractals: Engine compartment, passenger cabin, luggage compartment

:performance

(list :provide_space_for_the_engine :provide_space_for_the_passengers_and_luggage)

:G1 (list :minimise :volume :c)

:G2 (list :maximise :comfort :c)

:G3 (list :maximise :visualisation :c)

Assembly model

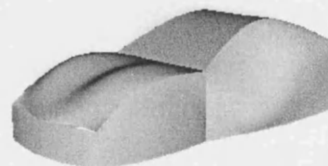


Figure 6.1 (d): The model of automotive body in the case base: Cb004

Cb005

FUNCTION MODEL

Function-1: Support chassis

Function-2: Provide space for passengers

Function-3: Stylise

Relationship ((Function-1, Function-2, and), (Function-2, Function-3, and), (Function-3, Function-1, and))

FEATURE MODEL

Behaviour feature

((Length 4569mm)

(Width 1853mm)

(Height 1674mm)

(Wheelbase 2795mm)

(Front track 1524mm)

(Rear track 1542mm))

Function feature

((Type Estate)

(Fuel consumption 39.2 ltr/100km)

(Top speed 198km/h)

(Acceleration 0-100km/h 8.3sec))

Structure feature

((Weight 1820kg))

STRUCTURE MODEL

Sub-Fractals: Engine compartment, passenger cabin, luggage compartment

:performance

(list :provide_space_for_the_engine :provide_space_for_the_passengers_and_luggage)

:G1 (list :minimise :volume :c)

:G2 (list :maximise :comfort :c)

:G3 (list :maximise :visualisation :c)

Assembly model

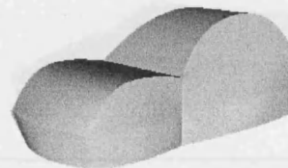


Figure 6.1 (e): The model of automotive body in the case base: Cb005

Cb006

FUNCTION MODEL

Function-1: Support chassis

Function-2: Provide space for passengers

Function-3: Stylise

Relationship ((Function-1, Function-2, and), (Function-2, Function-3, and), (Function-3, Function-1, and))

FEATURE MODEL

Behaviour feature

((Length 5500mm)

(Width 2000mm)

(Height 1492mm)

(Wheelbase 2990mm)

(Front track 1578mm)

(Rear track 1582mm))

Function feature

((Type Saloon)

(Fuel consumption 19.5 ltr/100km)

(Top speed 180km/h)

(Acceleration 0-100km/h 10.1sec))

Structure feature

((Weight 2000kg))

STRUCTURE MODEL

Sub-Fractals: Engine compartment, passenger cabin, luggage compartment

:performance (list :provide_space_for_the_engine :provide_space_for_the_passengers :provide_space_for_the_luggage)

:G1 (list :minimise :volume :c)

:G2 (list :maximise :comfort :c)

:G3 (list :maximise :visualisation :c)

Assembly model

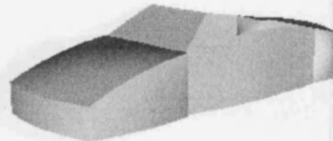


Figure 6.1 (f): The model of automotive body in the case base: Cb006

Cb007

FUNCTION MODEL

Function-1: Support chassis

Function-2: Provide space for passengers

Function-3: Stylise

Relationship ((Function-1, Function-2, and), (Function-2, Function-3, and), (Function-3, Function-1, and))

FEATURE MODEL

Behaviour feature

((Length 4871mm)

(Width 1855mm)

(Height 1372mm)

(Wheelbase 2781mm)

(Front track 1567mm)

(Rear track 1584mm))

Function feature

((Type Saloon)

(Fuel consumption 19.1 ltr/100km)

(Top speed 248km/h)

(Acceleration 0-100km/h 4.6sec))

Structure feature

((Weight 1785kg))

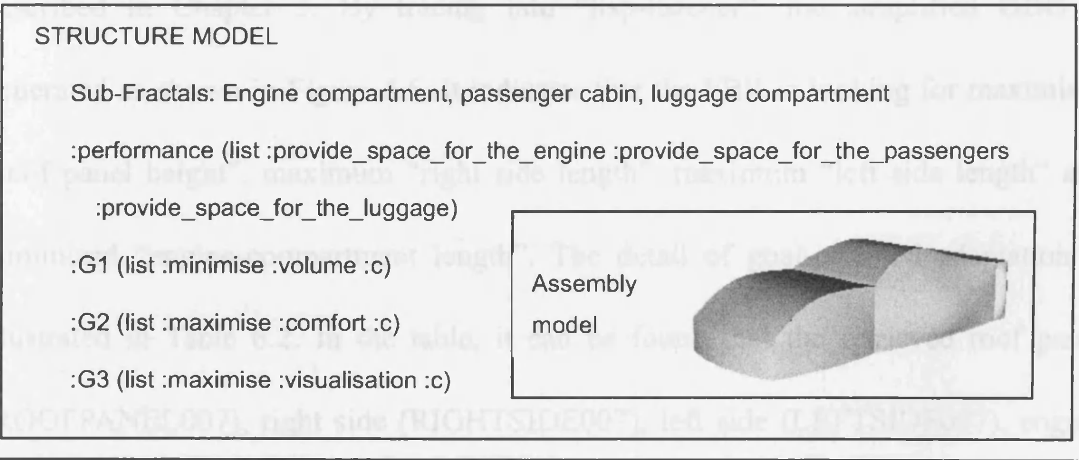


Figure 6.1 (g): The model of automotive body in the case base: Cb007

To query the case base, a set of requirements for the desired product can be formulated based on the requirements for an automotive body. The set of requirements for the design of this new automotive body are presented in Table 6.1. These requirements are implemented in an ICAD input model to query the FBR system. This is presented in Figure 6.2. Then the user is asked whether he wants to retrieve a case only or to retrieve and adapt a case, as shown in Figure 6.3. In this example, the user selects to retrieve and adapt. Using the similarity assessment method proposed in Chapter 4, the similarity between the input model and the cases in the case base is measured, and the most similar case is retrieved. Figure 6.4 shows the result of retrieval. By tracing into "lisp-listener", it can be found that it is the case "cb007" that has been retrieved. The system next asks the user to select the adaptation method. According to the initial requirement, the user selects the goal-oriented substitution as the method to be used in adaptation as shown in Figure 6.5. The method is as described in Chapter 5. By tracing into "lisp-listener", the simplified GDG is generated as shown in Figure 6.6. It indicates that the FBR is looking for maximised "roof panel height", maximum "right side length", maximum "left side length" and minimised "engine-compartment length". The detail of goal-oriented adaptation is illustrated in Table 6.2. In the table, it can be found that the retrieved roof panel (ROOFPANEL007), right side (RIGHTSIDE007), left side (LEFTSIDE007), engine compartment (EC007) have been substituted by ROOFPANEL001, RIGHTSIDE 001, RIGHTSIDE001 and EC001. This is because the latter fractals are able to maximise or minimise the particular attributes to realise the goals. As shown in the table, the

engine compartment EC001 directly substitutes EC007, while the others substitute only the structure without changing the dimensions. This is because the engine compartment is a fractal at the assembly level and the others are at the part level. Fractals at assembly level can be substituted as a whole, while the dimensional changes made at part levels may affect other parts. Finally, the result of adaptation is presented in Figure 6.7.

Feature	Requirement
Car-type	Saloon
Length	About 4700mm
Width	About 1700mm
Height	About 1300mm
Wheelbase	About 2900mm
Front track	About 1600mm
Rear track	About 1600mm
Fuel-consumption	About 20ltr/100km
Top-speed	220km/h
Length of passenger-cabin	2400mm
Width of passenger-cabin	1700mm
Height of passenger cabin	1300mm
Goal	"to maximise visualisation and comfort"

Table 6.1: A design specification for the automotive design

FUNCTIONS:	(LIST :SUPPORT-CHASIS :PROTECT-PASSENGER :ST! YLING)
FUNCTION-RELATIONSHIP:	(LIST :SUPPORT-CHASIS-AND-PROTECT-PASSENGER ! :PROTECT-PASSENGER-AND-STYLING :STYLING-AND-! SUPPORT-CHASIS)
LENGTH:	4700
WIDTH:	1700
HEIGHT:	1300
WHEELBASE:	2700
FRONTTRACK:	1600
REARTRACK:	1600
CAR-TYPE:	:SALOON
FUEL-CONSUMPTION:	20
OPTIONAL-FEATURES:	(LIST (LIST :PASSENGER-CABIN :LENGTH 2400) (! LIST :PASSENGER-CABIN :WIDTH 1700) (LIST :PA! SSENGER-CABIN :HEIGHT 1300))
WEIGHTS:	(LIST 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.! 5)
TOLERANCE-BAND:	(LIST 0.1 0.08 0.05 0.08 0.05 0.05 0.2)
WEIGHTS-OPTIONAL-FEATURES:	(LIST 0.5 0.5 0.5)
TOLERANCE-BAND-OPTIONAL-FEATURES:	(LIST 0.1 0.1 0.1)
W1:	1
W2:	0
W3:	0

Figure 6.2: Inputs to query the FBR system

Choice Attribute

Accept! Default! Cancel!

Cbr

Do you need to retrieve or retrieve and adapt a design? (choose one)

Select Accept! or Default! to use default "Retrieve"

Retrieve

Retrieve And Adapt

Figure 6.3: Selection of retrieval or adaptation

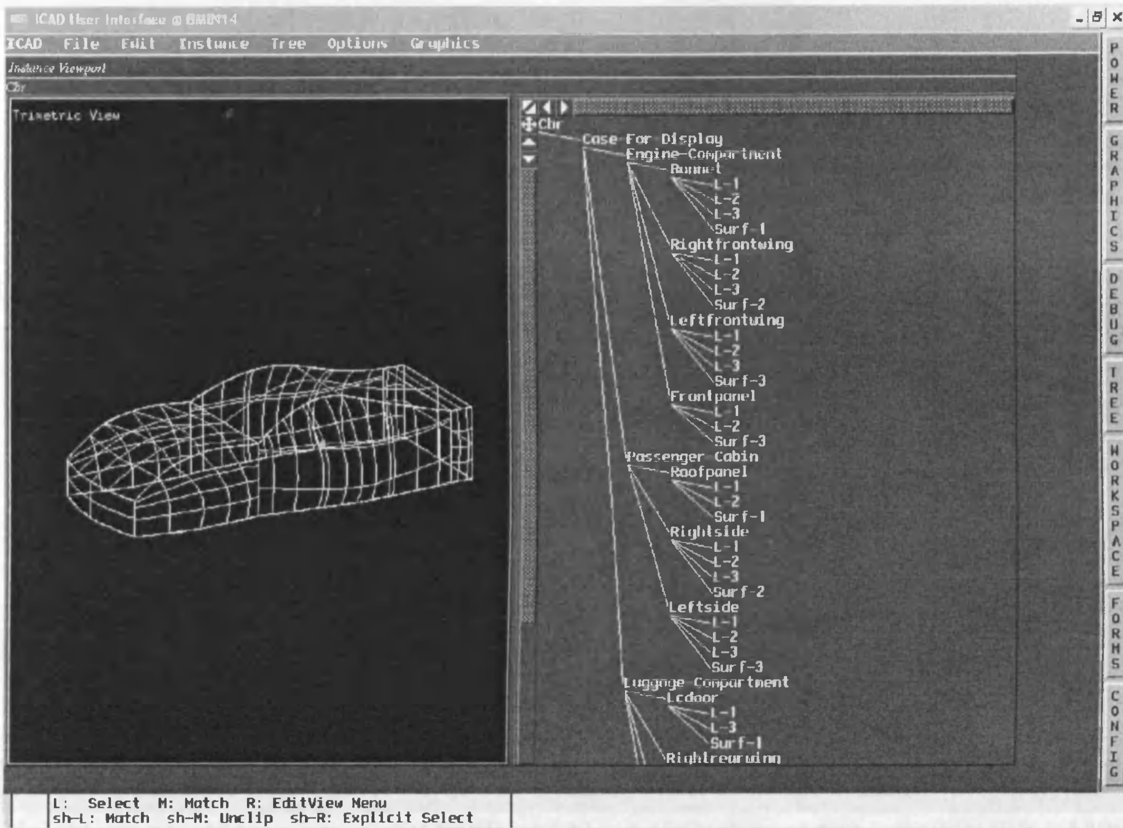


Figure 6.4: The result of retrieval

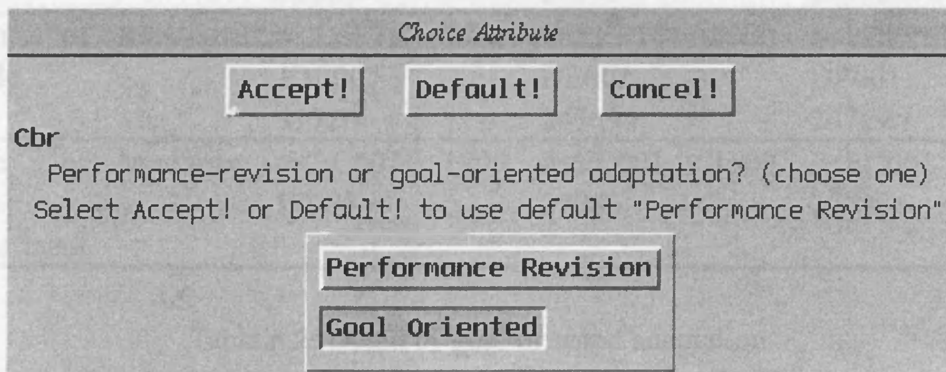


Figure 6.5: Selection of adaptation methods

```
((:ROOFPANEL :MAXIMISE :HEIGHT :I :G2)

(:RIGHTSIDE :MAXIMISE :LENGTH :I :G2)

 (:LEFTSIDE :MAXIMISE :LENGTH :I :G2))

 (:ENGINE-COMPARTMENT :MINIMSE :LENGTH :I :G3))
```

Figure 6.6: Tracing simplified GDG in lisp-listener

Attributes	indicator	Retrieved case	Substituting case	Adapted case
Height of roof panel	Maximise	#<ROOFPANEL007 31408> 1372	#<ROOFPANEL001 24333> 1492	#<ROOFPANEL001 24333> 1372
Length of right side	Maximise	#<RIGHTSIDE007 31567> 2781	#<RIGHTSIDE001 24552> 3000	#<RIGHTSIDE001 24552> 2781
Length of left side	Maximise	#<LEFTSIDE007 31726> 2781	#<LEFTSIDE001 24769> 3000	#<LEFTSIDE001 24769> 2781
Length of engine compartment	Minimise	#<EC007 27085> 1790	#<EC001 27085> 1500	#<EC001 27085> 1500

Table 6.2: Detail of goal-oriented adaptation

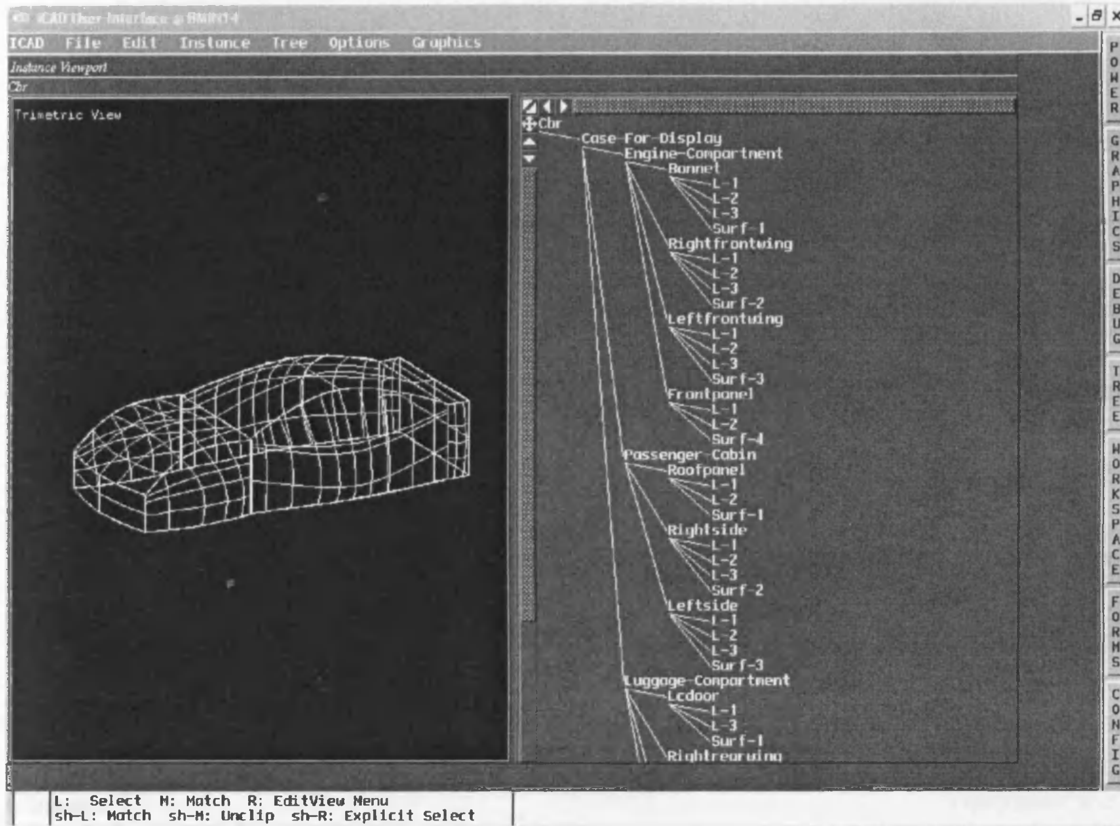


Figure 6.7: The result of adaptation

6.4 Discussion

In this chapter, it was illustrated how the approach presented in this thesis could be applied to support the design of a real-world product. The knowledge that was required to design the product was presented, which demonstrated how this can be formalised using the fractal-based modelling approach presented in Chapter 3. A formal design case for automotive body design was presented. In this case study, each model is comprised of approximately ten basic fractal units, fifty elements, ninety attributes and fifty relationships. The entire case base contains about eighty basic fractal units, three hundred elements, four hundred attributes and three hundred relationships. It was also demonstrated how to use this knowledge to guide the process of design. Measuring the similarity between an input model and existing cases was discussed using the method presented in Chapter 4. A goal-oriented substitution method was then employed to adapt the retrieved case study product, as presented in Chapter 5.

A comparison between the requirements and the results is given in Table 6.3. It can be seen that the initial design requirements have been fulfilled. The conclusion can thus be drawn that FBR has successfully retrieved and adapted the automotive body design.

Feature	Requirement	Retrieved case	Substituting case	Adapted case
Car-type	Saloon	Saloon		Saloon
Length	About 4700mm	4871mm		4581mm
Width	About 1700mm	1855mm		1902mm
Height	About 1300mm	1372mm		1372mm
Wheelbase	About 2900mm	2781mm		2781mm
Front track	About 1600mm	1567mm		1567mm
Rear track	About 1600mm	1584mm		1584mm
Fuel-consumption	About 20ltr/100km	19.1ltr/100km		19.1ltr/100km
Top-speed	220km/h	248km/h		248km/h
Length of passenger-cabin	2400mm	2781mm		2781mm
Width of passenger-cabin	1700mm	1855mm		1855mm
Height of passenger cabin	1300mm	1372mm		1372mm
Height of roof panel		1372mm	1492mm	1372mm
Length of right side		2781mm	3000mm	2781mm
Length of left side		2781mm	3000mm	2781mm
Length of engine compartment		1790mm	1500mm	1500mm

Table 6.3: Comparison of the design specification and result of adaptation

6.5 Summary

A case study in automotive body design was given to demonstrate the successful implementation of FBR. The chapter started with a short review of automotive body design in Section 6.2. A case study in automotive body design was presented in Section 6.3 and the results were discussed in Section 6.4.

Chapter 7

Conclusions

7.1 Preliminaries

This chapter presents the conclusions of this work, outlines the main contributions of the research, and makes recommendations for further studies.

7.2 Conclusions

In Chapter 1, the objectives of this research were presented. These were as follows:

1) To identify and externalise design knowledge using a fractal-like model.
2) To understand the role of design knowledge in conceptual design.
3) To use design knowledge as a guide for every stage of concept development.
4) To provide a framework for supporting conceptual design, using the techniques of case-based reasoning and fractal theory, for reasoning about design and development of computer-based design aids.

This section discusses how the research presented in this thesis achieved these

objectives.

Designers have limited time to build up experience and are, in any event, unlikely to become experts in all relevant fields. Design knowledge needs to be captured, stored and reused. In Chapter 3, to address objectives 1 and 2, a design modelling technique was presented upon which the research reported in this thesis was based. The model is in the form of attributed graphs containing design knowledge about function, feature, structure, performance and goal. It can represent a design from an abstract to a physical level. Knowledge related to both the design objects and to the design process is represented. Fractals are introduced to represent the design knowledge related to the design process. The presented model has the fractal characteristics of self-similarity, self-organisation, goal-orientation and dynamism. These characteristics play important roles in the process of case-based conceptual design. It can be concluded that design knowledge has been successfully externalised using a fractal-like model.

To address objective 3, this research incorporates design knowledge to assess design model similarity as described in Chapter 4. Previous research had the problems that product information was insufficiently incorporated to compare similarity of complex designs and that complicated geometric comparisons were conducted without taking account of the criteria of domain specific technical knowledge. Design is a stage in which human knowledge is involved. It is the knowledge rather than the shape model that should be compared. In the proposed method, design knowledge such as function,

feature and structure is extracted and used for the comparison of designs. To address objective 3, this research also incorporates design knowledge, such as performance and goals, to guide the process of design adaptation as described in Chapter 5. Performance revision adapts a design by substituting fractals according to the identified performance; goal-oriented adaptation provides an efficient way for adaptation by using the goals of a design problem to guide the process of adaptation. Therefore, it can be seen that design knowledge has been successfully used as a guide for every stage of concept development.

In addition to fractal theory, the thesis has also shown how CBR can be used to support the human designer during conceptual design. In Chapter 4, similarity assessment in case retrieval was introduced. Similarity of design models is measured by concurrently applying a feature-based similarity measure and a structure-based similarity measure. A weighting method is then adopted to assess the results. In Chapter 5, adaptation strategies which include performance revision and goal-oriented substitution with the help of fractal characteristics were discussed. The integration of the case-based reasoning and fractal theory into a framework for supporting conceptual design has been presented to address objective 4. The approach was then validated using a case study on the automotive body design problem in Chapter 6. The goal to provide computer support for the conceptual design process using CBR and fractal theory has, therefore, been achieved.

7.3 Contributions

The research presented in this thesis has contributed to the area of design modelling and knowledge representation. A fractal-like design modelling technique has been proposed, which is able to represent various aspects of design knowledge from abstract to physical levels and from knowledge related to design objects to knowledge related to design processes.

The research also contributed to the area of intelligent design. The research, for the first time, has applied fractal theory to conceptual design. The important fractal characteristics play very important roles from modelling to adaptation. The research has achieved the automation of the majority of the tasks involved in conceptual design.

Another important contribution has been in the area of case-based design. In particular, case retrieval has been conducted by measuring similarity using various aspects of design knowledge; case adaptation has been treated as a performance revision process and as a goal-oriented process.

Finally, this work has added a contribution to the area of similarity measure. A combinative feature-based and structure-based similarity measure method has been presented.

7.4 Recommendations

This section presents a number of recommendations for further research. The possible topics are as follows:

- Collaborative fractal-based design. In a broader scope, fractals can be regarded as an agent-based structure and the fractal-based design can be conducted in a distributed environment. In industry, because of the distributed nature of the design teams, a product is designed through the collective and joint efforts of many designers. There is a need to develop a tool in order to support collaborative design. Fractals will play a more important role in a dynamic collaborative environment.

- Automation of creative design. Since this research mainly focuses on routine design, it involves only knowledge related to this particular type. Creative design, which constitutes the remaining challenge in the area of design automation, also requires a supporting tool. One possible avenue for further study would be the development of tools which assist in modelling the knowledge of product evolution and which can then be used to generate innovative products.

- Integration of a CAD model. The geometric model used to query the case base is in the form of a graph model. A real CAD model, e.g. STEP, as a widely used

geometric format, might be more effectively used to query the case base for a similar shape or geometric model.

Appendix A

An overview of the ICAD system

This appendix outlines the ICAD system and some of the features that make it particularly suitable for the work presented in this thesis.

ICAD is a knowledge-based engineering environment that can be used for developing "intelligent" design systems. It consists of an Emacs editor, in which programs are written, and the ICAD browser. The programming language used is the ICAD Design Language (IDL), based on Common Lisp.

ICAD uses an engineering methodology known as generative technology to encapsulate the essential information required to design, analyse and manufacture a product. This information is stored in what is known as an ICAD Product Model. The product model represents the engineering intent behind the physical product. It can store information such as product information (attributes of the physical product such as geometry, material type, and functional constraints), process information, drawings and reports. This makes ICAD different from conventional CAD systems which produce models that contain mostly geometric information only. An ICAD product model contains all of the information required by the design. Once an ICAD product model is created, an engineer can use it to generate, evaluate, or configure new designs simply by changing the input to the model.

Object-oriented programming: The fundamental building block in IDL is a "Defpart" (DEFinition of a PART). Basically, a "Defpart" represents a class of parts. It is composed of a defpart name, a mixin list (which allows the defpart to be built from other defparts), and features (which describe the rules and part structure). Specific designs are generated by providing different values for the input parameters. Design instances are generated by giving specific input values to a defpart. The process of generating a design instance from a defpart is known as instantiation. By object-oriented programming, the design of complex systems is modularised by grouping similar parts into the same class; a part can be characterised by using different kinds of attributes, not just geometric attributes.

Part-whole representation: Complex parts and systems can be simplified by representing them within a tree-like structure. This "product-structure-tree", as is called in ICAD, allows the division of complex assemblies into sub-assemblies. Each sub-assembly can be further divided. This more accurately reflects the way that design and manufacturing engineers think about design.

Attributes: Attributes provide a way to attach information to a part design. Any kind of information can be represented in an attribute, including part geometry, engineering information, process information, etc. These attributes can be written in many ways, including arithmetic expressions, conditional rules, relationships to other attributes, database lookups, connections to outside computer programs, and user inputs.

Demand-driven evaluation: The rules that are embedded in a product model are evaluated only when they are required. For example, if the system is required to draw

the geometry of a model, only those rules required to draw the geometry are evaluated. The system will not evaluate rules that determine the weight or cost of that model when drawing the geometry. Once a rule has been calculated, its value is cached until it is needed again. The entire assembly structure is demand-driven. Sub-assemblies are not computed until they are needed. This means the designer can efficiently work on portions of an overall assembly without the system needing to compute the entire assembly. Demand-driven evaluation has such advantages: the system functions more efficiently; the IDL is order-independent; working on small pieces of a large model is easier; traditional programming problems such as memory management and flow of control are avoided.

Appendix B

Code for case base

This appendix gives the code for case representation. For simplicity, only the code for one case is given here. The case base used in this research consists of a number of other cases like this. They are used in conjunction with the other codes listed in Appendix C, D, E, and F. This code can be used to illustrate the fractal-like design modelling technique discussed in Chapter 3. Part of this code is used for the illustrative example of Chapter 4, and the code as a whole is used for the example and case study of Chapter 5. The code is written in IDL, which is based on Common Lisp.

```
(defpart case-base ()
  :parts
  ((cab :type cb001) (cab-2 :type cb002)   (cab-3 :type cb003) (cab-
4 :type cb004) (cab-5 :type cb005) (cab-6 :type cb006) (cab-7 :type
cb007)
   ))

;;;;;;;;;;;;;;;;;;;;;;;; CASE cb001 ;;;;;;;;;;;;;;;;;;;

(defpart cb001 (box)
  :attributes
  (:functions (list :support-chasis :protect-passenger :stylise)
   :function-relationship (list :support-chasis-and-protect-
passenger :protect-passenger-and-stylise :stylise-and-support-chasis)
   :function-model (append (the :functions) (the :function-
relationship))
   :width (max (the :engine-compartment :width) (the :passenger-
cabin :width) (the :luggage-compartment :width))
   :length (+ (the :engine-compartment :length) (the :passenger-
cabin :length) (the :luggage-compartment :length))
   :height (max (the :engine-compartment :height) (the :passenger-
cabin :height) (the :luggage-compartment :height))
   :car-type :saloon
   :wheelbase 2990
   :fronttrack 1578
```

```
   :reartrack 1582
   :weight 1865
   :fuel-consumption 15.5
   :top-speed 237
   :acceleration 8.1
   :behaviour-feature
          (list (the :length) (the :width) (the :height)
(the :wheelbase) (the :fronttrack) (the :reartrack))
   :function-feature (list (the :car-type) (the :fuel-consumption)
(the :top-speed) (the :acceleration))
   :structure-feature (list (the :weight))
   :features (append (the :behaviour-feature) (the :function-feature)
(the :structure-feature))
   :performance
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :performance))
          (cc (collect! (defaulting bb () ))))
     ((return-when empty? cc)))
   :corporate-goals (list (the :G1) (the :G2) (the :G3))
   :G1 (list :minimise :volume :c)
   :G2 (list :maximise :comfort :c)
   :G3 (list :maximise :visualisation :c)
   :sub-goals
   (let-streams
         ((aa (in-tree self))
          (bb (defaulting (the-object aa :goals)))
          (dd (the-object aa :name-for-display))
          (ee (if (not (equal bb :not-applicable))
                (append bb (list (list dd)))
              'nil))
          (cc (collect! ee )))
     ((return-when empty? (remove 'nil cc)))     )

   :reform-sg
   (let-streams
       ((a (in (the :sub-goals)))
      (b (lastcar a))
      (c (remove b a))
      (e (let-streams
             ((aa (in c))
           (bb (append b aa))
           (cc (collect! bb)))
          ((return-when empty? cc))))
      (d (collect! e)))
     ((return-when empty? d)))

   :reform-sg2
   (let-streams
       ((a (in (the :reform-sg)))
      (b (stream-append a)))
     ((return-when empty? b)))



   :find-individual-goals
   (let-streams
       ((aa (in (the :reform-sg2)))
      (bb (nth 3 aa))
      (individual? (equal bb :i))
      (cc (collect-if! individual?  aa 'nil)))
```

```
      ((return-when empty?  (remove 'nil cc))))


    :sub-relations
    (let-streams
          ((aa (in (the :children)))
           (bb (the-object aa :name-for-display))
           (cc (collect! bb)))
      ((return-when empty? cc)))
    :sub-cases
    (let-streams
          ((aa (in (the :children)))
           (bb (the-object aa :type))
           (cc (collect! bb)))
      ((return-when empty? cc)))


    )

   :report-attributes
   (:vrml-out
   (write-vrml-file (list self)
   "f:/cb001.wrl"
   :specified-planarity 0.1
   :specified-linearity 0.1
   :camera-on? t))

   :parts
   ((engine-compartment
     :type ec001
     :position (:bottom 0.0))
    (passenger-cabin
     :type pc001
     :position (:front (:from (the :engine-compartment) (the :engine-
compartment :length))))
    (luggage-compartment
     :type lc001
     :position (:front (:from (the :passenger-cabin) (the :passenger-
cabin :length)) :bottom 0.0 ))
    ))


(defpart ec001 (box)
  :attributes
  (:width 1902
   :length 1500
   :height 1000
   :performance (list :provide_space_for_the_engine)
   :goals (list (the :G1-G1) (the :G3-G1))
   :G1-G1 (list :minimise :volume :c :G1)
   :G3-G1 (list :minimise :length :i :G3)

   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :name-for-display))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))

  :report-attributes
```

```
  (:vrml-out
  (write-vrml-file (list self)
  "f:/ec001.wrl"
  :specified-planarity 0.1
  :specified-linearity 0.1
  :camera-on? t))

  :parts
  ((bonnet :type bonnet001)
   (rightfrontwing :type rightfrontwing001)
   (leftfrontwing :type leftfrontwing001)
   (frontpanel  :type frontpanel001)))


(defpart bonnet001 (box)
   :attributes
  (:s-1-points-1 (list (the (:edge-center :right :front))
                   (translate (the (:edge-center :right :top)) :down
100 :front 250)
                   (the (:vertex :top :right :rear))
              )
   :s-1-points-2 (list (translate (the (:face-center :front)) :front
150)
                   (translate (the (:face-center :top)) :down
50 :front 200)
                   (the (:edge-center :rear :top))
              )
   :s-1-points-3 (list (the (:edge-center :left :front))
                   (translate (the (:edge-center :left :top)) :down
100 :front 250)
                   (the (:vertex :top :left :rear))
                   )
   :display-controls (merge-display-controls
                  '(:color :magenta))
   :performance (list :provide_top_cover_for_the_engine_compartment)
   :goals (list (the :G1-G1-G1))
   :G1-G1-G1 (list :minimise :length :i :G1-G1)
   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :type))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))

  :parts
  ((l-1 :type fitted-wire
      :points (the :s-1-points-1))
   (l-2 :type fitted-wire
      :points (the :s-1-points-2))
   (l-3 :type fitted-wire
      :points (the :s-1-points-3))
   (surf-1 :type lofted-sheet
         :wires (list (the :l-1) (the :l-2) (the :l-3)))

   ))

(defpart rightfrontwing001 (box)
   :attributes
  (:s-2-points-1 (list (the (:edge-center :right :front))
```

```
                    (translate (the (:edge-center :right :top)) :down
100 :front 250)
                    (the (:vertex :top :right :rear)))
   :s-2-points-2 (list (translate (the (:edge-
center :right :front)) :down 100)
                    (translate (the (:face-center :right)) :right 100)
                    (the (:edge-center :right :rear))
               )
   :s-2-points-3 (list (the (:vertex :bottom :front :right))
                    (translate (the (:edge-
center :right :bottom)) :right 100)
                    (the (:vertex :bottom :right :rear)))
   :display-controls (merge-display-controls
                   '(:color :magenta))
   :performance
(list :provide_right_side_cover_for_the_engine_compartment)
   :goals (list (the :G1-G1-G2))
   :G1-G1-G2 (list :minimise :height :i :G1-G1)
   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :type))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))

   :parts
   ((l-1 :type fitted-wire
      :points (the :s-2-points-1))
   (l-2 :type fitted-wire
      :points (the :s-2-points-2))
   (l-3 :type fitted-wire
      :points (the :s-2-points-3))
    (surf-2 :type lofted-sheet
         :wires (list (the :l-1) (the :l-2) (the :l-3)))

   ))

(defpart leftfrontwing001 (box)
   :attributes
  (:s-3-points-1 (list (the (:edge-center :left :front))
                    (translate (the (:edge-center :left :top)) :down
100 :front 250)
                    (the (:vertex :top :left :rear))
                    )
   :s-3-points-2 (list (translate (the (:edge-
center :left :front)) :down 100)
                    (translate (the (:face-center :left)) :left 100)
                    (the (:edge-center :left :rear))
                    )
   :s-3-points-3 (list (the (:vertex :bottom :front :left))
                    (translate (the (:edge-
center :left :bottom))  :right 100)
                    (the (:vertex :bottom :left :rear))
                    )
   :display-controls (merge-display-controls
                   '(:color :magenta))
   :performance
(list :provide_left_side_cover_for_the_engine_compartment)
   :goals (list (the :G1-G1-G2))
   :G1-G1-G2 (list :minimise :height :i :G1-G1)
```

```
    :sub-relations
    (let-streams
          ((aa (in (the :children)))
           (bb (the-object aa :type))
           (dd (list (the :name-for-display) bb))
           (cc (collect! dd)))
        ((return-when empty? cc))))
    :parts
   ((l-1 :type fitted-wire
       :points (the :s-3-points-1))
    (l-2 :type fitted-wire
       :points (the :s-3-points-2))
    (l-3 :type fitted-wire
       :points (the :s-3-points-3))
     (surf-3 :type lofted-sheet
          :wires (list (the :l-1) (the :l-2) (the :l-3)))
    ))

 (defpart frontpanel001 (box)
   :attributes
   (:s-4-points-1 (list (the (:edge-center :right :front))
                    (translate (the (:face-center :front)) :front 150)
                    (the (:edge-center :left :front))
                    )
    :s-4-points-2 (list (the (:vertex :bottom :front :right))
                    (translate (the (:edge-
center :bottom :front )) :front 150)
                    (the (:vertex :bottom :front :left))
                    )
    :display-controls (merge-display-controls '(:color :magenta))
    :performance (list :provide_front_cover_for_the_engine_compartment)
    :goals (list (the :G1-G1-G4))
    :G1-G1-G4 (list :minimise :width :i :G1-G1)
    :sub-relations
    (let-streams
          ((aa (in (the :children)))
           (bb (the-object aa :type))
           (dd (list (the :name-for-display) bb))
           (cc (collect! dd)))
        ((return-when empty? cc))))
   :parts
   ((l-1 :type fitted-wire
       :points (the :s-4-points-1))
    (l-2 :type fitted-wire
       :points (the :s-4-points-2))
    (surf-4 :type lofted-sheet
          :wires (list (the :l-1) (the :l-2) ))
    ))

 (defpart pc001 (box)
   :attributes
   (:width 1902
    :length 3000
    :height 1492
    :performance (list :provide_space_for_the_passengers)
    :goals (list (the :G1-G2) (the :G2-G1))
    :G1-G2 (list :minimise :volume :c :G1)
    :G2-G1 (list :maximise :volume :c :G2)
    :sub-relations
    (let-streams
          ((aa (in (the :children)))
```

```
          (bb (the-object aa :name-for-display))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))

  :report-attributes
  (:vrml-out
  (write-vrml-file (list self)
  "f:/pc001.wrl"
  :specified-planarity 0.1
  :specified-linearity 0.1
  :camera-on? t)
   )

  :parts
  (
   (roofpanel   :type roofpanel001)
   (rightside :type rightside001)
   (leftside :type leftside001)
   ))


(defpart roofpanel001 (box)
  :attributes
  (:s-1-points-1 (list (translate (the
(:vertex :front :bottom :right)) :up (the :engine-
compartment :height))
                    (translate (the
(:vertex :top :front :right)) :down 100 :rear 800 :left 200)
                    (translate (the (:vertex :top :rear :right)) :down
100 :front 800 :left 200)
                    (translate (the
(:vertex :rear :bottom :right)) :up (the :luggage-
compartment :height))
              )
   :s-1-points-2 (list (translate (the
(:vertex :front :bottom :left)) :up (the :engine-compartment :height))
                    (translate (the (:vertex :top :front :left)) :down
100 :rear 800 :right 200)
                    (translate (the (:vertex :top :rear :left)) :down
100 :front 800 :right 200)
                    (translate (the (:vertex :rear :bottom :left)) :up
(the :luggage-compartment :height))
                    )
   :display-controls (merge-display-controls '(:color :magenta))
   :performance (list :provide_top_cover_for_the_passenger_cabin)
   :goals (list (the :G1-G2-G1)(the :G1-G2-G2) (the :G2-G1-G1))
   :G1-G2-G1 (list :minimise :height :i :G1)
   :G1-G2-G2 (list :minimise :width :i :G1)
   :G2-G1-G1 (list :maximise :height :i :G2)
   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :type))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))
  :parts
  ((l-1 :type fitted-wire
      :points (the :s-1-points-1))
   (l-2 :type fitted-wire
```

```
      :points (the :s-1-points-2))
   (surf-1 :type lofted-sheet
          :wires (list (the :l-1) (the :l-2) ))
))

(defpart rightside001 (box)
  :attributes
  (:s-2-points-1 (list (translate (the
(:vertex :front :bottom :right)) :up (the :engine-
compartment :height))
                     (translate (the
(:vertex :top :front :right)) :down 100 :rear 800 :left 200)
                     (translate (the (:vertex :top :rear :right)) :down
100 :front 800 :left 200)
                     (translate (the
(:vertex :rear :bottom :right)) :up (the :luggage-
compartment :height))
               )
   :s-2-points-2 (list (translate (the
(:vertex :front :bottom :right)) :up (- (the :engine-
compartment :height) 100))
                     (translate (the (:face-center :right)) :right
100 :front 500)
                     (translate (the (:face-center :right)) :right
100 :rear 500)
                     (translate (the
(:vertex :rear :bottom :right)) :up (- (the :luggage-
compartment :height) 100))
               )
   :s-2-points-3 (list (the (:vertex :bottom :right :front))
                     (translate (the (:edge-
center :right :bottom)) :right 100 :front 500)
                     (translate (the (:edge-
center :right :bottom)) :right 100 :rear 500)
                     (the (:vertex :bottom :right :rear))
                     )
   :display-controls (merge-display-controls '(:color :magenta))
   :performance
(list :provide_right_side_cover_for_the_passenger_cabin)
   :goals (list (the :G1-G2-G3) (the :G2-G1-G2))
   :G1-G2-G3 (list :minimise :length :i :G1)
   :G2-G1-G2 (list :maximise :length :i :G2)
   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :type))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))
  :parts
  ((l-1 :type fitted-wire
      :points (the :s-2-points-1))
   (l-2 :type fitted-wire
      :points (the :s-2-points-2))
   (l-3 :type fitted-wire
      :points (the :s-2-points-3))
   (surf-2 :type lofted-sheet
         :wires (list (the :l-1) (the :l-2) (the :l-3) ))
))

(defpart leftside001 (box)
```

```
  :attributes
  (:s-3-points-1 (list (translate (the
(:vertex :front :bottom :left)) :up (the :engine-compartment :height))
                    (translate (the (:vertex :top :front :left)) :down
100 :rear 800 :right 200)
                    (translate (the (:vertex :top :rear :left)) :down
100 :front 800 :right 200)
                    (translate (the (:vertex :rear :bottom :left)) :up
(the :luggage-compartment :height))
                    )
   :s-3-points-2 (list (translate (the
(:vertex :front :bottom :left)) :up (- (the :engine-
compartment :height) 100))
                    (translate (the (:face-center :left)) :top
400 :left 100 :front 500)
                    (translate (the (:face-center :left)) :top
400 :left 100 :rear 500)
                    (translate (the (:vertex :rear :bottom :left)) :up
(- (the :luggage-compartment :height) 100))
               )
   :s-3-points-3 (list (the (:vertex :bottom :left :front))
                    (translate (the (:edge-
center :left :bottom)) :left 100 :front 500)
                    (translate (the (:edge-
center :left :bottom)) :left 100 :rear 500)
                    (the (:vertex :bottom :left :rear))
                    )
   :display-controls (merge-display-controls '(:color :magenta))
   :performance
(list :provide_left_side_cover_for_the_passenger_cabin)
   :goals (list (the :G1-G2-G4) (the :G2-G1-G3))
   :G1-G2-G4 (list :minimise :length :i :G1)
   :G2-G1-G3 (list :maximise :length :i :G2)
   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :type))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))
  :parts
  ((l-1 :type fitted-wire
      :points (the :s-3-points-1))
   (l-2 :type fitted-wire
      :points (the :s-3-points-2))
   (l-3 :type fitted-wire
      :points (the :s-3-points-3))
   (surf-3 :type lofted-sheet
         :wires (list (the :l-1) (the :l-2) (the :l-3) ))
))

(defpart lc001 (box)
  :attributes
  (:width 1900
   :length 529
   :height 1000
   :performance (list :provide_space_for_the_luggage)
   :goals (list (the :G1-G3))
   :G1-G3 (list :minimise :volume :c :G1)
   :sub-relations
   (let-streams
```

```
          ((aa (in (the :children)))
           (bb (the-object aa :name-for-display))
           (dd (list (the :name-for-display) bb))
           (cc (collect! dd)))
        ((return-when empty? cc))))

   :report-attributes
  (:vrml-out
  (write-vrml-file (list self)
  "f:/lc001.wrl"
  :specified-planarity 0.1
  :specified-linearity 0.1
  :camera-on? t)
  )

  :parts
  ((lcdoor :type lcdoor001)
   (rightrearwing :type rightrearwing001)
   (leftrearwing :type leftrearwing001)
   (rearpanel :type rearpanel001)
   ))

(defpart lcdoor001 (box)
  :attributes
  (:s-1-points-1 (list (the (:vertex :top :right :front))
                   (the (:vertex :top :right :rear)))
   :s-1-points-2 (list (the (:edge-center :front :top))
                   (translate (the (:edge-center :rear :top)) :rear
150))
   :s-1-points-3 (list (the (:vertex :top :left :front))
                   (the (:vertex :top :left :rear)))
   :display-controls (merge-display-controls '(:color :magenta))
   :performance (list :provide_top_cover_for_the_luggage_compartment)
   :goals (list (the :G1-G3-G1))
   :G1-G3-G1 (list :minimise :length :i :G1)
   :sub-relations
   (let-streams
          ((aa (in (the :children)))
           (bb (the-object aa :type))
           (dd (list (the :name-for-display) bb))
           (cc (collect! dd)))
        ((return-when empty? cc))))
  :parts
  ((l-1 :type fitted-wire
      :points (the :s-1-points-1))
   (l-2 :type fitted-wire
      :points (the :s-1-points-2))
   (l-3 :type fitted-wire
      :points (the :s-1-points-3))
   (surf-1 :type lofted-sheet
         :wires (list (the :l-1) (the :l-2) (the :l-3) ))
))

(defpart rightrearwing001 (box)
  :attributes
  (:s-2-points-1 (list (the (:vertex :top :right :front))
                   (the (:edge-center :right :top))
                   (the (:vertex :top :right :rear))
              )
   :s-2-points-2 (list (the (:edge-center :right :front))
                   (translate (the (:face-center :right)) :right 100)
```

```
                  (the (:edge-center :right :rear))
              )
   :s-2-points-3 (list (the (:vertex :bottom :front :right))
                  (the (:edge-center :right :bottom))
                  (the (:vertex :bottom :right :rear))
                  )
   :display-controls (merge-display-controls '(:color :magenta))
   :performance
(list :provide_right_side_cover_for_the_luggage_compartment)
   :goals (list (the :G1-G3-G2))
   :G1-G3-G2 (list :minimise :height :i :G1)
   :sub-relations
   (let-streams
        ((aa (in (the :children)))
         (bb (the-object aa :type))
         (dd (list (the :name-for-display) bb))
         (cc (collect! dd)))
      ((return-when empty? cc))))
  :parts
  ((l-1 :type fitted-wire
      :points (the :s-2-points-1))
   (l-2 :type fitted-wire
      :points (the :s-2-points-2))
   (l-3 :type fitted-wire
      :points (the :s-2-points-3))
   (surf-2 :type lofted-sheet
        :wires (list (the :l-1) (the :l-2) (the :l-3) ))
               ))

(defpart leftrearwing001 (box)
  :attributes
  (:s-3-points-1 (list (the (:vertex :top :left :front))
                  (the (:edge-center :left :top))
                  (the (:vertex :top :left :rear))
                  )
   :s-3-points-2 (list (the (:edge-center :left :front))
                  (translate (the (:face-center :left)) :left 100)
                  (the (:edge-center :left :rear))
              )
   :s-3-points-3 (list (the (:vertex :bottom :front :left))
                  (the (:edge-center :left :bottom))
                  (the (:vertex :bottom :left :rear))
                  )
   :display-controls (merge-display-controls '(:color :magenta))
   :performance
(list :provide_left_side_cover_for_the_luggage_compartment)
   :goals (list (the :G1-G3-G3))
   :G1-G3-G3 (list :minimise :height :i :G1)
   :sub-relations
   (let-streams
        ((aa (in (the :children)))
         (bb (the-object aa :type))
         (dd (list (the :name-for-display) bb))
         (cc (collect! dd)))
      ((return-when empty? cc))))
  :parts
  ((l-1 :type fitted-wire
      :points (the :s-3-points-1))
   (l-2 :type fitted-wire
      :points (the :s-3-points-2))
   (l-3 :type fitted-wire
```

```
      :points (the :s-3-points-3))
   (surf-3 :type lofted-sheet
         :wires (list (the :l-1) (the :l-2) (the :l-3) ))
))

(defpart rearpanel001 (box)
  :attributes
  (:s-4-points-1 (list (the (:vertex :right :rear :top))
                  (translate  (the (:edge-center :rear :top)) :rear
150)
                  (the (:vertex :left :rear :top))
              )
   :s-4-points-2 (list (the (:vertex :bottom :right :rear))
                  (translate (the (:edge-
center :bottom :rear )) :rear 150)
                  (the (:vertex :bottom :left :rear))
                  )
   :display-controls (merge-display-controls '(:color :magenta))
   :performance (list :provide_back_cover_for_the_luggage_compartment)
   :goals (list (the :G1-G3-G4))
   :G1-G3-G4 (list :minimise :width :i :G1)
   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :type))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))

  :parts
  ((l-1 :type fitted-wire
      :points (the :s-4-points-1))
   (l-2 :type fitted-wire
      :points (the :s-4-points-2))
   (surf-4 :type lofted-sheet
         :wires (list (the :l-1) (the :l-2) ))
))
```

Appendix C

Code for similarity measure and case retrieval

This appendix shows the ICAD code for the approach of graph-based similarity measure and case retrieval presented in Chapter 4. It is used in conjunction with the code of case base, which is partly presented in Appendix B.

```
(defpart case-retrieval (box)
 :choice-attributes
  (:w1
   (:prompt "Enter weight for feature-based similarity measure"
    :domain (:number :from 0 :to 1)
    :default 0.4)
   :w2
   (:prompt "Enter weight for structure-based similarity measure on
function"
    :domain (:number :from 0 :to 1)
    :default 0.3)
   :w3
   (:prompt "Enter weight for structure-based similarity measure on
structure"
    :domain (:number :from 0 :to 1)
    :default 0.3)
   :weights
   (:prompt "Enter weights for length,width, height, wheelbase,
fronttrack, reartrack, car-type, fuel-consumption, top-speed,
acceleration and weight"
    :default (list 0.5 0.5 0.2 0.2 0.2 0.2 0.5 0.3 0.3 0.3 0.3))
   :tolerance-band
   (:prompt "Enter tolerance-band for length,width, height, wheelbase,
fronttrack, reartrack, car-type, fuel-consumption, top-speed,
acceleration and weight"
    :default(list 0.1 0.08 0.05 0.08 0.05 0.05 1 0.1 0.1 0.1 0.3))
    )

  :optional-inputs
  (
   :functions (list :support-chasis :protect-passenger :stylise)
   :function-relationship (list :support-chasis-and-protect-
passenger :protect-passenger-and-stylise :stylise-and-support-chasis)
   :length 5029
   :width 1902
```

```
   :height 1492
   :wheelbase 2990
   :fronttrack 1578
   :reartrack 1582
   :car-type :saloon
   :fuel-consumption 15.5
   :top-speed 237
   :acceleration 8.1
   :weight 1865
   :input-structure (the :case-base :cab)
   :optional-features (list (list :passenger-cabin :length
3000)(list :passenger-cabin :width 1902) (list :passenger-
cabin :height 1490) (list :luggage-compartment :length 529) )
   :weights-optional-features (list 0.5 0.5 0.3 0.3)
   :tolerance-band-optional-features (list 0.1 0.1 0.06 0.1)
   )

  :attributes
  (
   :according-optional-features ;;get the according features in the
case base
   (let-streams
       ((a (in (the :case-contents)))
      (case (the :case-base a))
      (b (let-streams
             ((c (in (the :optional-features)))
            (c1 (first c))
            (c2 (second c))
            (d (let-streams
                   ((e (in-tree case))
                  (e1 (the-object e :name-for-display))
                  (equal? (equal e1 c1))
                  (f (collect-if! equal?  (defaulting (list a e1 c2
(the-object e c2))))))
                ((return-when empty? f))    ))
            (g (collect! d)))
          ((return-when empty? g))))
      (data (collect! b)))
     ((return-when empty? data)))

   :according-optional-features-1 ;; sort according-optional-features
   (let-streams
       ((a (in (the :according-optional-features)))
      (b (stream-append a)))
     ((return-when empty? b)))

   :according-optional-features-2 ;; sort according-optional-
features-1
   (let-streams
       ((a (in (the :according-optional-features-1)))
      (b (stream-append a)))
     ((return-when empty? b)))


   :fbsm-optional-start ;; similarity measure on optional features
   (let-streams
       ((a (in (the :optional-features)))
      (w (in (the :weights-optional-features)))
      (t1 (in (the :tolerance-band-optional-features)))
      (b (let-streams
             ((a1 (first a))
```

```
             (a2 (second a))
             (a3 (lastcar a))
             (c (in (the :according-optional-features-2)))
             (c1 (first c))
             (c2 (second c))
             (c3 (third c))
             (c4 (fourth c))
             (match? (and (equal c2 a1) (equal c3 a2)))
             (equal? (if match? (and (< c4 (+ (* t1 a3) a3))
                                     (> c4 (- a3 (* t1 a3))))
                         nil))
             (result (if equal? (list w c1) nil))
             (result1 (collect! result)))
            ((return-when empty? (remove nil result1)))    ))    ;;( c1
a3 c4 match? equal?  result)
        (data (collect! b)))
       ((return-when empty? data))       )


   :fbsm-optional-2 ;; sort fbsm-optional-start, obtain a list of
(weight, name of case)
   (let-streams
       ((a (in (the :fbsm-optional-start)))
        (b (stream-append a)))
       ((return-when empty? b)))


   :fbsm-optional
   (let-streams
       ((a (in (the :case-contents)))
        (b (let-streams
             ((c (in (the :fbsm-optional-2)))
              (c1 (first c))
              (c2 (second c))
              (match? (equal a c2))
              (count (fby 0 (if match? (+ count c1) count))))
             ((return-when empty? (list count a)))    (a c1 c2 match?
count)    ))
        (data (collect! b)))
       ((return-when empty?  (safe-sort data #'> :key #'first))))


   :fbsm
   (let-streams
       ((a (in (the :fbsm-optional)))
        (a1 (first a))
        (a2 (second a))
        (b (let-streams
             ((c (in (the :fbsm-feature)))
              (c1 (first c))
              (c2 (second c))
              (match? (equal a2 c2))
              (count (fby a1 (if match? (+ count c1) count))))
             ((return-when empty? (list count a2)))    (a c1 c2 match?
count)    ))
        (data (collect! b)))
       ((return-when empty?  (safe-sort data #'> :key #'first))))


   :input-features
```

```
   (list (the :length) (the :width) (the :height) (the :wheelbase)
(the :fronttrack) (the :reartrack)
       (the :car-type)(the :fuel-consumption) (the :top-speed)
(the :acceleration)
       (the :weight))

   :input-functions (append (the :functions) (the :function-
relationship))

   :case-contents
   (let-streams
         ((aa (in (the :case-base :children)))
          (bb (the-object aa :name-for-display))
          (cc (collect! bb)))
       ((return-when empty? cc)))

   :fbsm-feature         ;;feature-based-similarity-measure
   (let-streams
       ((a (in (the :case-contents)))
      (case (the :case-base a))
      (matches  (let-streams
                  ((t1 (in (the :input-features)))
                   (t2 (in (the-object case :features)))
                     (t3 (in (the :tolerance-band)))
                     (t4 (in (the :weights)))
                       (feature-equal (if (numberp t1)
                                (and (< t2 (+ (* t3 t1) t1))
                                     (> t2 (- t1 (* t3 t1))))
                              (equal t1 t2)))
                   (count (fby 0 (if feature-equal (+ t4 count)
count ))))
                  ((return-when empty? (list count a)))
                ))
      (data (collect! matches)))
     ((return-when empty? (safe-sort data #'> :key
#'first))) ) ;;returns the list according to fbsm


   :sbsm-f         ;;structure-based-similarity-measure on function
model
   (let-streams
       ((a (in (the :case-contents)))
      (case (the :case-base a))
      (matches  (let-streams
                  ((t1 (in (the :input-functions)))
                   (in? (member t1 t2 ))
                   (t2 (fby (the-object case :function-model) (if in?
(remove t1 t2) t2)))
                   (equal-count (fby 0 (if in? (1+ equal-count)
equal-count)))
                   (n1 (length  (the :input-functions)))
                   (n2 (length  t2))
                   (dissm (+ (- n1 equal-count) n2)))
                ((return-when empty? (list dissm a)))
               ))
       (data (collect! matches)))
     ((return-when empty? (safe-sort data #'< :key #'first)))
(case)) ;;returns the list according to sbsm-f

   :input-relations     ;;get the relations of the elements from
input structure
```

```
   (let-streams
       ((t1 (in-tree (the :input-structure)))
      (result (collect! (defaulting
                         (the-object t1 :sub-relations) ()  ))))
     ((return-when empty?  (remove nil result))) )

   :compared-structure           ;;make list of the relations
   (let-streams
       ((t1 (in (the :input-relations)))
      (result (fby () (append result t1))))
     ((return-when empty? result)) )

   :temp                ;;get the relations of the elements from
cases and make a list of relations
   (let-streams
       ((a (in (the :case-contents)))
      (relations
       (let-streams
           ((t1 (in-tree (the :case-base a)))
            (result (collect! (defaulting
                               (the-object t1 :sub-relations) ()  ))))
         ((return-when empty?  (remove nil result))) ))
      (case  (let-streams
               ((t1 (in (the-object relations)))
                (result (fby () (append result t1))))
              ((return-when empty? result)) ))
      (data (collect!  (list case a))))
     ((return-when empty? data)))

   :sbsm-s      ;;structure-based-similarity-measure on structure
model, calculate the amount of actions
   (let-streams
       ((a (in (the :temp)))
      (b (first a))
      (c (second a))
      (matches   (let-streams
                   ((t1 (in (the-object b)))
                  (in? (member t1 t2 :test #'equal))
                  (t2 (fby (the :compared-structure) (if in? (remove
t1 t2 :count 1) t2)))
                  (equal-count (fby 0 (if in? (1+ equal-count) equal-
count)))
                  (n1 (length  (the-object b)))
                  (n2 (length  t2))
                  (dissm (* 2 (+ (- n1 equal-count) n2))))
                 ((return-when empty? (list dissm c)   ))
                ))
       (data (collect! matches)))
     ((return-when empty? (safe-sort data #'< :key
#'first))) ) ;;returns the list according to sbsm-f

   :rank-fbsm
   (let-streams
       ((a (in (the :fbsm)))
      (a1 (fby 1 (+ a1 1)))
      (a2 (append (list a1) a))
      (a3 (collect! a2)))
     ((return-when empty? a3)))

   :rank-sbsm-f
   (let-streams
```

```
    ((a (in (the :sbsm-f)))
    (a1 (fby 1 (+ a1 1)))
    (a2 (append (list a1) a))
    (a3 (collect! a2)))
   ((return-when empty? a3)))

 :rank-sbsm-s
 (let-streams
    ((a (in (the :sbsm-s)))
    (a1 (fby 1 (+ a1 1)))
    (a2 (append (list a1) a))
    (a3 (collect! a2)))
   ((return-when empty? a3)))

 :sa
 (let-streams
    ((a (in (the :rank-fbsm)))
    (a1 (first a))
    (a2 (third a))
    (b1 (let-streams
          ((b (in (the :rank-sbsm-f)))
           (b2 (third b))
           (equal? (equal a2 b2))
           (b11 (if equal? (first b) 0)))
          ((return-when equal? b11)) (b b2 equal?)  ))
    (c1 (let-streams
          ((c (in (the :rank-sbsm-s)))
           (c2 (third c))
           (equal? (equal a2 c2))
           (c11 (if equal? (first c) 0)))
          ((return-when equal? c11)) (c c2 equal?)  ))
    (sa (+ (+ (* (the :w1) a1) (* (the :w2) b1)) (* (the :w3) c1)))
    (result (collect! (list sa a2))))
   ((return-when empty? (safe-sort result #'< :key #'first)))  )

 :retrieved-case-name (second (first (the :sa)))
 :get-type
 (let-streams
      ((aa (in (the :case-base :children)))
       (bb (the-object aa :name-for-display))
       (dd (the-object aa :type))
       (cc (collect! (list bb dd))))
   ((return-when empty? cc)))

 :find-type
 (let-streams
    ((aa (in (the :get-type)))
    (bb (first aa))
    (equals? (equal bb (the :retrieved-case-name))))
   ((return-when equals? (second aa))))
   )

:parts
( (retrieved-case
  :type (the :find-type))
               )

:pseudo-parts
((case-base :type case-base)))
```

Appendix D

Code for performance retrieval

This appendix lists the ICAD code for the adaptation approach of performance retrieval introduced in Chapter 5. It is used in conjunction with the code of case base, which is partly presented in Appendix B.

```
(defpart performance-revision (box)
  :choice-attributes
  (:case-ca
   (:prompt "Enter the case name"
    :domain (:item-list
(list :cb001 :cb002 :cb003 :cb004 :cb005 :cb006 :cb007))
    :default :cb001)
   :input-performance
   (:prompt "Enter the performance"
    :domain (:selection-list
(list :provide_space_for_the_engine :provide_space_for_the_passengers
_and_luggage :provide_space_for_the_passengers :provide_space_for_the
_luggage)
                        :minimum-selections 1
                        :maximum-selections 3)
    :default (list :provide_space_for_the_engine))
   )

  :modifiable-defaulted-inputs
  (

   :length 5029
   :width 1902
   :height 1492
   )

  :attributes
  (:retrieved-case (cond ((equal (the :case-ca) :cb001) (first
(the :list-cases)))
```

```
                    ((equal (the :case-ca) :cb002) (second (the :list-
cases)))
                    ((equal (the :case-ca) :cb003) (nth 2 (the :list-
cases)))
                    ((equal (the :case-ca) :cb004) (nth 3 (the :list-
cases)))
                    ((equal (the :case-ca) :cb005) (nth 4 (the :list-
cases)))
                    ((equal (the :case-ca) :cb006) (nth 5 (the :list-
cases)))
                    ((equal (the :case-ca) :cb007) (nth 6 (the :list-
cases))))

   :retrieved-case-performance   ;;give the performance of the
retrieved case
   (let-streams
       ((a (in (the :retrieved-case :performance)))
      (b (stream-append a)))
     ((return-when empty? b)))

   :retrieved-case-subcases     ;;give the performance of the
retrieved cased alongside with the according case name
   (let-streams
       ((a (in (the :retrieved-case-performance)))
      (b (in (the :retrieved-case :sub-cases)))
      (c (collect! (list a b))))
     ((return-when empty? c)))

   :kept-subcases    ;;subcases from the retrieved case that can be
kept in the new situation
   (let-streams
       ((aa (in (the :retrieved-case-subcases)))
      (aa1 (first aa))
      (aa2 (second aa))
      (bb (member aa1 (the :input-performance)))
      (cc (collect-if! bb (defaulting aa2))))
     ((return-when empty? cc)) )

   :kept-performance  ;;performance from the retrieved case that can
be kept in the new situation
   (let-streams
       ((aa (in (the :retrieved-case-subcases)))
      (aa1 (first aa))
      (aa2 (second aa))
      (bb (member aa1 (the :input-performance)))
      (cc (collect-if! bb (defaulting aa1))))
     ((return-when empty? cc)) )

   :added-performance  ;;performance that need to be added from the
case base
   (let-streams
       ((a (in (the :kept-performance)))
      (b (fby (the :input-performance) (remove a b))))
     ((return-when empty? b)))

   :trial
(list :provide_space_for_the_passengers  :provide_space_for_the_lugga
ge)

   :search-performance    ;;search the added-performance provided by
which subcases
```

```
   (let-streams
       ((aa (in (the :added-performance)))
      (bb (let-streams
            ((a (in-tree (the :case-base)))
             (a1 (the-object a :type))
             (b (defaulting (the-object a :performance)))
             (equal? (equal (list aa) b))
             (c (collect-if! equal?  a1)))
            ((return-when empty? c))    ))
      (cc (collect! (list aa bb))))
     ((return-when empty? cc)) )
((:PROVIDE_SPACE_FOR_THE_PASSENGERS (PC001
PC003))(:PROVIDE_SPACE_FOR_THE_LUGGAGE (LC001 LC003)))

   :added-subcases    ;;sort the subcases to be added to the
retrieved case
   (let-streams
       ((a (in (the :search-performance)))
      (a1 (first a))
      (a2 (second a))
      (b (first a2))
      (c (collect! (list a1 b))))
     ((return-when empty? c)))
((:PROVIDE_SPACE_FOR_THE_PASSENGERS PC001)
(:PROVIDE_SPACE_FOR_THE_LUGGAGE LC001))

   :kept-performance-and-subcases
   (let-streams
       ((aa (in (the :retrieved-case-subcases)))
      (aa1 (first aa))
      (aa2 (second aa))
      (bb (member aa1 (the :input-performance)))
      (cc (collect-if! bb (list (defaulting aa1) aa2))))
     ((return-when empty? cc)) )

   :adapted-subcases-list (append (the :added-subcases) (the :kept-
performance-and-subcases))

   :sort-adapted-subcases-list
   (let-streams
       ((a (in (the :input-performance)))
      (b (let-streams
             ((b1 (in (the :adapted-subcases-list)))
            (b11 (first b1))
            (b12 (second b1))
            (match? (equal a b11))
            (b3 (collect-if! match?  b12)))
           ((return-when empty? b3))    ))
      (c (collect! b)))
     ((return-when empty? c)) (a b c)   )

   :sort-adapted-subcases-list-2
   (let-streams
       ((a (in (the :sort-adapted-subcases-list)))
      (b (stream-append a)))
     ((return-when empty? b)))

   :list-cases (the :case-base :children)
  )

  :parts
```

```
  ( (adapted-case
    :type case-for-adaptation)
          )

  :pseudo-parts
  (
   (case-base :type case-base)
    )
 )

(defpart case-for-adaptation (box)
  :attributes
  (:functions (the-object (make-part 'cbr) :retrieved-case :functions)
   :function-relationship (the-object (make-part 'cbr) :retrieved-
case :function-relationship)
   :function-model (append (the :functions) (the :function-
relationship))
   :width (max (the :engine-compartment :width) (the :passenger-
cabin :width) (the :luggage-compartment :width))
   :length (+ (the :engine-compartment :length) (the :passenger-
cabin :length) (the :luggage-compartment :length))
   :height (max (the :engine-compartment :height) (the :passenger-
cabin :height) (the :luggage-compartment :height))
   :car-type (the-object (make-part 'cbr) :retrieved-case :car-type)
   :wheelbase (the-object (make-part 'cbr) :retrieved-case :wheelbase)
   :fronttrack (the-object (make-part 'cbr) :retrieved-
case :fronttrack)
   :reartrack (the-object (make-part 'cbr) :retrieved-case :reartrack)
   :weight (the-object (make-part 'cbr) :retrieved-case :weight)
   :fuel-consumption (the-object (make-part 'cbr) :retrieved-
case :fuel-consumption)
   :top-speed (the-object (make-part 'cbr) :retrieved-case :top-speed)
   :acceleration (the-object (make-part 'cbr) :retrieved-
case :acceleration)
   :behaviour-feature
          (list (the :length) (the :width) (the :height)
(the :wheelbase) (the :fronttrack) (the :reartrack))
   :function-feature (list (the :car-type) (the :fuel-consumption)
(the :top-speed) (the :acceleration))
   :structure-feature (list (the :weight))
   :features (append (the :behaviour-feature) (the :function-feature)
(the :structure-feature))
   :performance
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :performance))
          (cc (collect! (defaulting bb () ))))
       ((return-when empty? cc)))

   :local-adapted-subcases-list (the-object (make-part 'cbr) :sort-
adapted-subcases-list-2)
   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :name-for-display))
          (cc (collect! bb)))
       ((return-when empty? cc)))

   )

  :report-attributes
```

```
  (:vrml-out
  (write-vrml-file (list self)
  "f:/carbody.wrl"
  :specified-planarity 0.1
  :specified-linearity 0.1
  :camera-on? t))

  :parts
  ((engine-compartment
    :type (first (the :local-adapted-subcases-list))
    :position (:bottom 0.0))
   (passenger-cabin
    :type (second (the :local-adapted-subcases-list))
    :position (:front (:from (the :engine-compartment) (the :engine-
compartment :length))))
   (luggage-compartment
    :type (if (equal (third (the :local-adapted-subcases-list)) nil)
'null-part
          (third (the :local-adapted-subcases-list)))
    :position (:front (:from (the :passenger-cabin) (the :passenger-
cabin :length)) :bottom 0.0 ))
    )
  )
```

Appendix E

Code for goal-oriented substitution

This appendix presents the ICAD code for the adaptation approach of goal-oriented substitution described in chapter 5. It is used in conjunction with the code of case base, which is partly presented in Appendix B.

```
(defpart cbr (box)
  :modifiable-defaulted-inputs
  (:case-ca (list :cb001)
   :Goal1 (list :minimise :volume :c)
   :Goal2 (list :maximise :comfort :c)
   :Goal3 (list :maximise :visualisation :c)
   :input-goals (list "G1")
     )

  :attributes
 (:retrieved-case (cond ((equal (first (the :case-ca)) :cb001) (first
(the :list-cases)))
                    ((equal (first (the :case-ca)) :cb002) (second
(the :list-cases)))
                    ((equal (first (the :case-ca)) :cb003) (nth 2
(the :list-cases)))
                    ((equal (first (the :case-ca)) :cb004) (nth 3
(the :list-cases)))
                    ((equal (first (the :case-ca)) :cb005) (nth 4
(the :list-cases)))
                    ((equal (first (the :case-ca)) :cb006) (nth 5
(the :list-cases)))
                    ((equal (first (the :case-ca)) :cb007) (nth 6
(the :list-cases))))


   :s-review-goals (the :retrieved-case :corporate-goals)   )
(:maximise :visualisation :c))

   :review-goals (the :retrieved-case :corporate-goals)
```

```
   :review-individual-goals (the :retrieved-case :find-individual-
goals)
   :input-goals-propagation (let-streams
                          ((a (in (the :input-goals)))
                           (b (let-streams
                                ((aa (in (the :review-individual-
goals)))
                                 (bb (lastcar aa))
                                 (bb1 (symbol-name bb))
                                 (bb2 (char bb1 1))
                                 (aa1 (char a 1))
                                 (cc (equal aa1 bb2))
                                 (dd (collect-if! cc aa nil)))
                                ((return-when empty? dd))))
                           (c (collect! b)))
                          ((return-when empty? c)))
   :reform-igp (let-streams
                 ((a (in (the :input-goals-propagation)))
                  (b (stream-append a)))
               ((return-when empty? b)))

   :find-conflict (let-streams
                    ((a (in (the :reform-igp)))
                     (a1 (first a))
                     (a2 (second a))
                     (a3 (nth 2 a))
                     (b (remove a (the :reform-igp)))
                     (c (let-streams
                          ((aa (in b))
                           (aa1 (first aa))
                           (aa2 (second aa))
                           (aa3 (nth 2 aa))
                           (conflict? (and (and (equal aa1 a1) (equal
aa3 a3)) (not (equal aa2 a2))))
                          (bb (collect-if! conflict? (list aa a) nil)))
                          ((return-when empty? (remove nil bb)))))
                     (d (collect!  c)))
                  ((return-when empty? (remove nil d))))
   :reform-find-conflict (let-streams
                          ((a (in (the :find-conflict)))
                           (b (stream-append a)))
                        ((return-when empty? b)))
   :reform-find-conflict-2 (let-streams
                          ((a (in (the :reform-find-conflict)))
                           (b (stream-append a)))
                          ((return-when empty? b)))
   :simplified-goals (let-streams
                     ((a (in (the :reform-find-conflict-2)))
                      (b (fby (the :reform-igp) (remove a b))))
                     ((return-when empty? b)))

   :value-in-sg (let-streams
                 ((a (in (the :simplified-goals)))
                  (a1 (first a))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (b (let-streams
                       ((aa (in-tree (the :retrieved-case)))
                        (aa1 (the-object aa :name-for-display))
                        (bb (equal a1 aa1))
                        (dd (collect-if! bb (the-object aa a3) nil)))
```

```
                    ((return-when empty?  (remove nil dd)))  (a a1
aa1 bb)    ))
                 (c (collect! (append (list a1 a2 a3) b))))

              ((return-when empty? c))   )

   :find-substitution (let-streams
                    ((a (in (the :value-in-sg)))
                     (a1 (first a))
                     (a2 (second a))
                     (a3 (nth 2 a))
                     (a4 (nth 3 a))
                     (b (let-streams
                          ((aa (in-tree (the :case-base)))
                           (aa1 (the-object aa :name-for-display))
                           (bb1 (equal a1 aa1))
                           (aa2 (the-object aa a3))
                           (bb2 (if bb1 (and (> aa2 a4) (equal
a2 :maximise)) nil))
                           (bb3 (if bb1 (and (< aa2 a4) (equal
a2 :minimise)) nil))
                           (aa4 (the-object aa :type))
                           (cc (collect-if! (or bb2 bb3) (list aa1
aa4))))
                         ((return-when empty? cc))))

                     (c (collect!  b)))
                  ((return-when empty?  (remove nil c))))

   :reform-substitution (let-streams
                      ((a (in (the :find-substitution)))
                       (a1 (first a))
                       (b (collect! a1)))
                    ((return-when empty? b)))

   :rs-performance (let-streams
                   ((a (in (the :reform-substitution)))
                  (a1 (first a))
                  (b (let-streams
                         ((aa (in-tree (the :retrieved-case)))
                        (aa1 (the-object aa :name-for-display))
                        (aa2 (defaulting (the-object
aa :performance)))
                        (bb (equal aa1 a1))
                        (cc (collect-if! bb aa2)))
                       ((return-when empty? cc))))
                  (c (collect! (append (list a) b))))
                 ((return-when empty? c)))

   :reform-rsp (let-streams
              ((a (in (the :rs-performance)))
               (a1 (first a))
               (a2 (second a))
               (b (collect! (append a1 a2))))
            ((return-when empty? b)))

   :list-cases (the :case-base :children)

   :functions (the :retrieved-case :functions)
   :function-relationship (the :retrieved-case :function-relationship)
```

```
   :function-model (append (the :functions) (the :function-
relationship))
   :width (max (the :engine-compartment :width) (the :passenger-
cabin :width) (the :luggage-compartment :width))
   :length (+ (the :engine-compartment :length) (the :passenger-
cabin :length) (the :luggage-compartment :length))
   :height (max (the :engine-compartment :height) (the :passenger-
cabin :height) (the :luggage-compartment :height))
   :car-type (the :retrieved-case :car-type)
   :wheelbase (the :retrieved-case :wheelbase)
   :fronttrack (the :retrieved-case :fronttrack)
   :reartrack (the :retrieved-case :reartrack)
   :weight (the :retrieved-case :weight)
   :fuel-consumption (the :retrieved-case :fuel-consumption)
   :top-speed (the :retrieved-case :top-speed)
   :acceleration (the :retrieved-case :acceleration)
   :behaviour-feature
          (list (the :length) (the :width) (the :height)
(the :wheelbase) (the :fronttrack) (the :reartrack))
   :function-feature (list (the :car-type) (the :fuel-consumption)
(the :top-speed) (the :acceleration))
   :structure-feature (list (the :weight))
   :features (append (the :behaviour-feature) (the :function-feature)
(the :structure-feature))
   :performance
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :performance))
          (cc (collect! (defaulting bb () ))))
       ((return-when empty? cc)))
   :local-retrieved-case-sub-relations (the :retrieved-case :sub-
relations)
   :check-ec (equal (first (the :local-retrieved-case-sub-relations))
                :engine-compartment)
   :check-pc (equal (second (the :local-retrieved-case-sub-relations))
                :passenger-cabin)
   :check-lc (equal (defaulting (nth 2 (the :local-retrieved-case-
sub-relations)))
                 :luggage-compartment)
   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :name-for-display))
          (cc (collect! bb)))
       ((return-when empty? cc)))

  )

  :parts
  ((engine-compartment
    :type (if (the :check-ec) 'ecgoal 'null-part)
    :position (:bottom 0.0))
   (passenger-cabin
    :type (if (the :check-pc) 'pcgoal 'null-part)
    :position (:front (:from (the :engine-compartment) (the :engine-
compartment :length))))
   (luggage-compartment
    :type (if (the :check-lc) 'lcgoal 'null-part)
    :position (:front (:from (the :passenger-cabin) (the :passenger-
cabin :length)) :bottom 0.0 ))
    )
```

```
  :pseudo-parts
  ((case-base :type case-base))
 )

(defpart ecgoal (box)
  :attributes
  (:width (the :local-retrieved-case :engine-compartment :width)
   :length (the :local-retrieved-case :engine-compartment :length)
   :height (the :local-retrieved-case :engine-compartment :height)
   :performance (list :provide_space_for_the_engine)
   :goals (list (the :G1-G1) (the :G3-G1))
   :G1-G1 (list :minimise :volume :c :G1)
   :G3-G1 (list :minimise :length :i :G3)
   :local-retrieved-case (the-object (make-part 'cbr) :retrieved-case)
   :local-reform-rsp (the-object (make-part 'cbr) :reform-rsp)
   :check-bonnet (let-streams
                 ((a (in (the :local-reform-rsp)))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (c (equal
a3 :provide_top_cover_for_the_engine_compartment))
                  (d (collect-if! c (list c a2) nil   )))
              ((return-when empty? (first d))))
   :check-bonnet-2 (let-streams
                  ((a (in-tree (the :local-retrieved-case)))
                 (a1 (the-object a :type))
                 (a2 (defaulting (the-object a :performance)))
                 (b (equal a2
(list :provide_top_cover_for_the_engine_compartment)))
                 (c (collect-if! b a1)))
                ((return-when empty? (first c))))
   :check-rightfrontwing (let-streams
                 ((a (in (the :local-reform-rsp)))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (c (equal
a3 :provide_right_side_cover_for_the_engine_compartment))
                  (d (collect-if! c (list c a2) nil   )))
              ((return-when empty? (first d))))
   :check-rightfrontwing-2 (let-streams
                  ((a (in-tree (the :local-retrieved-case)))
                 (a1 (the-object a :type))
                 (a2 (defaulting (the-object a :performance)))
                 (b (equal a2
(list :provide_right_side_cover_for_the_engine_compartment)))
                 (c (collect-if! b a1)))
                  ((return-when empty? (first c))))
   :check-leftfrontwing (let-streams
                 ((a (in (the :local-reform-rsp)))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (c (equal
a3 :provide_left_side_cover_for_the_engine_compartment))
                  (d (collect-if! c (list c a2) nil   )))
              ((return-when empty? (first d))))
   :check-leftfrontwing-2 (let-streams
                  ((a (in-tree (the :local-retrieved-case)))
                 (a1 (the-object a :type))
                 (a2 (defaulting (the-object a :performance)))
```

```
                    (b (equal a2
(list :provide_left_side_cover_for_the_engine_compartment)))
                    (c (collect-if! b a1)))
                     ((return-when empty? (first c))))
   :check-frontpanel (let-streams
                   ((a (in (the :local-reform-rsp)))
                    (a2 (second a))
                    (a3 (nth 2 a))
                    (c (equal
a3 :provide_front_cover_for_the_engine_compartment))
                    (d (collect-if! c (list c a2) nil   )))
                 ((return-when empty? (first d))))
   :check-frontpanel-2 (let-streams
                     ((a (in-tree (the :local-retrieved-case)))
                    (a1 (the-object a :type))
                    (a2 (defaulting (the-object a :performance)))
                    (b (equal a2
(list :provide_front_cover_for_the_engine_compartment)))
                    (c (collect-if! b a1)))
                     ((return-when empty? (first c))))
   :check-collection (list (the :check-bonnet) (the :check-
rightfrontwing) (the :check-leftfrontwing) (the :check-frontpanel))
   :check-collection-2 (list (the :check-bonnet-2) (the :check-
rightfrontwing-2) (the :check-leftfrontwing-2) (the :check-
frontpanel-2))
   :bonnet-type  (if (and (the :goal-oriented?)
                       (equal (first (first (the :check-collection)))
't))
                  (second (first (the :check-collection)))
                (first (the :check-collection-2)))
   :goal-oriented? (equal 1 1)
   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :name-for-display))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))

  :report-attributes
  (:vrml-out
  (write-vrml-file (list self)
  "f:/engine-compartment.wrl"
  :specified-planarity 0.1
  :specified-linearity 0.1
  :camera-on? t))

  :parts
  ((bonnet :type (the :bonnet-type))
   (rightfrontwing :type (if (and (the :goal-oriented?)
                            (equal (first (second (the :check-
collection))) 't))
                  (second (second (the :check-collection)))
                (second (the :check-collection-2))))
   (leftfrontwing :type (if (and (the :goal-oriented?)
                       (equal (first (the :check-leftfrontwing)) 't))
                  (second (the :check-leftfrontwing))
                (the :check-leftfrontwing-2)))
   (frontpanel  :type (if (and (the :goal-oriented?)
                       (equal (first (the :check-frontpanel)) 't))
                  (second (the :check-frontpanel))
```

```
                  (the :check-frontpanel-2)))))

(defpart pcgoal (box)
  :attributes
  (:width (the :local-retrieved-case :passenger-cabin :width)
   :length (the :local-retrieved-case :passenger-cabin :length)
   :height (the :local-retrieved-case :passenger-cabin :height)
   :performance (list :provide_space_for_the_passengers)
   :goals (list (the :G1-G2) (the :G2-G1))
   :G1-G2 (list :minimise :volume :c :G1)
   :G2-G1 (list :maximise :volume :c :G2)
   :local-retrieved-case (the-object (make-part 'cbr) :retrieved-case)
   :local-reform-rsp (the-object (make-part 'cbr) :reform-rsp)
   :check-roofpanel (let-streams
                  ((a (in (the :local-reform-rsp)))
                   (a2 (second a))
                   (a3 (nth 2 a))
                   (c (equal
a3 :provide_top_cover_for_the_passenger_cabin))
                   (d (collect-if! c (list c a2) nil   )))
                 ((return-when empty? (first d))))
   :check-roofpanel-2 (let-streams
                    ((a (in-tree (the :local-retrieved-case)))
                   (a1 (the-object a :type)(the-object (make-part
'lcgoal) :leftrearwing))
                   (a2 (defaulting (the-object a :performance)))
                   (b (equal a2
(list :provide_top_cover_for_the_passenger_cabin )))
                   (c (collect-if! b a1)))
                 ((return-when empty? (first c))))
   :check-rightside (let-streams
                  ((a (in (the :local-reform-rsp)))
                   (a2 (second a))
                   (a3 (nth 2 a))
                   (c (equal
a3 :provide_right_side_cover_for_the_passenger_cabin))
                   (d (collect-if! c (list c a2) nil   )))
                 ((return-when empty? (first d))))
   :check-rightside-2 (let-streams
                    ((a (in-tree (the :local-retrieved-case)))
                   (a1 (the-object a :type))
                   (a2 (defaulting (the-object a :performance)))
                   (b (equal a2
(list :provide_right_side_cover_for_the_passenger_cabin)))
                   (c (collect-if! b a1)))
                    ((return-when empty? (first c))))
   :check-leftside (let-streams
                  ((a (in (the :local-reform-rsp)))
                   (a2 (second a))
                   (a3 (nth 2 a))
                   (c (equal
a3 :provide_left_side_cover_for_the_passenger_cabin))
                   (d (collect-if! c (list c a2) nil   )))
                 ((return-when empty? (first d))))
   :check-leftside-2 (let-streams
                    ((a (in-tree (the :local-retrieved-case)))
                   (a1 (the-object a :type))
                   (a2 (defaulting (the-object a :performance)))
                   (b (equal a2
(list :provide_left_side_cover_for_the_passenger_cabin)))
                   (c (collect-if! b a1)))
```

```
                ((return-when empty? (first c))))
   :goal-oriented? (equal 1 1)

   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :name-for-display))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))

  :report-attributes
  (:vrml-out
  (write-vrml-file (list self)
  "f:/paasenger-cabin.wrl"
  :specified-planarity 0.1
  :specified-linearity 0.1
  :camera-on? t)
   )

  :parts
  ((roofpanel  :type (if (and (the :goal-oriented?)
                       (equal (first (the :check-roofpanel)) 't))
                    (second (the :check-roofpanel))
                 (the :check-roofpanel-2)))
   (rightside  :type (if (and (the :goal-oriented?)
                       (equal (first (the :check-rightside)) 't))
                    (second (the :check-rightside))
                 (the :check-rightside-2)))
   (leftside  :type (if (and (the :goal-oriented?)
                       (equal (first (the :check-leftside)) 't))
                    (second (the :check-leftside))
                    (the :check-leftside-2)))))

(defpart lcgoal (box)
  :attributes
  (:width (the :local-retrieved-case :luggage-compartment :width)
   :length (the :local-retrieved-case :luggage-compartment :length)
   :height (the :local-retrieved-case :luggage-compartment :height)
   :performance (list :provide_space_for_the_luggage)
   :goals (list (the :G1-G3))
   :G1-G3 (list :minimise :volume :c :G1)
   :local-retrieved-case (the-object (make-part 'cbr) :retrieved-case)
   :local-reform-rsp (the-object (make-part 'cbr) :reform-rsp)
   :check-lcdoor (let-streams
                ((a (in (the :local-reform-rsp)))
                 (a2 (second a))
                 (a3 (nth 2 a))
                 (c (equal
a3 :provide_top_cover_for_the_luggage_compartment))
                 (d (collect-if! c (list c a2) nil   )))
             ((return-when empty? (first d))))
   :check-lcdoor-2 (let-streams
                 ((a (in-tree (the :local-retrieved-case)))
                 (a1 (the-object a :type))
                 (a2 (defaulting (the-object a :performance)))
                 (b (equal a2
(list :provide_top_cover_for_the_luggage_compartment)))
                 (c (collect-if! b a1)))
             ((return-when empty? (first c))))
   :check-rightrearwing (let-streams
```

```
                 ((a (in (the :local-reform-rsp)))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (c (equal
a3 :provide_right_side_cover_for_the_luggage_compartment))
                  (d (collect-if! c (list c a2) nil   )))
               ((return-when empty? (first d))))
   :check-rightrearwing-2 (let-streams
                   ((a (in-tree (the :local-retrieved-case)))
                  (a1 (the-object a :type))
                  (a2 (defaulting (the-object a :performance)))
                  (b (equal a2
(list :provide_right_side_cover_for_the_luggage_compartment)))
                  (c (collect-if! b a1)))
                   ((return-when empty? (first c))))
   :check-leftrearwing (let-streams
                 ((a (in (the :local-reform-rsp)))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (c (equal
a3 :provide_left_side_cover_for_the_luggage_compartment))
                  (d (collect-if! c (list c a2) nil   )))
               ((return-when empty? (first d))))
   :check-leftrearwing-2 (let-streams
                   ((a (in-tree (the :local-retrieved-case)))
                  (a1 (the-object a :type))
                  (a2 (defaulting (the-object a :performance)))
                  (b (equal a2
(list :provide_left_side_cover_for_the_luggage_compartment)))
                  (c (collect-if! b a1)))
                   ((return-when empty? (first c))))
   :check-rearpanel (let-streams
                 ((a (in (the :local-reform-rsp)))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (c (equal
a3 :provide_back_cover_for_the_luggage_compartment))
                  (d (collect-if! c (list c a2) nil   )))
               ((return-when empty? (first d))))
   :check-rearpanel-2 (let-streams
                   ((a (in-tree (the :local-retrieved-case)))
                  (a1 (the-object a :type))
                  (a2 (defaulting (the-object a :performance)))
                  (b (equal a2
(list :provide_back_cover_for_the_luggage_compartment)))
                  (c (collect-if! b a1)))
               ((return-when empty? (first c))))
   :goal-oriented? (equal 1 1)

   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :name-for-display))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))

  :report-attributes
  (:vrml-out
  (write-vrml-file (list self)
  "f:/luggage-compartment.wrl"
```

```
  :specified-planarity 0.1
  :specified-linearity 0.1(the-object (make-part
'lcgoal) :leftrearwing)
  :camera-on? t)
  )

  :parts
  ((lcdoor  :type (if (and (the :goal-oriented?)
                        (equal (first (the :check-lcdoor)) 't))
                   (second (the :check-lcdoor))
                 (the :check-lcdoor-2)))
   (rightrearwing :type (if (and (the :goal-oriented?)
                        (equal (first (the :check-rightrearwing)) 't))
                   (second (the :check-rightrearwing))
                 (the :check-rightrearwing-2)))
   (leftrearwing :type (if (and (the :goal-oriented?)
                        (equal (first (the :check-leftrearwing)) 't))
                   (second (the :check-leftrearwing))
                 (the :check-leftrearwing-2)))
   (rearpanel :type (if (and (the :goal-oriented?)
                        (equal (first (the :check-rearpanel)) 't))
                   (second (the :check-rearpanel))
                   (the :check-rearpanel-2)))))
```

Appendix F

Code for fractal-based re-design

This appendix gives the ICAD code for fractal-based re-design described in Chapter 5. The fractal-based re-design is an integrated system which is made up of the subsystems described in Chapter 4 and Chapter 5, with their codes in Appendix D, E, and F. The code presented here is also used for the case study in Chapter 5, in conjunction with the code of case base, which is partly presented in Appendix B.

```
 (defpart cbr (box)
  :choice-attributes
  (:retrieve-or-adapt
   (:prompt "Do you need to retrieve or retrieve and adapt a design?"
    :domain (:item-list (list :retrieve :retrieve-and-adapt))
    :default :retrieve)
   :adaptation-method
   (:prompt "Performance-revision or goal-oriented adaptation?"
    :domain (:item-list (list :performance-revision :goal-oriented))
    :default :performance-revision)
   :input-performance
   (:prompt "Enter the performance"
    :domain (:selection-list
(list :provide_space_for_the_engine :provide_space_for_the_passengers
_and_luggage :provide_space_for_the_passengers :provide_space_for_the
_luggage)
                      :minimum-selections 1
                      :maximum-selections 3)
    :default (list :provide_space_for_the_engine))

   )

   :modifiable-defaulted-inputs
  (
   :Goal1 (list :minimise :volume :c)
   :Goal2 (list :maximise :comfort :c)
   :Goal3 (list :maximise :visualisation :c)
   :input-goals (list "G1" "G2")
```

```
   :functions (list :support-chasis :protect-passenger :stylise)
   :function-relationship (list :support-chasis-and-protect-
passenger :protect-passenger-and-stylise :stylise-and-support-chasis)
   :length 5029
   :width 1902
   :height 1492
   :wheelbase 2990
   :fronttrack 1578
   :reartrack 1582
   :car-type :saloon
   :fuel-consumption 15.5
   :top-speed 237
   :acceleration 8.1
   :weight 1865
   :input-structure (the :case-base :cab)
   :weights (list 0.5 0.5 0.2 0.2 0.2 0.2 0.5 0.3 0.3 0.3
0.3)     ;;weights for feature
   :tolerance-band (list 0.1 0.08 0.05 0.08 0.05 0.05 1 0.1 0.1 0.1
0.3)
   :w1 0.4   ;;weight for feature-based similarity measure
   :w2 0.3   ;;weight for structure-based similarity measure on
function
   :w3 0.3   ;;weight for structure-based similarity measure on
structure
   :optional-features (list (list :passenger-cabin :length
3000)(list :passenger-cabin :width 1902) (list :passenger-
cabin :height 1490) (list :luggage-compartment :length 529) )
   :weights-optional-features (list 0.5 0.5 0.3 0.3)
   :tolerance-band-optional-features (list 0.1 0.1 0.06 0.1)
   )

  :attributes
  (
   :according-optional-features ;;get the according features in the
case base
   (let-streams
       ((a (in (the :case-contents)))
      (case (the :case-base a))
      (b (let-streams
             ((c (in (the :optional-features)))
            (c1 (first c))
            (c2 (second c))
            (d (let-streams
                   ((e (in-tree case))
                  (e1 (the-object e :name-for-display))
                  (equal? (equal e1 c1))
                  (f (collect-if! equal?  (defaulting (list a e1 c2
(the-object e c2))))))
                 ((return-when empty? f))    ))
            (g (collect! d)))
           ((return-when empty? g))))
      (data (collect! b)))
     ((return-when empty? data)))

   :according-optional-features-1 ;; sort according-optional-features
   (let-streams
       ((a (in (the :according-optional-features)))
      (b (stream-append a)))
     ((return-when empty? b)))
```

```
   :according-optional-features-2 ;; sort according-optional-
features-1
   (let-streams
       ((a (in (the :according-optional-features-1)))
      (b (stream-append a)))
     ((return-when empty? b)))


   :fbsm-optional-start ;; similarity measure on optional features
   (let-streams
       ((a (in (the :optional-features)))
      (w (in (the :weights-optional-features)))
      (t1 (in (the :tolerance-band-optional-features)))
      (b (let-streams
             ((a1 (first a))
            (a2 (second a))
            (a3 (lastcar a))
            (c (in (the :according-optional-features-2)))
            (c1 (first c))
            (c2 (second c))
            (c3 (third c))
            (c4 (fourth c))
            (match? (and (equal c2 a1) (equal c3 a2)))
            (equal? (if match? (and (< c4 (+ (* t1 a3) a3))
                                (> c4 (- a3 (* t1 a3))))
                        nil))
            (result (if equal? (list w c1) nil))
            (result1 (collect! result)))
           ((return-when empty? (remove nil result1)))    ))    ;;( c1
a3 c4 match? equal?  result)
      (data (collect! b)))
     ((return-when empty? data))     )


   :fbsm-optional-2 ;; sort fbsm-optional-start, obtain a list of
(weight, name of case)
   (let-streams
       ((a (in (the :fbsm-optional-start)))
      (b (stream-append a)))
     ((return-when empty? b)))


   :fbsm-optional
   (let-streams
       ((a (in (the :case-contents)))
      (b (let-streams
            ((c (in (the :fbsm-optional-2)))
             (c1 (first c))
             (c2 (second c))
             (match? (equal a c2))
             (count (fby 0 (if match? (+ count c1) count))))
            ((return-when empty? (list count a)))    (a c1 c2 match?
count)    ))
      (data (collect! b)))
     ((return-when empty?  (safe-sort data #'> :key #'first))))


   :fbsm
   (let-streams
       ((a (in (the :fbsm-optional)))
      (a1 (first a))
```

```
       (a2 (second a))
       (b (let-streams
             ((c (in (the :fbsm-feature)))
              (c1 (first c))
              (c2 (second c))
              (match? (equal a2 c2))
              (count (fby a1 (if match? (+ count c1) count))))
             ((return-when empty? (list count a2)))   (a c1 c2 match?
count)   ))
       (data (collect! b)))
      ((return-when empty?  (safe-sort data #'> :key #'first))))


   :input-features
   (list (the :length) (the :width) (the :height) (the :wheelbase)
(the :fronttrack) (the :reartrack)
        (the :car-type)(the :fuel-consumption) (the :top-speed)
(the :acceleration)
        (the :weight))

   :input-functions (append (the :functions) (the :function-
relationship))

   :case-contents
   (let-streams
         ((aa (in (the :case-base :children)))
          (bb (the-object aa :name-for-display))
          (cc (collect! bb)))
       ((return-when empty? cc)))

   :fbsm-feature         ;;feature-based-similarity-measure
   (let-streams
       ((a (in (the :case-contents)))
       (case (the :case-base a))
       (matches  (let-streams
                   ((t1 (in (the :input-features)))
                    (t2 (in (the-object case :features)))
                      (t3 (in (the :tolerance-band)))
                      (t4 (in (the :weights)))
                        (feature-equal (if (numberp t1)
                                  (and (< t2 (+ (* t3 t1) t1))
                                       (> t2 (- t1 (* t3 t1))))
                                (equal t1 t2)))
                    (count (fby 0 (if feature-equal (+ t4 count)
count ))))
                   ((return-when empty? (list count a)))
                  ))
       (data (collect! matches)))
      ((return-when empty? (safe-sort data #'> :key
#'first))) ) ;;returns the list according to fbsm


   :sbsm-f         ;;structure-based-similarity-measure on function
model
   (let-streams
       ((a (in (the :case-contents)))
       (case (the :case-base a))
       (matches  (let-streams
                   ((t1 (in (the :input-functions)))
                    (in? (member t1 t2 ))
```

```
                    (t2 (fby (the-object case :function-model) (if in?
(remove t1 t2) t2)))
                    (equal-count (fby 0 (if in? (1+ equal-count)
equal-count)))
                    (n1 (length  (the :input-functions)))
                    (n2 (length  t2))
                    (dissm (+ (- n1 equal-count) n2)))
                 ((return-when empty? (list dissm a)))
               ))
        (data (collect! matches)))
      ((return-when empty? (safe-sort data #'< :key #'first)))
(case)) ;;returns the list according to sbsm-f




   :input-relations     ;;get the relations of the elements from
input structre
   (let-streams
       ((t1 (in-tree (the :input-structure)))
      (result (collect! (defaulting
                          (the-object t1 :sub-relations) ()   ))))
     ((return-when empty?  (remove nil result))) )

   :compared-structure           ;;make list of the relations
   (let-streams
       ((t1 (in (the :input-relations)))
      (result (fby () (append result t1))))
     ((return-when empty? result)) )

   :temp                 ;;get the relations of the elements from
cases and make a list of relations
   (let-streams
       ((a (in (the :case-contents)))
      (relations
       (let-streams
           ((t1 (in-tree (the :case-base a)))
            (result (collect! (defaulting
                                (the-object t1 :sub-relations) ()   ))))
         ((return-when empty?  (remove nil result))) ))
      (case  (let-streams
               ((t1 (in (the-object relations)))
                (result (fby () (append result t1))))
             ((return-when empty? result)) ))
      (data (collect!  (list case a))))
     ((return-when empty? data)))

   :sbsm-s      ;;structure-based-similarity-measure on structure
model, calculate the amount of actions
   (let-streams
       ((a (in (the :temp)))
      (b (first a))
      (c (second a))
      (matches   (let-streams
                    ((t1 (in (the-object b)))
                   (in? (member t1 t2 :test #'equal))
                   (t2 (fby (the :compared-structure) (if in? (remove
t1 t2 :count 1) t2)))
                   (equal-count (fby 0 (if in? (1+ equal-count) equal-
count)))
                   (n1 (length  (the-object b)))
```

```
                (n2 (length  t2))
                (dissm (* 2 (+ (- n1 equal-count) n2))))
               ((return-when empty? (list dissm c)    ))
              ))
      (data (collect! matches)))
     ((return-when empty? (safe-sort data #'< :key
#'first))) ) ;;returns the list according to sbsm-f


   :rank-fbsm
   (let-streams
       ((a (in (the :fbsm)))
      (a1 (fby 1 (+ a1 1)))
      (a2 (append (list a1) a))
      (a3 (collect! a2)))
     ((return-when empty? a3)))

   :rank-sbsm-f
   (let-streams
       ((a (in (the :sbsm-f)))
      (a1 (fby 1 (+ a1 1)))
      (a2 (append (list a1) a))
      (a3 (collect! a2)))
     ((return-when empty? a3)))

   :rank-sbsm-s
   (let-streams
       ((a (in (the :sbsm-s)))
      (a1 (fby 1 (+ a1 1)))
      (a2 (append (list a1) a))
      (a3 (collect! a2)))
     ((return-when empty? a3)))

   :sa
   (let-streams
       ((a (in (the :rank-fbsm)))
      (a1 (first a))
      (a2 (third a))
      (b1 (let-streams
            ((b (in (the :rank-sbsm-f)))
             (b2 (third b))
             (equal? (equal a2 b2))
             (b11 (if equal? (first b) 0)))
            ((return-when equal? b11)) (b b2 equal?)   ))
      (c1 (let-streams
            ((c (in (the :rank-sbsm-s)))
             (c2 (third c))
             (equal? (equal a2 c2))
             (c11 (if equal? (first c) 0)))
            ((return-when equal? c11)) (c c2 equal?)   ))
      (sa (+ (+ (* (the :w1) a1) (* (the :w2) b1)) (* (the :w3) c1)))
      (result (collect! (list sa a2))))
     ((return-when empty? (safe-sort result #'< :key #'first)))   )


   :retrieved-case-name (second (first (the :sa)))


   :get-type
   (let-streams
         ((aa (in (the :case-base :children)))
```

```
          (bb (the-object aa :name-for-display))
          (dd (the-object aa :type))
          (cc (collect! (list bb dd))))
     ((return-when empty? cc)))

   :find-type
   (let-streams
       ((aa (in (the :get-type)))
      (bb (first aa))
      (equals? (equal bb (the :retrieved-case-name))))
     ((return-when equals? (second aa))))

   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   ;;ADAPTATION;;;;;;;;;;;;;;;;;PERFORMANCE REVISION;;;;
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      :retrieved-case-performance   ;;give the performance of the
retrieved case
   (let-streams
       ((a (in (the :retrieved-case :performance)))
      (b (stream-append a)))
     ((return-when empty? b)))

   :retrieved-case-subcases     ;;give the performance of the
retrieved cased alongside with the according case name
   (let-streams
       ((a (in (the :retrieved-case-performance)))
      (b (in (the :retrieved-case :sub-cases)))
      (c (collect! (list a b))))
     ((return-when empty? c)))

   :kept-subcases    ;;subcases from the retrieved case that can be
kept in the new situation
   (let-streams
       ((aa (in (the :retrieved-case-subcases)))
      (aa1 (first aa))
      (aa2 (second aa))
      (bb (member aa1 (the :input-performance)))
      (cc (collect-if! bb (defaulting aa2))))
     ((return-when empty? cc)) )

   :kept-performance  ;;performance from the retrieved case that can
be kept in the new situation
   (let-streams
       ((aa (in (the :retrieved-case-subcases)))
      (aa1 (first aa))
      (aa2 (second aa))
      (bb (member aa1 (the :input-performance)))
      (cc (collect-if! bb (defaulting aa1))))
     ((return-when empty? cc)) )

   :added-performance  ;;performance that need to be added from the
case base
   (let-streams
       ((a (in (the :kept-performance)))
      (b (fby (the :input-performance) (remove a b))))
     ((return-when empty? b)))

   :trial
(list :provide_space_for_the_passengers  :provide_space_for_the_lugga
ge)
```

```
   :search-performance    ;;search the added-performanc provided by
which subcases
   (let-streams
       ((aa (in (the :added-performance)))
      (bb (let-streams
             ((a (in-tree (the :case-base)))
              (a1 (the-object a :type))
              (b (defaulting (the-object a :performance)))
              (equal? (equal (list aa) b))
              (c (collect-if! equal?  a1)))
             ((return-when empty? c))    ))
      (cc (collect! (list aa bb))))
     ((return-when empty? cc)) )
((:PROVIDE_SPACE_FOR_THE_PASSENGERS (PC001
PC003))(:PROVIDE_SPACE_FOR_THE_LUGGAGE (LC001 LC003)))

   :added-subcases    ;;sort the subcases to be added to the
retrieved case
   (let-streams
       ((a (in (the :search-performance)))
      (a1 (first a))
      (a2 (second a))
      (b (first a2))
      (c (collect! (list a1 b))))
     ((return-when empty? c)))
((:PROVIDE_SPACE_FOR_THE_PASSENGERS PC001)
(:PROVIDE_SPACE_FOR_THE_LUGGAGE LC001))

   :kept-performance-and-subcases
   (let-streams
       ((aa (in (the :retrieved-case-subcases)))
      (aa1 (first aa))
      (aa2 (second aa))
      (bb (member aa1 (the :input-performance)))
      (cc (collect-if! bb (list (defaulting aa1) aa2))))
     ((return-when empty? cc)) )

   :adapted-subcases-list (append (the :added-subcases) (the :kept-
performance-and-subcases))

   :sort-adapted-subcases-list
   (let-streams
       ((a (in (the :input-performance)))
      (b (let-streams
             ((b1 (in (the :adapted-subcases-list)))
            (b11 (first b1))
            (b12 (second b1))
            (match? (equal a b11))
            (b3 (collect-if! match?  b12)))
           ((return-when empty? b3))    ))
      (c (collect! b)))
     ((return-when empty? c)) (a b c)   )

   :sort-adapted-subcases-list-2
   (let-streams
       ((a (in (the :sort-adapted-subcases-list)))
      (b (stream-append a)))
     ((return-when empty? b)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;GOAL-ORIENTED-ADAPATION;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

   :s-review-goals (the :retrieved-case :corporate-
goals)  ;;((:MINIMISE :VOLUME :C) (:MAXIMISE :COMFORT :C)
(:MAXIMISE :VISUALISATION :C))

   :review-goals (the :retrieved-case :corporate-goals)
   :review-individual-goals (the :retrieved-case :find-individual-
goals)
   :input-goals-propagation (let-streams
                          ((a (in (the :input-goals)))
                           (b (let-streams
                                ((aa (in (the :review-individual-
goals)))
                                 (bb (lastcar aa))
                                 (bb1 (symbol-name bb))
                                 (bb2 (char bb1 1))
                                 (aa1 (char a 1))
                                 (cc (equal aa1 bb2))
                                 (dd (collect-if! cc aa nil)))
                                ((return-when empty? dd))))
                           (c (collect! b)))
                          ((return-when empty? c)))
   :reform-igp (let-streams
                ((a (in (the :input-goals-propagation)))
                 (b (stream-append a)))
              ((return-when empty? b)))

   :find-conflict (let-streams
                   ((a (in (the :reform-igp)))
                    (a1 (first a))
                    (a2 (second a))
                    (a3 (nth 2 a))
                    (b (remove a (the :reform-igp)))
                    (c (let-streams
                         ((aa (in b))
                          (aa1 (first aa))
                          (aa2 (second aa))
                          (aa3 (nth 2 aa))
                          (conflict? (and (and (equal aa1 a1) (equal
aa3 a3)) (not (equal aa2 a2))))
                         (bb (collect-if! conflict? (list aa a) nil)))
                         ((return-when empty? (remove nil bb)))))
                    (d (collect!  c)))
                  ((return-when empty? (remove nil d))))
   :reform-find-conflict (let-streams
                         ((a (in (the :find-conflict)))
                          (b (stream-append a)))
                       ((return-when empty? b)))
   :reform-find-conflict-2 (let-streams
                         ((a (in (the :reform-find-conflict)))
                          (b (stream-append a)))
                         ((return-when empty? b)))
   :simplified-goals (let-streams
                    ((a (in (the :reform-find-conflict-2)))
                     (b (fby (the :reform-igp) (remove a b))))
                    ((return-when empty? b))))

   :value-in-sg (let-streams
```

```
               ((a (in (the :simplified-goals)))
                (a1 (first a))
                (a2 (second a))
                (a3 (nth 2 a))
                (b (let-streams
                     ((aa (in-tree (the :retrieved-case)))
                      (aa1 (the-object aa :name-for-display))
                      (bb (equal a1 aa1))
                      (dd (collect-if! bb (the-object aa a3) nil)))
                   ((return-when empty?  (remove nil dd)))   (a a1
aa1 bb)    ))
                (c (collect! (append (list a1 a2 a3) b))))

             ((return-when empty? c))    )

   :find-substitution (let-streams
                      ((a (in (the :value-in-sg)))
                       (a1 (first a))
                       (a2 (second a))
                       (a3 (nth 2 a))
                       (a4 (nth 3 a))
                       (b (let-streams
                            ((aa (in-tree (the :case-base)))
                             (aa1 (the-object aa :name-for-display))
                             (bb1 (equal a1 aa1))
                             (aa2 (the-object aa a3))
                             (bb2 (if bb1 (and (> aa2 a4) (equal
a2 :maximise)) nil))
                             (bb3 (if bb1 (and (< aa2 a4) (equal
a2 :minimise)) nil))
                             (aa4 (the-object aa :type))
                             (cc (collect-if! (or bb2 bb3) (list aa1
aa4))))
                          ((return-when empty? cc))))

                       (c (collect!  b)))
                    ((return-when empty?  (remove nil c))))

   :reform-substitution (let-streams
                       ((a (in (the :find-substitution)))
                        (a1 (first a))
                        (b (collect! a1)))
                      ((return-when empty? b)))

   :rs-performance (let-streams
                    ((a (in (the :reform-substitution)))
                   (a1 (first a))
                   (b (let-streams
                          ((aa (in-tree (the :retrieved-case)))
                         (aa1 (the-object aa :name-for-display))
                         (aa2 (defaulting (the-object
aa :performance)))
                         (bb (equal aa1 a1))
                         (cc (collect-if! bb aa2)))
                        ((return-when empty? cc))))
                   (c (collect! (append (list a) b))))
                 ((return-when empty? c)))

   :reform-rsp (let-streams
               ((a (in (the :rs-performance)))
                (a1 (first a))
```

```
                 (a2 (second a))
                 (b (collect! (append a1 a2))))
              ((return-when empty? b)))
  )

  :parts
  ((case-for-display
    :type (cond ((equal (the :retrieve-or-adapt) :retrieve)
             'case-for-goal-oriented-adaptation)
            ((and (equal (the :retrieve-or-adapt) :retrieve-and-adapt)
                  (equal (the :adaptation-method) :performance-
revision))
              'case-for-adaptation)
            ((and (equal (the :retrieve-or-adapt) :retrieve-and-adapt)
                  (equal (the :adaptation-method) :goal-oriented))
              'case-for-goal-oriented-adaptation))
                )
                   )

  :pseudo-parts
  (
   (retrieved-case
    :type (the :find-type))
   (case-base :type case-base)
    )
 )


(defpart case-for-adaptation (box)
  :attributes
  (:functions (the-object (make-part 'cbr) :retrieved-case :functions)
   :function-relationship (the-object (make-part 'cbr) :retrieved-
case :function-relationship)
   :function-model (append (the :functions) (the :function-
relationship))
   :width (max (the :engine-compartment :width) (the :passenger-
cabin :width) (the :luggage-compartment :width))
   :length (+ (the :engine-compartment :length) (the :passenger-
cabin :length) (the :luggage-compartment :length))
   :height (max (the :engine-compartment :height) (the :passenger-
cabin :height) (the :luggage-compartment :height))
   :car-type (the-object (make-part 'cbr) :retrieved-case :car-type)
   :wheelbase (the-object (make-part 'cbr) :retrieved-case :wheelbase)
   :fronttrack (the-object (make-part 'cbr) :retrieved-
case :fronttrack)
   :reartrack (the-object (make-part 'cbr) :retrieved-case :reartrack)
   :weight (the-object (make-part 'cbr) :retrieved-case :weight)
   :fuel-consumption (the-object (make-part 'cbr) :retrieved-
case :fuel-consumption)
   :top-speed (the-object (make-part 'cbr) :retrieved-case :top-speed)
   :acceleration (the-object (make-part 'cbr) :retrieved-
case :acceleration)
   :behaviour-feature
          (list (the :length) (the :width) (the :height)
(the :wheelbase) (the :fronttrack) (the :reartrack))
   :function-feature (list (the :car-type) (the :fuel-consumption)
(the :top-speed) (the :acceleration))
   :structure-feature (list (the :weight))
   :features (append (the :behaviour-feature) (the :function-feature)
(the :structure-feature))
   :performance
```

```
    (let-streams
          ((aa (in (the :children)))
           (bb (the-object aa :performance))
           (cc (collect! (defaulting bb () ))))
        ((return-when empty? cc)))
    :sub-relations
    (let-streams
          ((aa (in (the :children)))
           (bb (the-object aa :name-for-display))
           (cc (collect! bb)))
        ((return-when empty? cc)))
    )

   :report-attributes
   (:vrml-out
   (write-vrml-file (list self)
   "f:/carbody.wrl"
   :specified-planarity 0.1
   :specified-linearity 0.1
   :camera-on? t))

   :parts
   ((engine-compartment
     :type (first (the-object (make-part 'cbr) :sort-adapted-subcases-
list-2))
     :position (:bottom 0.0))
    (passenger-cabin
     :type (second (the-object (make-part 'cbr) :sort-adapted-
subcases-list-2))
     :position (:front (:from (the :engine-compartment) (the :engine-
compartment :length))))
    (luggage-compartment
     :type (if (equal (third (the-object (make-part 'cbr) :sort-
adapted-subcases-list-2)) nil) 'null-part
           (third (the-object (make-part 'cbr) :sort-adapted-subcases-
list-2)))
     :position (:front (:from (the :passenger-cabin) (the :passenger-
cabin :length)) :bottom 0.0 ))
     )
  )

(defpart case-for-goal-oriented-adaptation (box)
  :attributes
  (:functions (the-object (make-part 'cbr) :retrieved-case :functions)
   :function-relationship (the-object (make-part 'cbr) :retrieved-
case :function-relationship)
   :function-model (append (the :functions) (the :function-
relationship))
   :width (max (the :engine-compartment :width) (the :passenger-
cabin :width) (the :luggage-compartment :width))
   :length (+ (the :engine-compartment :length) (the :passenger-
cabin :length) (the :luggage-compartment :length))
   :height (max (the :engine-compartment :height) (the :passenger-
cabin :height) (the :luggage-compartment :height))
   :car-type (the-object (make-part 'cbr) :retrieved-case :car-type)
   :wheelbase (the-object (make-part 'cbr) :retrieved-case :wheelbase)
   :fronttrack (the-object (make-part 'cbr) :retrieved-
case :fronttrack)
   :reartrack (the-object (make-part 'cbr) :retrieved-case :reartrack)
   :weight (the-object (make-part 'cbr) :retrieved-case :weight)
```

```
   :fuel-consumption (the-object (make-part 'cbr) :retrieved-
case :fuel-consumption)
   :top-speed (the-object (make-part 'cbr) :retrieved-case :top-speed)
   :acceleration (the-object (make-part 'cbr) :retrieved-
case :acceleration)
   :behaviour-feature
          (list (the :length) (the :width) (the :height)
(the :wheelbase) (the :fronttrack) (the :reartrack))
   :function-feature (list (the :car-type) (the :fuel-consumption)
(the :top-speed) (the :acceleration))
   :structure-feature (list (the :weight))
   :features (append (the :behaviour-feature) (the :function-feature)
(the :structure-feature))
   :performance
   (let-streams
        ((aa (in (the :children)))
         (bb (the-object aa :performance))
         (cc (collect! (defaulting bb () ))))
      ((return-when empty? cc)))

   :check-ec (equal (first (the-object (make-part 'cbr) :retrieved-
case :sub-relations))
                 :engine-compartment)
   :check-pc (equal (second (the-object (make-part 'cbr) :retrieved-
case :sub-relations))
                 :passenger-cabin)
   :check-lc (equal (defaulting (nth 2 (the-object (make-part
'cbr) :retrieved-case :sub-relations)))
                  :luggage-compartment)
   :sub-relations
   (let-streams
        ((aa (in (the :children)))
         (bb (the-object aa :name-for-display))
         (cc (collect! bb)))
      ((return-when empty? cc)))

   )

  :report-attributes
  (:vrml-out
  (write-vrml-file (list self)
  "f:/carbody.wrl"
  :specified-planarity 0.1
  :specified-linearity 0.1
  :camera-on? t))

  :parts
  ((engine-compartment
    :type (if (the :check-ec) 'ecgoal 'null-part)
    :position (:bottom 0.0))
   (passenger-cabin
    :type (if (the :check-pc) 'pcgoal 'null-part)
    :position (:front (:from (the :engine-compartment) (the :engine-
compartment :length))))
   (luggage-compartment
    :type (if (the :check-lc) 'lcgoal 'null-part)
    :position (:front (:from (the :passenger-cabin) (the :passenger-
cabin :length)) :bottom 0.0 ))
    )
  )
```

```
(defpart ecgoal (box)
  :attributes
  (:width (the-object (make-part 'cbr) :retrieved-case :engine-
compartment :width)
   :length (the-object (make-part 'cbr) :retrieved-case :engine-
compartment :length)
   :height (the-object (make-part 'cbr) :retrieved-case :engine-
compartment :height)
   :performance (list :provide_space_for_the_engine)
   :goals (list (the :G1-G1) (the :G3-G1))
   :G1-G1 (list :minimise :volume :c :G1)
   :G3-G1 (list :minimise :length :i :G3)
   :check-bonnet (let-streams
                 ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (c (equal
a3 :provide_top_cover_for_the_engine_compartment))
                  (d (collect-if! c (list c a2) nil   )))
               ((return-when empty? (first d))))
   :check-bonnet-2 (let-streams
                   ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                  (a1 (the-object a :type))
                  (a2 (defaulting (the-object a :performance)))
                  (b (equal a2
(list :provide_top_cover_for_the_engine_compartment)))
                  (c (collect-if! b a1)))
                 ((return-when empty? (first c))))
   :check-rightfrontwing (let-streams
                 ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (c (equal
a3 :provide_right_side_cover_for_the_engine_compartment))
                  (d (collect-if! c (list c a2) nil   )))
               ((return-when empty? (first d))))
   :check-rightfrontwing-2 (let-streams
                   ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                  (a1 (the-object a :type))
                  (a2 (defaulting (the-object a :performance)))
                  (b (equal a2
(list :provide_right_side_cover_for_the_engine_compartment)))
                  (c (collect-if! b a1)))
                   ((return-when empty? (first c))))
   :check-leftfrontwing (let-streams
                 ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (c (equal
a3 :provide_left_side_cover_for_the_engine_compartment))
                  (d (collect-if! c (list c a2) nil   )))
               ((return-when empty? (first d))))
   :check-leftfrontwing-2 (let-streams
                   ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                  (a1 (the-object a :type))
                  (a2 (defaulting (the-object a :performance)))
                  (b (equal a2
(list :provide_left_side_cover_for_the_engine_compartment)))
```

```
                  (c (collect-if! b a1)))
                   ((return-when empty? (first c))))
   :check-frontpanel (let-streams
                 ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                  (a2 (second a))
                  (a3 (nth 2 a))
                  (c (equal
a3 :provide_front_cover_for_the_engine_compartment))
                  (d (collect-if! c (list c a2) nil   )))
               ((return-when empty? (first d))))
   :check-frontpanel-2 (let-streams
                   ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                  (a1 (the-object a :type))
                  (a2 (defaulting (the-object a :performance)))
                  (b (equal a2
(list :provide_front_cover_for_the_engine_compartment)))
                  (c (collect-if! b a1)))
                 ((return-when empty? (first c))))
   :goal-oriented? (equal (the-object (make-part 'cbr) :adaptation-
method) :goal-oriented)
   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :name-for-display))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))

  :report-attributes
  (:vrml-out
  (write-vrml-file (list self)
  "f:/engine-compartment.wrl"
  :specified-planarity 0.1
  :specified-linearity 0.1
  :camera-on? t))

  :parts
  ((bonnet :type (if (and (the :goal-oriented?)
                   (equal (first (the :check-bonnet)) 't))
                (second (the :check-bonnet))
              (the :check-bonnet-2)))          ; top_surface
   (rightfrontwing :type (if (and (the :goal-oriented?)
                   (equal (first (the :check-rightfrontwing)) 't))
                (second (the :check-rightfrontwing))
              (the :check-rightfrontwing-2)))
   (leftfrontwing :type (if (and (the :goal-oriented?)
                   (equal (first (the :check-leftfrontwing)) 't))
                (second (the :check-leftfrontwing))
              (the :check-leftfrontwing-2)))
   (frontpanel  :type (if (and (the :goal-oriented?)
                   (equal (first (the :check-frontpanel)) 't))
                (second (the :check-frontpanel))
                (the :check-frontpanel-2)))))

(defpart pcgoal (box)
  :attributes
  (:width (the-object (make-part 'cbr) :retrieved-case :passenger-
cabin :width)
   :length (the-object (make-part 'cbr) :retrieved-case :passenger-
cabin :length)
```

```
   :height (the-object (make-part 'cbr) :retrieved-case :passenger-
cabin :height)
   :performance (list :provide_space_for_the_passengers)
   :goals (list (the :G1-G2) (the :G2-G1))
   :G1-G2 (list :minimise :volume :c :G1)
   :G2-G1 (list :maximise :volume :c :G2)
   :check-roofpanel (let-streams
                  ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                   (a2 (second a))
                   (a3 (nth 2 a))
                   (c (equal
a3 :provide_top_cover_for_the_passenger_cabin))
                   (d (collect-if! c (list c a2) nil   )))
                ((return-when empty? (first d))))
   :check-roofpanel-2 (let-streams
                    ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                   (a1 (the-object a :type)(the-object (make-part
'lcgoal) :leftrearwing))
                   (a2 (defaulting (the-object a :performance)))
                   (b (equal a2
(list :provide_top_cover_for_the_passenger_cabin )))
                   (c (collect-if! b a1)))
                ((return-when empty? (first c))))
   :check-rightside (let-streams
                  ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                   (a2 (second a))
                   (a3 (nth 2 a))
                   (c (equal
a3 :provide_right_side_cover_for_the_passenger_cabin))
                   (d (collect-if! c (list c a2) nil   )))
                ((return-when empty? (first d))))
   :check-rightside-2 (let-streams
                    ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                   (a1 (the-object a :type))
                   (a2 (defaulting (the-object a :performance)))
                   (b (equal a2
(list :provide_right_side_cover_for_the_passenger_cabin)))
                   (c (collect-if! b a1)))
                    ((return-when empty? (first c))))
   :check-leftside (let-streams
                  ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                   (a2 (second a))
                   (a3 (nth 2 a))
                   (c (equal
a3 :provide_left_side_cover_for_the_passenger_cabin))
                   (d (collect-if! c (list c a2) nil   )))
                ((return-when empty? (first d))))
   :check-leftside-2 (let-streams
                    ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                   (a1 (the-object a :type))
                   (a2 (defaulting (the-object a :performance)))
                   (b (equal a2
(list :provide_left_side_cover_for_the_passenger_cabin)))
                   (c (collect-if! b a1)))
                  ((return-when empty? (first c))))
   :goal-oriented? (equal (the-object (make-part 'cbr) :adaptation-
method) :goal-oriented)
```

```
    :sub-relations
    (let-streams
          ((aa (in (the :children)))
           (bb (the-object aa :name-for-display))
           (dd (list (the :name-for-display) bb))
           (cc (collect! dd)))
        ((return-when empty? cc))))

   :report-attributes
   (:vrml-out
   (write-vrml-file (list self)
   "f:/paasenger-cabin.wrl"
   :specified-planarity 0.1
   :specified-linearity 0.1
   :camera-on? t)
    )

   :parts
   ((roofpanel  :type (if (and (the :goal-oriented?)
                     (equal (first (the :check-roofpanel)) 't))
                  (second (the :check-roofpanel))
                (the :check-roofpanel-2)))
    (rightside  :type (if (and (the :goal-oriented?)
                     (equal (first (the :check-rightside)) 't))
                  (second (the :check-rightside))
                (the :check-rightside-2)))
    (leftside  :type (if (and (the :goal-oriented?)
                     (equal (first (the :check-leftside)) 't))
                  (second (the :check-leftside))
                  (the :check-leftside-2)))))

(defpart lcgoal (box)
  :attributes
  (:width (the-object (make-part 'cbr) :retrieved-case :luggage-
compartment :width)
   :length (the-object (make-part 'cbr) :retrieved-case :luggage-
compartment :length)
   :height (the-object (make-part 'cbr) :retrieved-case :luggage-
compartment :height)
   :performance (list :provide_space_for_the_luggage)
   :goals (list (the :G1-G3))
   :G1-G3 (list :minimise :volume :c :G1)
   :check-lcdoor (let-streams
                  ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                   (a2 (second a))
                   (a3 (nth 2 a))
                   (c (equal
a3 :provide_top_cover_for_the_luggage_compartment))
                   (d (collect-if! c (list c a2) nil   )))
                ((return-when empty? (first d))))
   :check-lcdoor-2 (let-streams
                    ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                   (a1 (the-object a :type))
                   (a2 (defaulting (the-object a :performance)))
                   (b (equal a2
(list :provide_top_cover_for_the_luggage_compartment)))
                   (c (collect-if! b a1)))
                ((return-when empty? (first c))))
   :check-rightrearwing (let-streams
```

```
                  ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                   (a2 (second a))
                   (a3 (nth 2 a))
                   (c (equal
a3 :provide_right_side_cover_for_the_luggage_compartment))
                   (d (collect-if! c (list c a2) nil   )))
                 ((return-when empty? (first d))))
   :check-rightrearwing-2 (let-streams
                    ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                   (a1 (the-object a :type))
                   (a2 (defaulting (the-object a :performance)))
                   (b (equal a2
(list :provide_right_side_cover_for_the_luggage_compartment)))
                   (c (collect-if! b a1)))
                    ((return-when empty? (first c))))
   :check-leftrearwing (let-streams
                  ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                   (a2 (second a))
                   (a3 (nth 2 a))
                   (c (equal
a3 :provide_left_side_cover_for_the_luggage_compartment))
                   (d (collect-if! c (list c a2) nil   )))
                 ((return-when empty? (first d))))
   :check-leftrearwing-2 (let-streams
                    ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                   (a1 (the-object a :type))
                   (a2 (defaulting (the-object a :performance)))
                   (b (equal a2
(list :provide_left_side_cover_for_the_luggage_compartment)))
                   (c (collect-if! b a1)))
                    ((return-when empty? (first c))))
   :check-rearpanel (let-streams
                  ((a (in (the-object (make-part 'cbr) :reform-rsp)))
                   (a2 (second a))
                   (a3 (nth 2 a))
                   (c (equal
a3 :provide_back_cover_for_the_luggage_compartment))
                   (d (collect-if! c (list c a2) nil   )))
                 ((return-when empty? (first d))))
   :check-rearpanel-2 (let-streams
                    ((a (in-tree (the-object (make-part
'cbr) :retrieved-case)))
                   (a1 (the-object a :type))
                   (a2 (defaulting (the-object a :performance)))
                   (b (equal a2
(list :provide_back_cover_for_the_luggage_compartment)))
                   (c (collect-if! b a1)))
                 ((return-when empty? (first c))))
   :goal-oriented? (equal (the-object (make-part 'cbr) :adaptation-
method) :goal-oriented)


   :sub-relations
   (let-streams
         ((aa (in (the :children)))
          (bb (the-object aa :name-for-display))
          (dd (list (the :name-for-display) bb))
          (cc (collect! dd)))
       ((return-when empty? cc))))
```

```
  :report-attributes
  (:vrml-out
  (write-vrml-file (list self)
  "f:/luggage-compartment.wrl"
  :specified-planarity 0.1
  :specified-linearity 0.1(the-object (make-part
'lcgoal) :leftrearwing)
  :camera-on? t)
  )

  :parts
  ((lcdoor  :type (if (and (the :goal-oriented?)
                   (equal (first (the :check-lcdoor)) 't))
                (second (the :check-lcdoor))
              (the :check-lcdoor-2)))
   (rightrearwing :type (if (and (the :goal-oriented?)
                   (equal (first (the :check-rightrearwing)) 't))
                (second (the :check-rightrearwing))
              (the :check-rightrearwing-2)))
   (leftrearwing :type (if (and (the :goal-oriented?)
                   (equal (first (the :check-leftrearwing)) 't))
                (second (the :check-leftrearwing))
              (the :check-leftrearwing-2)))
   (rearpanel :type (if (and (the :goal-oriented?)
                   (equal (first (the :check-rearpanel)) 't))
                (second (the :check-rearpanel))
                (the :check-rearpanel-2)))))
```

References

Agarwal, M. & Cagan, J. 2000. On the use of shape grammars as expert systems for geometry-based engineering design. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol.14, pp. 431-439.

Ahmed, S. 2005. Encouraging reuse of design knowledge: a method to index knowledge. Design studies, Vol.26, pp. 565-592.

Aleixos, N., Company, P. & Contero, M. 2004. Integrated modeling with top-down approach in subsidiary industries. Computers in Industry, Vol.53(1), pp. 97-116.

Al-Hakim, L., Kusiak, A. & Mathew, J. 2000. A graph-theoretic approach to conceptual design with functional perspectives. Computer-Aided Design, Vol.32, pp. 867-875.

Alisantoso, D., Khoo, L. P. & Lu, W. F. 2005. Early design analysis using rough set theory. 1st Intelligent Production Machines and Systems (I*PROMS) Virtual Conference pp. 243-248.

Al-Shihabi, T. & Zeid, I. 1998. A design-plan-oriented methodology for applying case-based adaptation to engineering design. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol.12, pp. 463-478.

Altshuller, G. S. 1969. Algorithm of invention. Moscow worker, Moscow.

Altshuller, G. S. 1984. Creativity as an exact science: The theory of the solution of inventive problems. Gordon and Breach Science Publishing, New York.

Andersson, K. 1993. A Vocabulary for Conceptual Design - Part of a Design Grammar. IFIP WG-5.3 Workshop on Formal Design Methods for Computer-Aided-Design Tallinn, Estonia. pp. 139-152.

Anthony, L., Regli, W. C., John, J. E. & Lombeyda, S. V. 2001. An approach to capturing structure, behavior, and function of artifacts in computer-aided design. Journal of Computing and Information Science in Engineering, Vol.1(2), pp. 186-192.

Arangarasan, R., Dani, T. H., Chu, C.-C., Liu, X. & Gadh, R. 2000. Geometric Modeling in Multi-Modal, Multi-Sensory Virtual Environment. Proceedings of 2000 NSF Design and Manufacturing Research Conference Vancouver, Canada.

Arciszewski, T., Michalski, R. & Wnek, J. 1995. Constructive induction: The key to design creativity. Third International Round-Table Conference on Computational Models of Creative Design Heron Island, Queensland, Australia. pp. 397-426.

Bergmann, R. & Wilke, W. 1995. Learning abstract planning cases. European Conference on Machine Learning 1995 Heraklion, Greece. pp. 55-76.

Bergmann, R. & Wilke, W. 1998. Towards a new formal model of transformational adaptation in case-based reasoning. Proceeding of the 13th European Conference on Artificial Intelligence 1998 pp. 53-57.

Bespalov, D., Regli, W. C. & Shokoufandeh, A. 2003. Reeb graph based shape retrieval for CAD. DETC'03, 2003 ASME Design Engineering Technical Conferences Chicago, Illinois, USA.

Bilgic, T. & Fox, M. S. 1996. Constraint-based retrieval of engineering design cases: Context as constraints. Artificial Intelligence in Design '96 Stanford University.

Kluwer Academic Publishers, pp. 269-288.

Bo, Y. & Salustri, F. A. 1999. Function modeling based on interactions of mass, energy and information. Proceedings of the 12th International Florida Artificial Intelligence Research Society Conference Sanibel Island, USA. pp. 384-388.

Borst, W. N. 1997. Construction of engineering ontologies for knowledge sharing and reuse. PhD Thesis. University of Twente, Enschede, Netherlands.

Bridge, D. G. 1998. Defining and combining symmetric and asymmetric similarity measures. Proceedings of the 4th European Workshop on Case-based Reasoning, Berlin. pp. 52-63.

Campbell, M. I., Cagan, J. & Kotovsky, K. 1999. A-Design: An agent-based approach to conceptual design in a dynamic environment. Research in Engineering Design, Vol.11(3).

Castano, S., Antonellis, V. D., Fugini, M. G. & Pernici, B. 1998. Conceptual schema analysis: techniques and applications. ACM Transactions on Database Systems 1998, Vol.23(3), pp. 286-333.

Chen, L., Song, Z. & Feng, L. 2004. Internet-enabled real-time collaborative assembly modeling via an e-Assembly system: status and promise. Computer-Aided Design, Vol.36, pp. 835-847.

Cicirello, V. & Regli, W. C. 2001. Machining feature-based comparisons of mechanical parts. SMI 2001: International Conference on Shape Modelling and Applications Genoa, Italy. IEEE Computer Society Press, pp. 176-185.

Coyne, R. D., Rosenman, M. A., Radford, A. D., Balachandran, M. & Gero, J. S. 1990.

Knowledge-based design systems. Addison-Wesley.

Cvetkovic, D. & Parmee, I. 2002. Agent-based support within an interactive evolutionary design system. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol.16, pp. 331-342.

Daniel, J., Medland, T. & Newnes, L. 2004. A study of methodologies for the design of medical devices. Proceedings of the 5th International Conference on Integrated Design and Manufacturing in Mechanical Engineering 2004 Bath, UK.

Deng, Y.-M. 2002. Function and behavior representation in conceptual mechanical design. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol.16, pp. 343-362.

Deng, Y.-M., Britton, G. A. & Tor, S. B. 2000. Constraint-based functional design verification for conceptual design. Computer-Aided Design, Vol.32, pp. 889-899.

Denkena, B., Woelk, P.-O. & Apitz, R. 2005. Knowledge-based manufacturing within the extended enterprise. Proceedings of the 1st Intelligent Production Machines and Systems (I*PROMS) Virtual International Conference Cardiff, UK. Elsevier, Oxford, pp. 105-110.

Dezfuli, M. R. D. 2001. A framework for value-based conceptual engineering design. PhD Thesis, Mechanical Engineering, University of Southern California, LA, USA.

Ding, L. & Gero, J. S. 2001. The emergence of the representation of style in design. Environment and Planning B: Planning and Design, Vol.28(5), pp. 707-731.

Dym, C. L. 1994. Engineering design: A synthesis of views. Cambridge University Press, Cambridge, UK.

Elinson, A. & Nau, D. S. 1997. Feature-based similarity assessment of solid models. 4th ACM Symposium on Solid Modeling and Applications Atlanta, Georgia. pp. 297-310.

El-Mehalawi, M. & Miller, R. A. 2003. A database system of mechanical components based on geometric and topological similarity. Part II: indexing, retrieval, matching, and similarity assessment. Computer-Aided Design, Vol.35(1), pp. 95-105.

Fiorentino, M., de Amicis, R., Monno, G. & Stork, A. 2002. Spacedesign: a mixed reality workspace for aesthetic industrial design. In Proceeding of the IEEE and ACM International Symposium on Mixed and Augmented Reality, pp. 86-96.

Francois, A. R. J. & Medioni, G. G. 1996. Generic shape learning and recognition. Object Representation in Computer Vision II Cambridge, U.K. pp. 287-320.

French, M. J. 1985. Conceptual design for engineers. The Design Council and Springer-Verlag, London, UK.

Fuh, J. Y. H. & Li, W. D. 2005. Advances in collaborative CAD: the-state-of-the art. Computer-Aided Design, Vol.37, pp. 571-581.

Gero, J. S. 1996. Computers and creative design. The Global Design Studio, National University of Singapore, pp. 11-19.

Gero, J. S. & Jupp, J. R. 2003. Feature-based qualitative representations of Architectural plans. Proceedings of the International Conference of the Association for Computer Aided Architectural Design Research in Asia 2003 Rangsit University, Bangkok. pp. 117-128.

Gero, J. S. & Kannengiesser, U. 2004. The situated function-behaviour-structure framework. Design Studies, Vol.25, pp. 373-391.

Gero, J. S. & Tsai, J.-H. 2004. Application of bond graph models to the representation of buildings and their use. Proceedings of the 9th International Conference of the Association for Computer Aided Architectural Design Research in Asia 2004 Seoul, Korea. pp. 373-385.

Gourashi, N. S. E.-D. 2003. Knowledge-based conceptual design of robot grippers. PhD Thesis, Department of System Engineering, Cardiff University, Cardiff.

Hayes, E. E. & Regli, W. C. 2001. Integrating design process knowledge with CAD models. DETC'2001 ASME Design Engineering Technical Conferences & Computers and Information in Engineering Conference Pittsburgh, Pennsylvania.

Hilaga, M., Shinagawa, Y., Kohmura, T. & Kunii, T. 2001. Topology matching for fully automatic similarity estimation of 3D shapes. SIGGRAPH 2001 Los Angeles, USA. pp. 203-212.

Horvath, I., Vergeest, J. S. M. & Kuczogi, G. 1998. Development and application of design concept ontologies for contextual conceptualization. Proceedings of 1998 ASME Design Engineering Technical Conferences Atlanta, Georgia. ASME, New York.

Hsu, W. & Liu, B. 2000. Conceptual design: issues and challenges. Computer-Aided Design, Vol.32, pp. 849-850.

Hsu, W. & Woon, I. M. Y. 1998. Current and future research in the conceptual design of mechanical products. Computer-Aided Design, Vol.30(5), pp. 377-389.

Hu, X., Pang, J., Pang, Y., Atwood, M., Sun, W. & Regli, W. C. 2000. A survey of design rationale systems: Approaches, representation, capture and retrieval. Engineering with Computers, Vol.16(3-4), pp. 209-235.

Huang, G. Q., Lee, S. W. & Mak, K. L. 2003. Collaborative product definition on the Internet: a case study. Journal of Materials Processing Technology, Vol.6721, pp. 1-7.

Hurst, K. S. 1999. Engineering design principles. Arnold, London.

Iyer, N., Lou, K., Jayanti, S., Kalyanaraman, Y. & Ramani, K. 2003. Early results from 3DESS: A 3D engineering shape search system. International Symposium on Product Lifecycle Management (PLM 03), Indian Institute of Science, Bangalore, India.

Jarmulak, J., Craw, S. & Rowe, R. 2001. Using Case-Base Data to Learn Adaptation Knowledge for Design. Proceedings of the 17th IJCAI Conference pp. 1011-1016.

Jupp, J. R. & Gero, J. S. 2004. Qualitative representation and reasoning in design: A hierarchy of shape and spatial languages. Visual and Spatial Reasoning in Design III pp. 139-162.

Juster, N. P., Maxfield, J., Dew, P. M., Taylor, S., Fitchie, M., Ion, W. J., Zhao, J. & Thompson, M. 2001. Predicting product aesthetic quality using virtual environments. Journal of Computing and Information Science in Engineering, Vol.1, pp. 105-112.

Keane, M. T., Smyth, B. & O'Sullivan, J. 2001. Dynamic similarity: A processing perspective on similarity. In: Hahn, M.U. ed. Similarity & Categorisation. Oxford: Oxford University Press, pp. 179-192.

Kicinger, R. 2004. Emergent engineering design: Design creativity and optimality

inspired by nature. School of Information Technology and Engineering, George Mason University, Fairfax, VA, USA.

Kicinger, R., Arciszewski, T. & Jong, K. D. 2005a. Emergent Designer: An integrated research and design support tool based on models of complex systems. International Journal of Information Technology in Construction, Vol.10, pp. 329-347.

Kicinger, R., Arciszewski, T. & Jong, K. D. 2005b. Evolutionary computation and structural design: A survey of the state-of-the-art. Computers and Structures, Vol.83, pp. 1943-1978.

Kicinger, R., Arciszewski, T. & Jong, K. D. 2005c. Generative design in structural engineering. 2005 ASCE International Conference on Computing in Civil Engineering, Cancun, Mexico. American Society of Civil Engineers Press, Reston, VA.

Kicinger, R., Arciszewski, T. & Jong, K. D. 2005d. Parameterized versus Generative Representations in Structural Design: An Empirical Comparison. Genetic and Evolutionary Computation Conference Washington, DC.

Kima, K.-Y., Wang, Y., Muogboh, O. S. & Nnaji, B. O. 2004. Design formalism for collaborative assembly design. Computer-Aided Design, Vol.36, pp. 849-871.

Kitamura, Y. & Mizoguchi, R. 2003. Ontology-based description of functional design knowledge and its use in a functional way server. Expert Systems with Applications, Vol.24, pp. 153-166.

Kroll, E., Condoor, S. S. & Jansson, D. G. 2001. Innovative conceptual design. Cambridge University Press, Cambridge, UK.

KTI 2001. The ICAD System (Release 8.0.0) user's manual. Lexington, MA, USA.

Le, S. Q., Ho, T. B. & Phan, T. T. H. 2004. A Novel Graph-Based Similarity Measure for 2D Chemical Structures. Genome Informatics, Vol.15(2), pp. 82-91.

Leake, D. B., Birnbaum, L., Hammond, K., Marlow, C. & Yang, H. 1999. Integrating information resources A case study of engineering design support. International Conference on Case-Based Reasoning 1999 Munich, Germany.

Leake, D. B. & Sooriamurthi, R. 2002. Automatically selecting strategies for Multi-Case-Base Reasoning. European Conference on Case-Based Reasoning 2002: Advances in Case-Based Reasoning Berlin. Springer Verlag, pp. 204-233.

Leake, D. B. & Sooriamurthi, R. 2003. Dispatching cases versus merging case-bases: when MCBR matters. Proceedings of the 16th International Florida Artificial Intelligence Research Society Conference (FLAIRS-2003) St. Augustine, Florida. pp. 129-133.

Lee, J. 1997. Design rationale systems: Understanding the issues. AI in Design, IEEE. Expert, pp. 78-85.

Liu, Y.-C., Bligh, T. & Chakrabarti, A. 2003. Towards an 'ideal' approach for concept generation. Design Studies, Vol.24, pp. 341-355.

Maher, M. L., Balachandran, M. B. & Zhang, D. M. 1995. Case-based reasoning in design. Lawrence Erlbaum, Mahwah, New Jersey.

Mann, D. L. 2002. Hands-on systematic innovation. CREAX Press, Belgium.

McCormack, J. P., Cagan, J. & Vogel, C. M. 2004. Speaking the Buick language: capturing, understanding, and exploring brand identity with shape grammars. Design

Studies, Vol.25, pp. 1-29.

Nakakoji, K., Gross, M. D., Candy, L. & Edmonds, E. 2001. Tools, conceptual frameworks, and empirical studies for early stages of design. Conference on Human Factors in Computing Systems archive, CHI '01 extended abstracts on Human factors in computing systems Seattle, Washington. pp. 493-494.

Ng, K. W. 2006. A critical analysis of current engineering design decision making perspective. Proceedings of the 2nd Intelligent Production Machines and Systems (I*PROMS) Virtual International Conference Cardiff, UK.

Noy, N. F. & McGuinness, D. L. 2001. Ontology development 101: A guide to creating your first ontology. Stanford University, Stanford, CA, Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880.

Ohbuchi, R., Otagiri, T., Ibato, M. & Takei, T. 2002. Shape-similarity search of three-dimensional models using parameterized statistics. Pacific Graphics 2002 Beijing, China. pp. 265-275.

O'Sullivan, B. A. 1999. Constraint-aided conceptual design. PhD Thesis, Department of Computer science, University College Cork, Ireland.

O'Sullivan, B. A. 2002. Constraint-aided conceptual design. Professional Engineering Publishing Limited, London and Bury St Edmunds, UK.

Pahl, G. & Beitz, W. 1996. Engineering design: A systematic approach. The Design Council and Springer Verlag, London.

Papadopoulos, A. N. & Manolopoulos, Y. 1999. Structure-based similarity search with

graph histograms. DEXA/IWOSS International Workshop on Similarity Search Florence, Italy. pp. 174-178.

Pham, D. T., Gourashi, N. S. E.-D. & Eldukhri, E. E. 2005. Intelligent conceptual design of robot grippers for assembly tasks. Proceedings of the 1st Intelligent Production Machines and Systems (I*PROMS) Virtual Conference Cardiff, UK. pp. 321-326.

Pham, D. T. & Liu, H. 2006. A new TRIZ-based approach to design concept generation. Proceedings of the 4th International Conference on Manufacturing Research Liverpool. pp. 371-376.

Pham, D. T., Liu, H. & Dimov, S. 2006. An I-Ching-TRIZ inspired tool for retrieving conceptual design solutions. Proceedings of the 2nd Intelligent Production Machines and Systems (I*PROMS) Virtual International Conference Cardiff, UK.

Preston, M. & Mehandjiev, N. 2004. A framework for classifying intelligent design theories. 2004 ACM workshop on Interdisciplinary software engineering research Newport Beach, CA, USA. pp. 49-54.

Purvis, L. & Pu, P. 1995. Adaptation using constraint satisfaction techniques. Proceedings of the 1st International Conference on Case Based Reasoning, Berlin LNAI Series, Springer Verlag, pp. 289-300.

Qian, L. & Gero, J. S. 1996. Function-behaviour-structure paths and their role in analogy-based design. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol.10, pp. 289-312.

Qin, X. & Regli, W. C. 2000. Applying case-based reasoning to mechanical bearing design. DETC'00 2000 ASME Design Engineering Technical Conferences Baltimore,

Maryland.

Qin, X. & Regli, W. C. 2003. A study in applying case-based reasoning to engineering design: Mechanical bearing design. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol.17, pp. 235-252.

Renner, G. & Ekart, A. 2003. Genetic algorithms in computer aided design. Computer-Aided Design, Vol.35, pp. 709-726.

Rivard, H. & Fenves, S. J. 2000. SEED-Config: A case-based reasoning system for conceptual building design. Artificial Intelligence for Engineering Design, Analysis and Manufacturing . Vol.14, pp. 415-430.

Roozenburg, N. F. M. & Eekels, J. 1995. Product design: Fundamentals and methods. John Wiley & Sons, Chichester.

Rosenman, M. 2000. Case-based evolutionary design. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol.14, pp. 17-29.

Ryu, K. and Jung, M. 2003. Agent-based fractal architecture and modelling for developing distributed manufacturing systems. International Journal of Production Research, Vol.41, pp. 4233-4255.

Ryu, K., Son, Y. & Jung, M. 2003. Modeling and specifications of dynamic agents in fractal manufacturing systems. Computers in Industry, Vol.52, pp. 161-182.

Salustri, F. A. 1996. A formal theory for knowledge-based product model representation. Knowledge-Intensive CAD II: Proc. IFIP WG5.2 Workshop London. Chapman & Hall, pp. 59-78.

Salustri, F. A. 2000. Towards a logical framework for engineering design processes. 4th IFIP TC5 WG5.2 Workshop on Knowledge Intensive CAD Italy. pp. 201-212.

Salustri, F. A. 2001. Using design patterns to promote multidisciplinary design. Proceedings of the 2001 CSME International Conference on Multi-disciplinary Design Engineering Montreal, Canada.

Salustri, F. A. 2002. Mereotopology for product modelling: A new framework for product modelling based on logic. Journal of Design Research, Vol.2(1).

Salustri, F. A. 2005. Using pattern languages in design engineering. ICED 2005 (CD-ROM) Melbourne, Australia.

Salustri, F. A. & Parmar, J. 2003. Visualising early product design information with enhanced concept maps. 2003 International Conference on Engineering Design Stockholm.

Saridakis, K. M., Dentsoras, A. J., Radel, P. A., Saridakis, V. G. & Exintari, N. V. 2006. Neuro-fuzzy case-based design: An application in structural design. Proceedings of the 2nd Intelligent Production Machines and Systems (I*PROMS) Virtual International Conference Cardiff, UK.

Savransky, S. D. 2000. Engineering of creativity: Introduction to TRIZ methodology of inventive problem solving. CRC Press, Boca Raton.

Seebohm, T. & Wallace, W. 1998. Rule-based representation of design in architectural practice. Automation in Construction, Vol.8, pp. 73-85.

Seo, K., Goodman, E. D. & Rosenberg, R. C. 2005. Design of Air Pump System Using Bond Graph and Genetic Programming Method. Proceedings of the 2005

conference on Genetic and evolutionary computation Washington DC, USA. pp. 2215-2216.

Sequin, C. H. 2005. CAD tools for aesthetic engineering. Computer-Aided Design, Vol.37, pp. 737-750.

Setchi, R. M., Lagos, N. & Dimov, S. S. 2005. Semantic modelling of product support knowledge. Proceedings of the 1st Intelligent Production Machines and Systems (I*PROMS) Virtual Conference Cardiff, UK. pp. 275-280.

Sharma, M., Raja, V. & Fernando, T. 2006. Collaborative design review in a distributed environment. the 2nd Intelligent Production Machines and Systems (I*PROMS) Vritual International Conference Cardiff, UK.

Shyamsundar, N. & Gadh, R. 2002. Collaborative virtual prototyping of product assemblies over the Internet. Computer-Aided Design, Vol.34, pp. 755-768.

Smyth, B. & Cunningham, P. 1992. Deja Vu: A hierarchical case-based reasoning system for software design. Proceedings of the 10th European Conference on Artificial Intelligence Vienna, Austria. pp. 587-589.

Smyth, B. & Keane, M. T. 1998. Adaptation-guided retrieval: questioning the similarity assumption in reasoning. Artificial Intelligence, Vol.104, pp. 1-45.

Sqalli, M. H., Purvis, L. & Freuder, E. C. 1999. Survey of applications integrating constraint satisfaction and case-based reasoning. Proceedings of the 1st International Conference and Exhibition on the Practical Application of Constraint Technologies and Logic Programming London.

Suh, N. P. 1990. The principles of design. Oxford University Press, New York.

Sycara, K. P. 1998. Multiagent systems. AI magazine, Vol.19, pp. 79-92.

Szykman, S., Sriram, R. D. & Regli, W. C. 2001. The role of knowledge in next-generation product development systems. Journal of computation and information Science in Engineering, Vol.1(1), pp. 3-11.

Tirpak, T. M., Daniel, S. M., LaLonde, J. D. & Davis, W. J. 1992. A note on a fractal architecture for modelling and controlling flexible manufacturing systems. IEEE Transactions on Systems, Man,and Cybernetics, Vol.22(3), pp. 564-567.

Umeda, Y. & Tomiyama, T. 1997. Functional reasoning in design. IEEE Expert, Vol.12(2), pp. 42-48.

Vancza, J. 1999. Artificial intelligence support in design: a survey. CIRP international design seminar Dordrecht, Kluwer. pp. 57-68.

Waheed, A. & Adeli, H. 2005. Case-based reasoning in steel bridge engineering. Knowledge-Based Systems, Vol.18, pp. 37-46.

Wang, C., Horvath, I. & Vergeest, J. S. M. 2002a. Towards the reuse of shape information in CAD. Tools and Methods of Competitive Engineering 2002, Wuhan, China. pp. 103-116.

Wang, L., Shen, W., Xie, H., Neelamkavil, J. & Pardasani, A. 2002b. Collaborative conceptual design-state of the art and future trends. Computer-Aided Design, Vol.34(113), pp. 981-996.

Warnecke, H.-J. 1993. The fractal company. Springer-Verlag, Germany.

White, S. A. 1995. A framework for the development of domain specific design support systems. 1st World Conference on Integrated Design and Process Technology Austin, Texas, USA. pp. 27-42.

Zavbi, R. & Duhovnik, J. 2000. Conceptual design of technical systems using functions and physical laws. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol.14, pp. 69-83.

Zha, X. F. & Du, H. 2001. Mechanical systems and assemblies modeling using knowledge-intensive Petri nets formalisms. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol.15, pp. 145-171.