A distributed system structure for modular product architecture development and variation.

Shan. Bai

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/599

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000 ext. 3208.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600
A DISTRIBUTED SYSTEM STRUCTURE FOR MODULAR
PRODUCT ARCHITECTURE DEVELOPMENT AND VARIATION

by

Shan Bai

A Thesis
submitted to the Faculty of Graduate Studies and Research
through Industrial & Manufacturing Systems Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science
at the University of Windsor

Windsor, Ontario, Canada
2001
© 2001 Shan Bai
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
ABSTRACT

Customer need is variable and unstable. It confronts the manufacturing industry with increasing demand for needed variety with limited product design budget. A successful way to offer the needed variety while reducing the product design cost is to launch modular product families. Determining and controlling product architecture is one of the key activities during product family design or redesign. This research will present a systematic method for developing product architecture and configuring varieties, in which two linear programming models will be introduced to identify feasible sub-functions and configure product varieties. Then, a simple example is given to demonstrate the method.

Based on the methodology, a structure of distributed design support system will be formulated using Object-Oriented technology with Unified Modeling Language (UML). This structure will enable the automation of modular product family development and variant configuration in enterprise level.
ACKNOWLEDGEMENTS

I am deeply indebted to my advisor, Dr. M. Wang. His willingness to help when I needed it and to let me work independently at other times was greatly appreciated.

Also, I would like to thank Dr. D. Kao, Dr. S. Taboun, and Dr. W.H. ElMaraghy for their help and guidance throughout this research.

Finally, I want to express my appreciation to my wife, Tian, for her encouragement throughout this work.
3.2.1 Determine Independent Sub-function Ratings for Each Concept ..... 31
3.2.2 Compute Dependent Sub-function Ratings for Product Family ..... 33
3.2.3 Identify the Feasible Sub-functions ........................................... 37
3.2.4 Construct Generic Product Architecture ...................................... 39
3.2.5 Configure Product Variety ....................................................... 41

CHAPTER 4  CASE STUDY – Power Screwdriver Design ................... 49
4.1 Determine Independent Sub-function Ratings for Each Concept .......... 49
4.2 Compute Dependent Sub-function Ratings for Product Family .............. 53
4.3 Identify the Feasible Sub-functions .............................................. 57
4.4 Construct Generic Product Architecture ........................................ 60
4.5 Configure Product Variety ....................................................... 64

CHAPTER 5  RELEVANT SOFTWARE TECHNOLOGIES ..................... 67
5.1 Introduction of Distributed System .............................................. 67
5.2 System Overview ............................................................... 71
5.3 Object Oriented Methodology and UML .................................... 72

CHAPTER 6  CONCLUSION ........................................................ 78
6.1 Achievements and Purpose ..................................................... 78
6.2 Sensitivity Analysis .............................................................. 79
6.3 Future Work ................................................................. 82
<table>
<thead>
<tr>
<th>APPENDIX</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>USE CASE DIAGRAM</td>
<td>84</td>
</tr>
<tr>
<td>2</td>
<td>CLASS DIAGRAM</td>
<td>86</td>
</tr>
<tr>
<td>3</td>
<td>CLASS AND INTERFACE DEFINITION</td>
<td>87</td>
</tr>
<tr>
<td>4</td>
<td>COLLABORATION DIAGRAM</td>
<td>93</td>
</tr>
<tr>
<td>5</td>
<td>SEQUENCE DIAGRAM</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>VITA AUCTORIS</td>
<td>104</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

Table 1-1 Four Mapping Types ................................................................. 6
Table 1-2 Reasons for Dividing Products into Modules ................................. 7
Table 2-1 Comparison of Integral Architecture to Modular Architecture .......... 11
Table 2-2 Steps of Module Identification .................................................... 14
Table 2-3 Market-based Architecture Selection Process ................................. 16
Table 2-4 Overview of Product Architecture Design Methodology .................... 17
Table 3-1 Overview of the Modular Design Process ....................................... 30
Table 4-1 Customer Need Vector of Screwdriver Concept 1 ............................ 49
Table 4-2 Function List of Screwdriver Concept 1 ......................................... 51
Table 4-3 Customer Need-function Matrix of Screwdriver Concept 1 ................. 51
Table 4-4 Function List of the Screwdriver Family ........................................ 52
Table 4-5 Function-product Matrix of the Screwdriver Family ......................... 54
Table 4-6 Normalized Function-product Matrix of the Screwdriver Family .......... 55
Table 4-7 Correlation Weights among Sub-functions .................................... 55
Table 4-8 Design Costs of Sub-functions ................................................... 57
Table 4-9 Feasible Function List of the Screwdriver Family ............................ 59
Table 4-10 Module List of the Screwdriver Family ........................................ 61
Table 4-11 Cost of Module Instance ........................................................... 63
Table 4-12 Instance-performance Correlation Weight .................................... 63
Table 5-1 Advantages of Distributed System ............................................... 67
LIST OF FIGURES

Figure 1-1 Mapping in Design Process ........................................ 5
Figure 2-1 Dominant Flow Heuristic Applied to a Function Structure .......... 18
Figure 2-2 Flow Branching Heuristic Applied to a Function Structure .......... 19
Figure 2-3 Conversion-transmission Applied to a Set of Sub-functions .......... 20
Figure 2-4 Mapping among Three Views of Product Design ...................... 22
Figure 3-1 Elements of Product Family ........................................ 29
Figure 4-1 Three Power Screwdriver Concepts .................................. 50
Figure 4-2 Function Vectors of Three Screwdriver Concepts .................. 53
Figure 4-3 Function Structure for the Screwdriver Family ...................... 59
Figure 4-4 Mapping Sub-functions to Modules .................................. 60
Figure 4-5 Modules of the Power Screwdriver .................................. 62
Figure 4-6 Product Design Specification ....................................... 64
Figure 5-1 A Three-tiered Distributed Application .............................. 68
Figure 5-2 System Flow Chart .................................................. 71
Figure 5-3 Distributed System Architecture ................................... 72
Figure 5-4 Association Relationship .......................................... 73
Figure 5-5 Generalize Relationship .......................................... 74
Figure 5-6 Dependency Relationship .......................................... 75
Figure 5-7 Aggregate Relationship .......................................... 75
Figure 5-8 Realize Relationship ........................................... 76
1. **Basic function**: the function upon which all other functions (secondary function or sub-function) depend. It is at the top level of a function tree.

2. **Class diagram**: is a diagram that shows a set of classes, interfaces, and collaborations and their relationships.

3. **Collaboration diagram**: is one type of interaction diagram. It emphasizes the organization of the objects that participate in an interaction.

4. **Cost vector (Q)**: a vector indicating design costs committed to the sub-functions.

5. **Customer need vector (CN)**: a vector indicating the weight of each customer need.

6. **Customer need-function matrix (CF)**: to answer the question that how well each sub-function meets the customer needs.

7. **Family vector (P)**: a vector indicating normalized customer-need ratings for each sub-function of the product family.
8. **Feasible sub-function vector** (**H**): a vector indicating the feasible sub-functions of the product family.

9. **Function vector** (**F**): a vector indicating a product’s functionality and customer need ratings for each sub-function.

10. **Function-product matrix** (**FP**): a collection of function vectors which forms a \( m \times n \) matrix, where \( m \) is the total number of distinct sub-functions and \( n \) is the number of products.

11. **Instance-performance matrix** (**IP**): to indicate the inter-relationships between module instance and the performance.

12. **Instance vector** (**M_i**): a vector indicating the available instances of the \( i^{th} \) module.

13. **Module vector** (**M**): a vector indicating the available modules of the product family.

14. **Performance vector** (**G**): a vector indicating importance weight of each performance.

15. **Product variety**: the diversity of products that a production system provides to the marketplace.
16. **Product family**: a group of products that are similar in functions and customer needs.

17. **Product architecture**: is the scheme by which the functions of a product are allocated to physical components.

18. **Sequence diagram**: is one type of interaction diagram, emphasizing the time ordering of messages.

19. **Sub-function (secondary function)**: all functions at lower level of the hierarchical tree are secondary functions.

20. **Typology**: A discourse or treatise on types. A typology of architectures provides a vocabulary for discussing the implications of the choice of architecture on the performance of the manufacture firm.

21. **Use case diagram**: is a diagram that shows a set of use cases and actors and their relationships.
CHAPTER 1 INTRODUCTION

1.1 Background

Product variety has emerged as an important element of manufacturing competitiveness based on customer needs. It can be defined as the diversity of products that a production system provides to the marketplace. The variation may be in terms of the sets of functional elements implemented by the product, or in terms of the specific performance characteristics of the product related to a particular functional element.

Today, manufacturing companies keep developing a large variety of products to satisfy customer needs. There are some common problems for designers in developing product varieties:

♦ How to balance the need for higher variety and complexity of products with an accompanying reduction of development cost?
♦ How to reduce product variety development time?
♦ How to ease engineering changes?

Establishing a product family is a means to improve the commercial variety while limiting the development, manufacturing and servicing efforts by sharing a set of common elements, interfaces, or subsystems. Product family is a group of products that are similar in functions and customer needs. The products of a product family may share a common platform but have specific features and
functions required by different sets of customers. There are some examples of product family development in the follows:

(1) Aircraft

Aircraft manufacturers such as Boeing and Airbus Industries use common wings and nose and tail components to leverage many models by using different fuselage modules to create crafts of different lengths and passenger/ freight capacities. Boeings’ 757 and 767 are each design families (i.e., there is a base series and derived series with more seats, or extended range versions, or both). Although they are quite different types of aircraft, Boeing capitalized on the potential familiarity in avionics between the two types. Avionics is about one third of total development cost. Hence, the avionics commonality offered welcome economics of scale in technical development and in production.

(2) Automobiles

The automotive industry provides several examples of how to utilize one of the most expensive parts of a car, i.e., the platform, in a whole range of different models. Some argue that the development of the platform can be as expensive as up to 60% of total development cost. Therefore, the trend to increase the number of models from newly developed platforms seems to be strong. Honda’s advanced form of platform strategy has enabled them to create four unique models (Civic, Del Sol, Domani, and Integra) from their Accord platform within 2 years of its introduction.

(3) Personal Computers

With the ThinkPad PC product line, IBM provides a good example of the strategy
of using common parts and the same basic industrial design concept to develop unique end products. IBM had traditionally pursued a system-oriented strategy and emphasis had been placed on the integrity of office systems as a whole rather than the unique features of individual products. Nevertheless, the ThinkPad PC is reported to be a success. The new IBM PCs are now developed every 6 months instead of 12 to 18 months in 1992.

(4) Consumer Electronics

Sony HandyCam™ evolved from a basic and common architecture to become a notable commercial success. The first product, M8, specified the basic product architecture and interfaces serving as a platform for four additional models that were introduced into the market within 26 months. This advantageous evolution was made possible by the intelligent design of the product architecture, mainly the platform. Sony created and dominated the market for personal stereos (the Sony Walkman) with a worldwide market share around 40% for over a decade. Based on only four technological platforms, Sony created almost 250 models in the U.S. market during the 1980s. Although most of the models were created by changes in features, packaging and appearance, the platform always served as a basis for the new models.

As a starting point of product family development, determining generic product architecture is a key activity. Product architecture is the scheme by which the functions of a product are allocated to physical components. Ulrich (1995), defined product architecture more precisely as:
(1) The arrangement of functional elements

(2) The mapping from functional elements to physical components

(3) The specification of the interfaces among interacting physical components

There are two basic methods to create product architecture. One method is to use an integral platform. This is a monolithic part of the product that will be shared by all the products in the family. Given that platform, a development team then adds an individually designed portion to the product to create a finished variant design. The other method is to use modular architecture. In this case, the product is divided into modules that can be swapped by others of different size or functionality to create product varieties. With a modular architecture, a product variety can be configured by a set of modules.

1.2 Modular Design

Design involves a continuous interplay between what we want to achieve and how we want to achieve it. The design's objectives must be determined by defining it in terms of specific requirements called function requirements (FRs). Then, to satisfy these requirements, a physical embodiment characterized in terms of design parameters (DPs) must be created. The design process involves relating these FRs of function domain to the DPs of physical domain at every hierarchical level, it begins with the establishment of FRs in the functional domain to satisfy a given set of needs, and ends with the creation of an entity that satisfies these FRs.
There are two basic design methodologies - integral design and modular design. Modularity corresponds on flexibility and changeability. Integration corresponds to stability and optimization. According to Ulrich, and Tung (1991), modularity can be viewed as depending two characteristics of the design:

(1) Similarity between the physical and functional architecture of the design.

(2) Minimization of incidental interactions between physical components.

Erens, and Verhulst (1997) defined “a modular design to be a design in which a restricted number of functions is allocated to a module, or a restricted number of modules is allocated to a physical assembly”. They also analyzed four different possibilities of allocation process from the functional domain to the technology domain, which are given in Table 1-1.

Finally, they asserted that both one-to-many and many-to-many mappings are ambiguous, as it is not clear which part of function is realized in a technology module. Therefore, mappings must always be based on one-to-one or many-to-
one mappings in modular design. This concept is used as the premise for modularity in this research.

<table>
<thead>
<tr>
<th>Mapping type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-to-one</td>
<td>One function is allocated to one technology module. This is a completely modular design.</td>
</tr>
<tr>
<td>One-to-many</td>
<td>One function is mapping to several technology modules. This distribution of a function over several modules results in an integrated design on that level.</td>
</tr>
<tr>
<td>Many-to-one</td>
<td>Several functions are allocated to one technology module. Again, function sharing increases the level of integration.</td>
</tr>
<tr>
<td>Many-to-many</td>
<td>Several functions are allocated to several technology modules. Functions are distributed and shared, thereby further increasing the level of integration.</td>
</tr>
</tbody>
</table>

There are many advantages of modular design. Established companies usually have many modules already designed for previous products that could be reused, as well as the resources to design new versions of the same modules or modules with new functionality. In addition, there exists the possibility of purchasing modules from existing catalogs, or even outsourcing the design of new ones. Therefore, modular product architecture designs can result in economies of scale from producing larger volumes of the same modules, lower design costs from not having to redesign similar subsystems, and more flexible product variant configuration. Erixon, and Ostgren (1993) grouped the reasons for product models into five main areas:
Table 1-2: Reasons for Dividing Products into Modules

<table>
<thead>
<tr>
<th>Areas</th>
<th>Reasons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>? Parallel design of modules, reduction of development time.</td>
</tr>
<tr>
<td></td>
<td>? Selecting part of a product as a module due to future foreseen technology development.</td>
</tr>
<tr>
<td></td>
<td>? Simplified product planning.</td>
</tr>
<tr>
<td>Manufacture</td>
<td>? Common modules giving high volume and thus scale of economy advantages.</td>
</tr>
<tr>
<td></td>
<td>? Utilization of investments in special manufacturing processes</td>
</tr>
<tr>
<td></td>
<td>? Decreased rework by testing modules.</td>
</tr>
<tr>
<td></td>
<td>? Possibility for good work organization.</td>
</tr>
<tr>
<td>Product variants</td>
<td>? Possibility to adapt products for different markets by having some modules as “variant” modules.</td>
</tr>
<tr>
<td>Purchasing</td>
<td>? Supplier offers a system (module), which is cheaper than making it in-house.</td>
</tr>
<tr>
<td>After sale</td>
<td>? Possibilities for upgrading.</td>
</tr>
<tr>
<td></td>
<td>? Simplified maintenance / services.</td>
</tr>
<tr>
<td></td>
<td>? Possibility to rebuild a product.</td>
</tr>
<tr>
<td></td>
<td>? Modules make disassembly easier.</td>
</tr>
</tbody>
</table>

1.3 Objective

The main issues associated with modular product design include module creation and module selection. Modular product architecture development corresponds to module creation and variant configuration corresponds to module selection based on the generic modular product architecture. Few of current research community works try to present a systematic method to support modular product architecture development and variation, and integrate it into an distributed system with which the product data can be shared and reused by engineers in enterprise level.
This research intends to synthesize fragments of existing theory and knowledge related to modular design into a systematic method for developing product architecture and configuring variety, in which two optimization models will be introduced to identify feasible sub-functions and configure product varieties. Finally, a distributed system structure will be formulated based on this method using Unified Modeling Language (UML) to illuminate the practice of collaboration modular product design, and it can also be integrated into Product Data Management (PDM) system to fulfill the requirement of robust modular design in enterprise level.

1.4 Boundary
The life cycle of engineering design includes: feasible study, concept design, embodiment design, detail design, production, distribution, consumption, and retirement.

This research mainly discusses some modular product design issues from conceptual design to embodiment design phase, which does not consider defining the product geometry, product size, and choosing materials in detail design phase. Therefore, the boundary of this research is from product conceptual design to embodiment design, and assumes that modular product architecture will be adopted in product developing. Not all the aspects of modular product design problem will be covered in this research, but it can aid for design team deliberations at least.
1.5 Organization

This thesis is organized into six chapters. The second chapter, literature review, introduces some previous researches related to modular product architecture design and product variation. These previous works will be used as reference or theory basis for this research.

The third chapter gives the primary methodology and algorithm for developing generic modular product architecture and configuring product varieties based on the architecture.

Chapter 4 introduces a real example of screwdriver development that follows the methodology and algorithm given in the previous chapter.

Chapter 5 deals with distributed system, Object-Oriented methodology and Unified Modeling Language. These concepts and technologies are used for designing a distributed system structure, which can enable the automation of modular product design in enterprise level.

In the last chapter, some discussion on the sensitivity analysis, achievements and future work of this research will be given.
CHAPTER 2 LITERATURE REVIEW

2.1 Literature Review

This section presents the review of relevant literatures about product architecture development and product variant configuration. These previous works provide a fundament for this research.

The concepts related to modular design, product architecture, and variation are well presented in some previous researches. Ulrich, and Tung (1991) defined modularity, explored some benefits and costs of modularity, and classified the ways in which modularity can be used to standardize components and create product variety. They also asserted that a completely modular design embodies a one-to-one correspondence between each functional element of the design and a single physical component.

Ulrich (1995) intensively analyzed the role of product architecture in product design, and mentioned its effects to product variety. He defined the concept of product architecture, provided a typology of product architectures, and articulated the potential linkages between the architecture of the product and five areas of managerial importance: (1) product change; (2) product variety; (3) component standardization; (4) product performance; (5) product development management. He also compared the integral architecture and modular architecture, and gave the summary in terms of the five areas as follows:
### Table 2-1: Comparison of Integral Architecture to Modular Architecture

<table>
<thead>
<tr>
<th></th>
<th>Integral architecture</th>
<th>Modular architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Product change</strong></td>
<td>? Any change in functionality requires a change to several components.</td>
<td>? Functional changes can be made to a product in the field.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>? Manufacturers can change the function of subsequent model generations by changing a single component.</td>
</tr>
<tr>
<td><strong>Product variety</strong></td>
<td>? Variety not feasible without flexible component production processes.</td>
<td>? Products can be assembled in a combinatorial fashion from a relatively small set of component building blocks to create variety.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>? Variety possible even without flexible component production processes.</td>
</tr>
<tr>
<td><strong>Component standardization</strong></td>
<td>? none</td>
<td>? Components can be standardized across a product line.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>? Firms can use standard components provided by suppliers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>? Interfaces may adhere to an industry standard.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>? Decoupling interfaces may require additional mass and space.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>? Potentially result in physical redundancy.</td>
</tr>
<tr>
<td><strong>Product development management</strong></td>
<td>? Requires tight coordination of design tasks</td>
<td>? Design tasks can be cleanly separated, thus allowing the tasks to be completed in parallel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>? Specialization and division of labor possible.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>? Architecture innovation may be difficult.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>? Requires the top-down creation of global product architecture.</td>
</tr>
</tbody>
</table>

Erens, and Verhulst (1997) asserted that the development of a product family requires product architectures in three domains - defined functional domain, technology domain and physical domain of product architecture. They mainly discussed the structural aspects of development, namely, the definition of a product family in different domains, including the relationships between these
domains. They also compared modularity and integration. Modularity corresponds to flexibility and changeability, whereas integration corresponds to stability and optimization. They defined a modular design to be a design in which a restricted number of functions is allocated to a module, or a restricted number of modules is allocated to a physical assembly. Finally, they drew the following conclusions:

1. Product architectures are essential to separate the stable and variable parts of design. The stable aspects create a framework within which a variety of products can be developed.
2. The standardization of interfaces in one domain improves the possibility of combining components in such a way that a large variety in that domain is created.
3. The standardization of interfaces and N:1 function-module mappings reduce the number of technology modules and physical assemblies that is needed to create commercial variants in the functional domain.
4. The architecture of a product family is decoupled from the architectures of its components. The variety of these components has no consequences for the external interfaces of these components, which reduces design complexity.
5. Deviations from the product family definition (i.e. distributed functions and distributed technology modules) are effectively controlled with the generic product-structuring concept.
Determining product architecture and identifying modules are the key activities of product development. They are made during the early phases of the innovation design process. There are some researches related to determining product architecture and / or identifying modules for product development.

Erixon, and Ostgren (1993) proposed a method for modular design by naming Quality Function Deployment (QFD) matrix for modular analysis the Modular Function Deployment (MFD) matrix, and by outline a new evaluation tool. The method consists of:

(1) Ordinary QFD-analysis, making sure that the right product specification is derived.

(2) Module creation and analysis of manufacturing goals.

(3) New developed MFD-matrix (Modular Function Deployment).

(4) Evaluation of complexity for assortment and interfaces.

(5) Traditional DFMA analysis of each module.

The MFD-matrix is a QFD like way to give an indication of which sub-functional group(s) can form a module in terms of product development, manufacturing, product variety, purchasing and after sales. This matrix gives a good picture of which sub-functional groups that have one or more reasons to form modules. It also tells which of the sub-function groups have the strongest motive of becoming modules. Then, they evaluated the modular concept in the following areas:
(1) Lead-time in assembly – number of modules in product

(2) Manufacturing and assembly support system cost

(3) Product cost – assortment complexity

(4) Quality

(5) Lead time in development

(6) Development cost

(7) Development capacity

An approach to identify modules by analyzing the design matrix is proposed by Tseng, and Jiao (1997), which presents the mapping relationships between design objectives (function requirement) and physical solutions (Design Parameters) using group technology and axiomatic design. The procedure of module identification based on axiomatic design is suggested in the following table:

**Table 2-2: Steps of Module Identification**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The FRs are defined in the functional domain to satisfy a given set of customer needs.</td>
</tr>
<tr>
<td>2</td>
<td>Ideation of design solutions. A set of DPs are selected to satisfy the FRs defined in step 1. The designer finds a set of plausible DPs corresponding to each given FR of a set of FRs.</td>
</tr>
<tr>
<td>3</td>
<td>Construction of design matrix. The zigzagging process between FRs hierarchy and DPs hierarchy relates FRs and DPs in the design matrix at different level of abstraction.</td>
</tr>
<tr>
<td>4</td>
<td>Modular analysis. The clustering analysis is used to determine module boundaries.</td>
</tr>
</tbody>
</table>
However, the term “module” in above two researches refers to use of interchangeable units to create product variants, and allows one-to-many mappings from functions to modules. Therefore, it is not clear which part of function is realized in a technology module, and any change in functionality may require a change to several modules.

Yu, Gonzales-Zugasti, and Otto (1999) studied the relationship between customer need distributions and appropriate product architecture, and proposed a customer need basis for defining the architecture of a portfolio of products. They defined that one can assess a market population to establish target values for product features and present those targets as probability distributions (population distribution). One can trace the product through its use over time and establish a separate set of desired target values as a set of distributions (time distribution). Comparing these two distribution sets for every important customer needs can point to the type of architecture a market population desires. When population and time distributions match, feature adjustability is required. When these distributions are different but constant in time, a family of product varieties is more appropriate. When the population distribution changes over time, the feature must be isolated so it can be upgraded over time. If the distributions across both time and population are narrow, a single offering will supply the needs of the market. Their market-based architecture selection process is presented in the following table:
Table 2-3: Market-based Architecture Selection Process

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Interview customers to identify customer needs.</td>
</tr>
<tr>
<td>2</td>
<td>Send out questionnaires to determine average importance of each need. Sort out the more important needs.</td>
</tr>
<tr>
<td>3</td>
<td>Survey customers for need target values; calculate mean and standard deviation population distribution.</td>
</tr>
<tr>
<td>4</td>
<td>Choose typical customers from different segments and trace these values over time by sampling in different uses</td>
</tr>
<tr>
<td>5</td>
<td>Calculate mean, standard deviation for sample at different uses, for each need.</td>
</tr>
<tr>
<td>6</td>
<td>For each need, create table of mean and deviations from time-trace segments.</td>
</tr>
<tr>
<td>7</td>
<td>Based on these parameters, select the best architecture.</td>
</tr>
</tbody>
</table>

This method may be used to determine the type of product architecture in the beginning of product concept stage. However, they did not further discuss the development of product architect.

McAdams, Stone, and Wood (1999) studied functions of a set of product varieties based on customer needs. They introduced the concept of a quantitative function model – a vector representation of a product in which the number of distinct sub-functions determine the dimension of the vector and the customer need rankings of each sub-function determine the magnitude. They also developed a method for normalizing function vectors across a set of product varieties to rank the sub-functions, because different product complexity and customer enthusiasm (during the customer need acquisition process) will affect the magnitude of product function vectors. The philosophy used to normalize the function vectors consists of two complimentary aspects:
(1) All products are of equal importance. To equalize products, the customer
need value of each function is scaled so that sum of a given product's
importance level is equal to the average sum of the customer need
importance for all products.

(2) Products with more functions are more complex. To represent varying levels
of product complexity, each product function is scaled by the ratio of the
number of functions in that product to the average number of functions per
product. This method for normalizing the customer need rating of sub-
functions of product family will be used in this research.

design methodology.

| Table 2-4: Overview of Product Architecture Design Methodology |
|-----------------|-----------------|-----------------|
| **Step 1:**    | **Function**    | **Function**    |
| Gather Customer| modeling        | modeling        |
| Needs          | phase           | phase           |
| ? Interview    |                 |                 |
| ? Interpret    |                 |                 |
| ? Rank customer needs |         |                 |
| **Step 2:**    | **Function**    | **Function**    |
| Derive Function| modeling        | modeling        |
| Model          | phase           | phase           |
| ? Generate black box model |       |                 |
| ? Create function chains-sequential vs. parallel | |                 |
| ? Aggregate function chain into function model |     |                 |
| **Step 3:**    | **Function**    | **Function**    |
| Identify Product| modeling        | modeling        |
| Architecture   | phase           | phase           |
| ? Apply heuristic to identify modules (Dominant flow, Branching flow, Conversion-transmission) |     |                 |
| ? Choose unique modules for development |     |                 |
| **Step 4:**    | **Function**    | **Function**    |
| Generate Modular| modeling        | modeling        |
| Concepts       | phase           | phase           |
| ? Create rough geometric layouts |       |                 |
| ? Search for existing components |       |                 |
| ? Search for create modules and select concept |     |                 |
They also introduced three heuristic methods of dominant flow, branching flow and conversion-transmission function chains to identify modules of a product from a functional model. The heuristic method shifts the focus to modular concept variants where solutions are sought at a modular level rather than a functional level.

In dominant flow heuristic, the identified sub-functions form the boundary, or interface, of the module. Any other flows, in addition to the traced flow, that cross the boundary are interactions between the module and the remaining product. To implement the module, conduits must be specified to carry the interactions across the interface. The dominant flow heuristic is stated formally as: "the set of sub-functions which a flow passes through, from entry or initiation of the flow in the system to exit from the system or conversion of the flow within the system, define a module".

![Diagram of Dominant Flow Module](image)

**Figure 2-1: Dominant Flow Heuristic Applied to a Function Structure**
Branching flow heuristic requires identification of flows associated with parallel function chains. Each limb of a parallel function chain defines a potential module and is shown in Figure 2-2. The module is formed of the sub-functions that make up the limb (technically, each limb consists of a sequential function chain). All modules (one per limb) must interface with the product at the flow’s branch point. All flows that cross this interface are the interactions between the remaining product and the module. The branching flow heuristic is stated formally as: “the limbs of a parallel function chain constitute modules. Each of the modules interfaces with the remainder of the product through the flow at the branch point”.

![Diagram showing branching flow modules](image)

**Figure 2-2: Flow Branching Heuristic Applied to a Function Structure**

The conversion-transmission heuristic method deals with conversion sub-functions and conversion to transmission chains. Conversion sub-functions accept a flow of material or energy and convert the flow to another form of material or energy. In standard verb-object form, a conversion sub-function
appears as convert flow A to flow B. The method of the conversion-transmission heuristic is shown in Figure 2-3. The essential actions are as: identify conversion sub-functions and check for transmit or transport sub-functions downstream of the converted flow. If none exist, then the conversion sub-function is a module by itself. If transmit or transport sub-functions exist without any other sub-functions between them, then the convert-transmit (transport) pair represents a module. If other sub-functions exist between the convert and transmit (transport) sub-functions and those intermediate sub-functions only operate on the converted flow (i.e. the object in the sub-function verb-object pair is the converted flow), then the conversion-transmission (transportation) chain represents a module. Interfaces of a conversion-transmission module are defined in a similar manner as those for a dominant flow module. Two necessary interactions across the interface are the flow to be converted and the exiting converted flow. Additional flows may also cross the interface. The conversion-transmission heuristic is stated formally as: “a conversion sub-function or a conversion-transmission pair or proper chain of sub-functions constitutes a module”.

Figure 2-3: Conversion-transmission Applied to a Set of Sub-functions
Dahmus, Gonzalez-Zugasti, and Otto (2000) presented an approach to create a modular architecture for product family that shares inter-changeable modules. Rather than a fixed product platform upon which derivative products are created through substitution of various add-on modules, their approach permits the platform itself to be one of several possible sizes or types. Thus, the system is a collection of modules, each of which can be one of several types. They began by developing function structures of each product in the portfolio, where each embodies a specific physical principle underlying the common technology. These function structures are then compared to determine common and unique modules. Product modularity rules (i.e. dominant flow, branching, and conversion) are then applied to determine further possible modules. Application of any consistent set of modularity rules defines feasible portfolio architecture. Each portfolio architecture is presented using a modularity matrix of functions versus products, with shared/unique function levels indicated in the matrix.

Some researches focusing on the product variety design, they considered product varieties that have a modular product architecture already determined.

Fujita, Sakaguchi, and Akagi (1999) reviewed the module based product variety with a mixture of views from customer's need, functions, manufacturing modules and hierarchical representation of systems. They defined three views for understanding and designing a modular product.
(1) Customer view: customer’s needs and interests in customer’s language. This view relates to subjective origins of a product.

(2) Function view: function structure that emerges and explains technological phenomena of an artifact. This well corresponds to physical laws.

(3) Manufacturing view: structure or sequence in which a product is established from raw material through parts, components and modules.

Every view form hierarchical structures respectively, and the mapping from a view to another forms the relationships across individual views as shown in Figure 2-4.

Figure 2-4: Mapping among Three Views of Product Design

Then, they discussed product variety design under modular architecture and module commonalization toward a computational methodology based on a design problem of television receiver circuits. They applied three constraints in their product variety design method:
(1) Diversion feasibility constraint: A constraint that prohibits the diversion of a module originally for a product to another product. It is caused by function levels.

(2) Simultaneity constraint: A constraint that requires that a module and another module must be simultaneously diverted due to functional coupling.

(3) Capacity constraint: A constraint that determines the capacity of a hidden module, such as the capacity aspect of a power-supply module.

Gonzalez-Zugasti, and Otto (2000) presented a method for designing families of products built onto modular platforms. The problem of designing a family of products based on such a platform is formulated as an optimization exercise, from which an implementation is derived. They use a set $F$ to describe the desired family of $P$ products, where $F = \{A, B, C, ..., P\}$; and describe the design of each product $i$ by the vector of design variables $x_i$, where $i = A, B, C, ..., P$. Each product is split into modules indicated by the subscript $j$, given by the chosen generic product architecture, where $j = 1, 2, 3,..., J$. That is, all products have the same number of modules, $J$, even if some may not be used in particular members of the product family. Lastly, each module $j$ can have many instances that could be interchanged to provide the desired function at various levels, and are indicated by the superscript $k$, where $k = 1, 2, 3, ...K_j$. Each module can then have a different number of possible instances, $K_j$. The design space of all the elements that can be used in the family is then described by:

$$X = (x_1^1, x_1^2, ..., x_1^{K_1}, x_2^1, x_2^2, ..., x_2^{K_2}, ..., x_J^1, ..., x_J^{K_J})$$ (2.1)
Where $x_i^k$ represents the $j^{th}$ module $k^{th}$ instance

Then, they define the module mix matrix, $M$, to represent the mix of module instances used in a product family, such that:

$M_{ij} = 0$, if module $j$ is not used in product $i$.

$M_{ij} = k$, otherwise, where $k$ is the instance used for module $j$ in product $i$.

Finally, they formulate the problem of designing a product family as a multi-objective optimization. The problem is presented briefly as follows:

$$\text{Min}_{x,M} \begin{cases} f_i(x_i,M) \\ f(X,M) \end{cases} \quad i = A, B, \ldots, P$$

(2-2)

Subject to:

(1) Constraints on each variant.

(2) Constraints on the family as a whole.

(3) Constraints from sharing.

(4) Module compatibility constraints.

The objectives in the formulation depend on the design of the whole family, such as maximizing the profit for the set of products, and a set of individual objectives, one for each variant. For example, one of the variants may be optimized for a given performance criteria such as speed, while another variant in the same family may be optimized for minimum mass.
Their method allows for the design of the modules that are shared across multiple members of the family, or the platform, as well as the portions of the products that are individually designed. Also, the method allows for the use of both existing module instances from catalogs and new ones to be designed. The result of applying the method is a candidate design for the product family, both the combination of which modules should be shared and across which of the products, and the desired settings for the shared modules and the individual portions of each variety. However, product design is not only the problem of engineering but also the problem of marketing. They did not discuss about the customer needs and product target cost in their research.

There are several researches discussed the modular design, product architecture development and variant configuration in different points of view.

Rosen (1996) studied the design of modular product architectures in discrete design spaces, and formally defined such design spaces for the purposes of life cycle design, including material recycling and service concerns.

Newcomb, Bras, and Rosen (1996) discussed the application of product modularity to design for the life cycle. In their research, modularity with respect to life cycle viewpoints, not just product functionality and structure, was defined and applied in the analysis of product architecture characteristics.
Tsai, and Wang (1996) presented a methodology of modular-based design to support concurrent engineering. They classified the functions into different types of modules according to the correlation in design by using fuzzy cluster identification, and selected the optimal module types based on the considerations of the manufacture and assembly complexities of the system for progressive parallel design.

Gonzalez-Zugasti, Otto, and Baker (1999) presented a model to account for uncertainty present during the development of those product families. Real options concepts are introduced to model the risks and delayed decision benefits present under uncertainty in technologies, funding, etc.

Muffatto (1999) analyzed the introduction of a platform strategy in product development with reference to its application in the automobile industry. He discussed the implication and benefits of a platform strategy from both the technical and organizational points of view. He highlighted the effect the adoption of a platform strategy may have both on the product and on the development process.

Some researchers introduced different kinds of software tools to process the product data using different software techniques. Axling, and Haridi (1996) introduced an interactive configuration tool using knowledge-based techniques and object-oriented methodology. Ariano, and Dagnino (1996) presented a
system using object-oriented techniques, which integrates order entry, product configuration and dynamic generation of bill of materials functions in a customized office of furniture manufacturing company. Murdock, Szykman, and Sriram (1997) introduced an information-modeling framework to support representation of design artifacts for design databases and repositories and developed an object-oriented language for modeling design knowledge. Olsen, Saetre, and Thorstenson (1997) proposed a procedure-oriented generic bill of material structure based on a programming language notation. This notation can be used to describe the set of possible variants of a product by handling both functional and structural relations between components. Dove, Jundt, and Lucas (1998) described the use of object-oriented database (ODBMS) technology to store process configuration data in a process control system. Deneux, and Wang (1999) introduced a software prototype based on a knowledge model for function re-design. It is used to represent expert design knowledge and to support the decision-making process by suggesting solutions to design problems based on a fuzzy representation of design constraints.

2.2 Summary

Based on the literature review of the various approaches for product architecture or product variety design, it can be concluded as follows:

1. Product variety design, or variation, is tightly connected with the product architecture development. However, few researches try to present a
systematic process to support modular design from product architecture development to variant configuration.

2. Product design is the problem of not only engineering but also marketing. Therefore, customer needs, design budget, and product target cost are also key factors that should be taken into account in the modular product design process.

3. Product design is not simply a maximizing effort but an optimizing process as well. Few of current researches try to solved the modular product design problem using Linear Programming (LP) model, and to formulate a distributed system structure for modular design using Unified Modeling Language (UML).
CHAPTER 3  METHODOLOGY AND ALGORITHM

3.1 Overview

In modular design, each module may have more than one instances to provide various performance and interface, the term "module" can be treated as a logic conception with its instances to implement the sub-functions. The elements related to modular product family could be described in the following paragraph.

Generic product architecture consists of a set of modules \((m_1, m_2, \ldots, m_5)\). Product family can be described as a set of products. Product variety consists of a set of module instances \((m_i^j)\) represents the \(i^{th}\) module's \(j^{th}\) instance). An example to show the elements of a product family is given as follows:

![Diagram showing Generic Product Architecture and Product Family](image)

**Figure 3-1: Elements of Product Family**

Based on Figure 3-1, product varieties can be presented as follows:

\(P_1\) includes a set of modules: \(\{ m_1^1, m_1^2, m_1^3, m_1^4 \}\)
P₂ includes a set of modules: \{ m₂^1, m₂^3, m₂^4, m₁^5 \}

P₃ includes a set of modules: \{ m₂^1, m₂^2, m₁^3, m₁^4, m₂^5 \}

The Table 3-1 provides overview of the process for developing modular generic product architecture and configuring product varieties.

### Table 3-1: Overview of the Modular Design Process

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1. Determine independent sub-function ratings for each product concept. | ♦ Identify customer needs and determine average importance of each need  
♦ Create customer need-function matrix for each product concept  
♦ Determine independent sub-function ratings for each concept in terms of customer needs |
| 2. Compute dependent sub-function ratings for product family | ♦ Normalize sub-function ratings for each product concept  
♦ Determine inter-relationships of each sub-function  
♦ Compute dependent sub-function ratings for product family in terms of customer needs |
| 3. Identify the feasible sub-functions | ♦ Estimate the design cost committed to each candidate sub-function  
♦ Identify the feasible sub-functions |
| 4. Construct generic product architecture | ♦ Map the sub-functions to modules to construct a function-module matrix  
♦ Create module instances for each module |
| 5. Configure product varieties | ♦ Analyze product performance  
♦ Analyze product cost constraint  
♦ Analyze product function requirement  
♦ Maximize total performance rating to configure a product variety based on the generic product architecture |
3.2 Methodology and Algorithm

To design new modular product architecture, the customer need and design budget should be taken into account. This section will discuss the process to develop generic product architecture and configure product variety given in Table 3-1 in detail.

3.2.1 Determine Independent Sub-function Ratings for Each Concept

A function vector can be used to represent a product concept in which the number of distinct sub-functions determine the dimension of the vector and the customer need rankings of each sub-function determine the magnitude. This step is to create the function vector to indicate sub-function ratings for each product concept in terms of customer needs, and treats each sub-function as independent. It is divided into three tasks.

(1) Identify customer needs and determine average importance of each need.

Customer needs are customer's expectations of the product and presented by customer's languages, which can be identified by means of interviewing with customers. After identifying a set of customer needs, questionnaires can be used to survey a statistically significant number of customers for the relative importance of each need. Then, the average importance of each need can be determined by taking the mean of the importance. Finally, customer need vector for each product concept can be created to indicate the weight of each customer need.
\[ CN = (r_1, r_2, \ldots, r_N) \]  
(3 - 1)

Where \( r_i \) = average importance weight of the \( i^{th} \) customer need 

\[ N = \text{total number of customer need} \]

(2) Create customer need-function matrix for each product concept.

The usual definition of function is the properties that make something work or sell. It is the result desired by the customer. Any product should have a prime function or basic function. Basic function is the function upon which all other functions (secondary function) depend. The function at the top of a function tree is referred as basic function and all functions at lower level of the hierarchical tree are secondary functions. In this research the term "sub-function" is referred as secondary functions. Generally, product sub-functions are identified by engineering judgment. Most of sub-functions involve the exchange of information, materials, and energy.

After a set of sub-functions for each product concept has been identified, the customer need-function matrix can be created to answer the question that how well each sub-function meets the customer needs. Let \( K \) = the total number of sub-functions and \( N \) = the total number of customer need, the matrix can be presented as:

\[
CF = \begin{pmatrix}
C_{11} & C_{12} & \cdots & C_{1K} \\
C_{21} & C_{22} & \cdots & C_{2K} \\
\vdots & \vdots & \ddots & \vdots \\
C_{N1} & C_{N2} & \cdots & C_{NK}
\end{pmatrix}
\]  
(3 - 2)
Where, $c_{ij} = \text{correlation weight of the } i^{\text{th}} \text{ customer need and } j^{\text{th}} \text{ functions.}$

(3) Determine sub-function ratings for each concept in terms of customer needs.

We can calculate the sum of the value of the customer need-function correlation weight multiplied by weight of customer need to generate the function vector for each concept. Then, the function vector, $F$, can be presented as:

$$F = (f_1, f_2, \ldots, f_K) = (r_1, r_2, \ldots, r_N) \begin{pmatrix}
  c_{11} & c_{12} & \cdots & c_{1K} \\
  c_{21} & c_{22} & \cdots & c_{2K} \\
  \vdots & \vdots & \ddots & \vdots \\
  c_{N1} & c_{N2} & \cdots & c_{NK}
\end{pmatrix}$$

or

$$F = (f_1, f_2, \ldots, f_K) = \left( \sum_{i=1}^{N} r_i c_{i1}, \sum_{i=1}^{N} r_i c_{i2}, \ldots, \sum_{i=1}^{N} r_i c_{iK} \right) \quad (3 - 3)$$

Where $r_i = \text{the importance weight of the } i^{\text{th}} \text{ customer need.}$

$c_{ij} = \text{the correlation weight of the } i^{\text{th}} \text{ customer need and } j^{\text{th}} \text{ functions}$

$K = \text{total number of sub-functions of the product concept}$

$N = \text{total number of customer needs}$

### 3.2.2 Compute Dependent Sub-function Ratings for Product Family

Sub-functions may be dependent on each other. Therefore, the inter-relationship among sub-functions should be considered in product design. Family vector is used to indicate the normalized importance weight for each sub-function of the product family, and considers the correlation weight between each sub-function.

Calculating family vector can be broken into three tasks:
(1) Normalize sub-function ratings for each product concept.

Each product concept of the family may have different sub-functions. Therefore, customer need ratings should be normalized to account for varying complexity.

The function vectors of each product concept can be assembled into a function-product matrix. Each column of this matrix is a function vector gotten from the previous step including all sub-functions of the product family, and zeros are used to indicate the functions that are not presented in a product concept. It can be presented as follows:

$$FP = \begin{pmatrix}
    f_{11}' & f_{12}' & \cdots & f_{1L}' \\
    f_{21}' & f_{22}' & \cdots & f_{2L}' \\
    \vdots & \vdots & \ddots & \vdots \\
    f_{K'1}' & f_{K'2}' & \cdots & f_{K'L}'
\end{pmatrix}$$  \hspace{1cm} (3 - 4)

Where, $f_{ij}' =$ importance weight of the $j^{th}$ product concept's $i^{th}$ function.

($f_{ij}' = 0$, if the $i^{th}$ function is not presented in the $j^{th}$ product concept)

$L =$ total number of product concepts

$K' =$ total number of sub-functions of the product family

It can be realized that all product concepts of the family are of equal importance, and products with more sub-functions are more complex. Then, the method given by McAdams, Stone and Wood (1999) can be used to normalize the sub-function ratings.

First, scale the customer need rating of each sub-function, such that the sum of a given product’s sub-function customer need ratings equals the average sum of
the customer need rating for all products. This normalization expresses each subfunction customer need rating as a fraction of the average sum of customer need ratings for all products. The total customer need rating for the $j^{th}$ product concept of the family is:

$$S_j = \sum_{i=1}^{K'} f_{ij}$$  \hspace{1cm} (3 - 5)$$

The average customer need rating is:

$$\bar{S} = \frac{1}{L} \sum_{j=1}^{L} \sum_{i=1}^{K'} f_{ij}$$  \hspace{1cm} (3 - 6)$$

Then, scale each product concept sub-function by the ratio of the number of subfunctions to the average number of sub-functions per product. Thus, product with more sub-functions receives a higher normalized customer need rating. The number of functions in the $j^{th}$ product is:

$$N_j = \sum_{i=1}^{K'} Y(f_{ij})$$  \hspace{1cm} (3 - 7)$$

The average number of functions is:

$$\bar{N} = \frac{1}{L} \sum_{j=1}^{L} \sum_{i=1}^{K'} Y(f_{ij})$$  \hspace{1cm} (3 - 8)$$

Where \( Y(f_{ij}) = 0, \text{ if } f_{ij} = 0 \); \( Y(f_{ij}) = 1, \text{ if } f_{ij} > 0 \)

Therefore, the $i^{th}$ function’s normalized rating of the $j^{th}$ product concept, $u_{ij}$, can be presented by equation (3-9):

$$u_{ij} = f_{ij} \left( \frac{S}{S_j} \right) \cdot \left( \frac{N_j}{\bar{N}} \right)$$  \hspace{1cm} (3 - 9)$$
The normalized version of function-product matrix, $FP'$, can be presented as:

$$FP' = \begin{pmatrix}
  u_{11} & u_{12} & \cdots & u_{1L} \\
  u_{21} & u_{22} & \cdots & u_{2L} \\
  \vdots & \vdots & \ddots & \vdots \\
  u_{K'1} & u_{K'2} & \cdots & u_{K'L}
\end{pmatrix} \quad (3 - 10)$$

(2) Determine inter-relationships of each sub-function.

Inter-relationships can be treated as the interactions or dependencies of each sub-function. Correlation weight $\gamma_{ij}$ can be used to measure the inter-relationships between the $i^{th}$ and $j^{th}$ sub-function of the product family. It provides early recognition of correlated sub-functions, and can be given by analyzing the interactions or dependencies of among sub-functions. Therefore, a function-function matrix can be created:

$$FF = \begin{pmatrix}
  \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1K'} \\
  \gamma_{21} & \gamma_{22} & \cdots & \gamma_{2K'} \\
  \vdots & \vdots & \ddots & \vdots \\
  \gamma_{K'1} & \gamma_{K'2} & \cdots & \gamma_{K'K'}
\end{pmatrix} \quad (3 - 11)$$

Where, $\gamma_{ij}$ = correlation weight between the $i^{th}$ and $j^{th}$ sub-function.

$K'$= total sub-function number of the product family.

(3) Compute sub-function ratings for product family in terms of customer needs.

Unite normalized version function-product matrix to vector $V$:

$$V = (\sum_{j=1}^{L} u_{1j}, \sum_{j=1}^{L} u_{2j}, \ldots, \sum_{j=1}^{L} u_{K'j}) \quad (3 - 12)$$
Then considering about the inter-relationship of each sub-function, sub-function ratings for product family in terms of customer need can be express as a vector \( P \), which is equal to \( V^*F \).

\[
P = (p_1, p_2, \ldots, p_{K'}) = \left( \sum_{i=1}^{L} u_{i1}, \sum_{j=1}^{L} u_{2j}, \ldots, \sum_{j=1}^{L} u_{K'j} \right) \begin{pmatrix}
\gamma_{11} & \gamma_{12} & \ldots & \gamma_{1K'} \\
\gamma_{21} & \gamma_{22} & \ldots & \gamma_{2K'} \\
\vdots & \vdots & \ddots & \vdots \\
\gamma_{K'1} & \gamma_{K'2} & \ldots & \gamma_{K'K'}
\end{pmatrix}
\]

or

\[
P = (p_1, p_2, \ldots, p_{K'}) = \left( \sum_{i=1}^{K'} \sum_{j=1}^{L} u_{ij} * \gamma_{i1}, \sum_{i=1}^{K'} \sum_{j=1}^{L} u_{ij} * \gamma_{i2}, \ldots, \sum_{i=1}^{K'} \sum_{j=1}^{L} u_{ij} * \gamma_{iK'} \right)
\tag{3-13}
\]

### 3.2.3 Identify the Feasible Sub-functions

Identifying the feasible sub-functions to be developed for the product architecture is based on importance weight and the design budget. This step can be divided into following two tasks.

1. Estimate the design cost committed to each candidate sub-function.

Cost is the expenditure of money, time, labor, etc, to obtain requirement. After the sub-functions of a product concept has been identified based on the customer needs, the cost of each sub-function can be estimated by engineering judgment and experience. Then, the cost vector \( Q \) can be presented as:

\[
Q = (q_1, q_2, \ldots, q_{K'})
\tag{3-14}
\]

Where \( q_i \) = the design cost committed to the \( i^{th} \) sub-function.
(2) Identify the feasible sub-functions.

In many cases, design budget is not enough to develop all the sub-functions derived from customer needs. To solve this problem, an optimization model can be formulated based on the tradeoff between the customer’s satisfaction and the design budget. The objective is to maximize total customer satisfaction,

\[ \sum_{i=1}^{K} \theta_i p_i \]  \hspace{1cm} (3 - 15)

with total design cost subject to constraint of the design budget.

\[ \sum_{i=1}^{K} \theta_i q_i \leq B \]  \hspace{1cm} (3 - 16)

Where \( \theta_i = 1 \) or \( 0 \), it is decision variable for deciding whether the \( i^{th} \) candidate function will be developed or not.

\( p_i \) = the normalized importance weight for the \( i^{th} \) candidate sub-function.

\( q_i \) = the design cost committed to the \( i^{th} \) sub-function.

\( B \) = the design budget.

From the optimum solution of this linear programming, we can identify the feasibly sub-functions to be developed so that customer satisfaction can be maximized by committing resources to more important sub-functions with which one cost unit can “buy” the highest return in customer satisfaction. The result of this step is a feasible sub-function vector of the product family, \( H \).

\[ H = (h_1, h_2, \ldots, h_{K^*}) \]  \hspace{1cm} (3 - 17)
Where \( h_i = 1 \), represents the \( i^{th} \) feasible sub-function of product family

\[ K'' = \text{total number of feasible sub-functions of the product family} \]

### 3.2.4 Construct Generic Product Architecture

After we get feasible sub-functions of the product family, the next step is mapping the feasible sub-functions to candidate modules, then generate the instances for each module to construct generic product architecture. (As we know, function to module mappings of modular design should be based on one-to-one or many-to-one mappings. Both one-to-many and many-to-many mappings are integral design, it is not clear which part of a sub-function is realized in a technology module.) This step can be divided into two steps.

(1) Map the sub-functions to modules to construct a function-module matrix.

A family sub-function structure can be created base on the feasible sub-functions gotten in the previous step. A function structure is a set of sub-functions interconnected by flows (information, material, or energy flow). It can be used in preliminary identify module of product family by means of three heuristic methods (dominant flow heuristic, branching flow heuristic, conversion-transmission heuristic) introduced in Chapter 2. In addition, shared sub-functions of the product concepts should be grouped into a set of modules. These modules can be reused or shared across the product family. The variant sub-functions can be grouped into the other set of modules, these modules can be used to fulfill the requirements of product varieties. A module vector of the product family can be presented as:
\[ M = (m_1, m_2, \ldots, m_J) \]  

Where \( m_i \) = 1, represents the \( i^{th} \) module of the product family

\[ J = \text{total number of modules} \]

Let \( m_i = 1, \ \forall \ i \in \{1,2,\ldots,J\} \) and \( h_i = 1, \ \forall \ i \in \{1,2,\ldots,K''\} \). Then, the mapping between feasible sub-functions to modules can be formulated as follows:

\[
\begin{pmatrix}
 h_1 \\
 h_2 \\
 \vdots \\
 h_{K''}
\end{pmatrix} =
\begin{pmatrix}
 a_{11} & a_{12} & \cdots & a_{1J} \\
 a_{21} & a_{22} & \cdots & a_{2J} \\
 \vdots & \vdots & \ddots & \vdots \\
 a_{K''1} & a_{K''2} & \cdots & a_{K''J}
\end{pmatrix}
\begin{pmatrix}
 m_1 \\
 m_2 \\
 \vdots \\
 m_J
\end{pmatrix}
\]

(3 - 19)

Where: \( a_{ij} \) is an element of the mapping matrix

\[ a_{ij} = 0, \text{ if the } i^{th} \text{ sub-function is not implemented by } j^{th} \text{ module} \]

\[ a_{ij} = 1, \text{ if the } i^{th} \text{ sub-function is implemented by } j^{th} \text{ module} \]

\[ J = \text{total number of modules} \]

\[ K'' = \text{total number of feasible sub-functions} \]

\[ K'' \geq J \text{ and } \sum_{j=1}^{K''} a_{ij} = 1, \ \forall \ i \in \{1,2,\ldots,K''\} \]

(To ensure 1-to-1 or N-to-1 mapping)

Function-module mapping matrix \( A \) will be used for configuring the product variety in the future.

(2) Create module instances for each module.

After a set of modules of product family has been created based on the customer need, module instances can be defined to construct the generic product
architecture by considering the variation of products. In this research, all modules are supposed to be compatible with each other. Each module may have many instances that are same in functionality, but different in performance and cost. Therefore, performance and cost can be treated as criterions of module instance. The vector $M_i$ can be presented as:

$$M_i = (m_1^i, m_2^i, \ldots, m_X^i)$$  \hspace{1cm} (3 - 20)

Where $m_j^i = 1$, represents the $i^{th}$ module's $j^{th}$ instance is available. $X_i = \text{total number of instances of the } i^{th} \text{ module}$

Each criterion of a module instance will be measured by different methods or units. For example, performance importance weight and instance-performance correlation weight can be used to calculate the global performances of the product concept. Manufacture cost and assembly cost of each instance can be used to calculate the total cost of instance in terms of cost criterion. These criterions will be discussed in next section.

3.2.5 Configure Product Variety

Mittal and Frayman (1989) defined the activity of product variant configuration as follows:

Given: (1). A fix, predefined set of components, where a component is described by a set of properties, ports for connecting it to other components, constraints at each port that describe the components that can connected at that port, and
other structural constraints. (2). Some description of the desired configuration.
(3). Criteria for making optimal selections.

Build: One or more configurations that satisfy all the requirements, where a
configuration is a set of connections and a description of connections between
the components in the set.

Usually, variety configured based on customer need is technically feasible, but
prohibitively expensive. The challenge is to create desired variety economically.
In a modular product, instances of a module implement same sub-functions but
they have different performances and costs. Therefore, product configuration can
be formulated as an optimization problem: the objective is to get a set of module
instances for a product variety that can maximize the product performance rating,
while satisfying the product target cost and function requirements constraints.

(1) Analysis product performance.

Product performance can be defined as how well the product implements its
functions. Economic performance will not be discussed in this section, because it
is highly dependent on the production, service, sales and market activities.
Usually, the performance characteristics that arise from the physical properties of
most of the components of a product are called global performance
characteristics. They are tied to the entire product’s mass, size, shape, noise,
emissions and vibration, etc. The importance weight of each performance
characteristic can be assigned by product planning department. The performance
vector can be presented as:

\[ G = (g_1, g_2, \ldots, g_l) \]  \hfill (3-21)

Where, \( g_i \) = importance weight of the \( i^{th} \) performance characteristic

\( l \) = total number of the performance characteristics

The \( i^{th} \) module instance-performance matrix \( IP_i \) can be used to present the inter-relationships between instances and performances, which can be given by design team.

\[ IP_i = \begin{pmatrix}
\sigma_{i1}^{i1} & \sigma_{i1}^{i2} & \ldots & \sigma_{i1}^{il} \\
\sigma_{i2}^{i1} & \sigma_{i2}^{i2} & \ldots & \sigma_{i2}^{il} \\
\vdots & \vdots & \ddots & \vdots \\
\sigma_{iX_i}^{i1} & \sigma_{iX_i}^{i2} & \ldots & \sigma_{iX_i}^{il}
\end{pmatrix} \]  \hfill (3-22)

Where, \( \sigma_{jk}^i \) = instance-performance correlation weight, it present how well the \( i^{th} \) module’s \( j^{th} \) instance contribute to the \( k^{th} \) product performance.

\( X_i \) = total number of instances of the \( i^{th} \) module.

Therefore, product performance rating contributed by the \( i^{th} \) module’s \( j^{th} \) instance, \( PP_j^i \), can be presented as:

\[ PP_j^i = \sum_{k=1}^{l} \sigma_{jk}^i g_k \]  \hfill (3-23)
The total product performance rating, $TP$, is the sum of product performance ratings contributed by all module instances:

$$
\sum_{i=1}^{J} \sum_{j=1}^{X_i} \left( d^i_j \ast \left( \sum_{k=i}^{J} \sigma^i_{jk} g_k \right) \right)
$$  

(3 – 24)

Where: $d^i_j = 1$, if the $i^{th}$ module's $j^{th}$ instance will be used in the product.

$d^i_j = 0$, otherwise.

(2) Analysis product cost constraint.

Target cost is used for designing a product for a cost that will allow a price to be acceptable to the customer. It can be established based on a thorough analysis of customer needs and company profitability, and requires product teams to meet cost goals that have been established before detailed design has begun.

Cost estimation is concerned with the prediction of the costs related to a set of processes before they have actually been executed. Usually, the cost record of the previously manufactured products can be used as a template in the cost estimation process of the new products. The total cost of a product variety can be treated as the sum of the instance's costs plus the assembly costs. It can be assumed that the final assembly costs of every product variety only relate to total number of modules. The cost for assembling each module is a constant $R$. Then, total cost of the product variety, $TC$, can be presented as follows:
\[ TC = \sum_{i=1}^{J} \sum_{j=1}^{X_i} [d_{ij}^i \ast (C_{ij}^i + R)] - R \]  

(3 - 25)

Where \( C_{ij}^i \) = cost of the \( i^{th} \) module's \( j^{th} \) instance

\( R = \) cost for assembling one module instance into product variety

\( d_{ij}^i = 1, \) if the \( i^{th} \) module's \( j^{th} \) instance is selected

\( d_{ij}^i = 0, \) if otherwise

\[ \sum_{j=1}^{X_i} d_{ij}^i \leq 1 \]  (Only one instance can be selected for each module)

Therefore, the cost constraint can be express as: estimated total cost of product variety should equal to or less than the product target cost.

\[ \sum_{i=1}^{J} \sum_{j=1}^{X_i} [d_{ij}^i \ast (C_{ij}^i + R)] - R \leq T \]  

(3 - 26)

Where, \( T = \) target cost of the product variety.

(3) Analysis product function requirement.

After the generic product architecture is developed, each module is associated to one or more certain sub-functions. Once a set of sub-function requirements is fix, the candidate modules for the product variety can be selected. The sub-function requirement vector of a candidate product is given as:

\[ W = (w_1, w_2, \ldots, w_K) \]  

(3 - 27)

Where \( w_i = 1, \) if the \( i^{th} \) sub-function is included in the requirement

\( w_i = 0, \) otherwise
Then, the mapping between feasible sub-functions to candidate module can be formulated as:

\[
\begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1J} \\
  a_{21} & a_{22} & \cdots & a_{2J} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{K'\cdot1} & a_{K'\cdot2} & \cdots & a_{K'\cdotJ}
\end{pmatrix}
\begin{pmatrix}
  m_1 \sum_{j=1}^{x_1} d_{j}^1 \\
  m_2 \sum_{j=1}^{x_2} d_{j}^2 \\
  \vdots \\
  m_J \sum_{j=1}^{x_J} d_{j}^J
\end{pmatrix} =
\begin{pmatrix}
  w_1 \\
  w_2 \\
  \vdots \\
  w_{K''}
\end{pmatrix}
\]  

(3 - 28)

Where: 

- \( K'' \) = total number of sub-functions of the product family
- \( J \) = the total number of modules in the product family
- \( X_i \) = total number of instances of the \( i \)th module
- \( m_i = 1 \), presents the \( i \)th element of module vector of the product family
- \( d_j^i \) is the decision variable defined before.

Considering of the many to one mappings (many functions may be realized by one module), we should remove the constraints in which the value of sub-function requirement is zero (\( w_i = 0 \)).

(4) Maximize total performance rating to configure a product variety based on the generic product architecture.

Summarize the above discussion, the product configuration problem can be formulated as following optimization model:

The objective is to maximize the total performance rating of product variety, see formula (3-24).
\[
\sum_{i=1}^{J} \sum_{j=1}^{X_i} \left( d_{ij} \ast \left( \sum_{k=1}^{I} \sigma_{jk} g_k \right) \right) = \sum_{i=1}^{J} \sum_{j=1}^{X_i} \left( d_{ij} \ast \left( \sum_{k=1}^{I} \sigma_{jk} g_k \right) \right) \tag{3-24}
\]

Subject to:

(1) function constraint (exclude the constraints in which the value of sub-function requirement is zero).

\[
\begin{pmatrix}
    a_{11} & a_{12} & \cdots & a_{1J} \\
    a_{21} & a_{22} & \cdots & a_{2J} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{K-1} & a_{K-2} & \cdots & a_{K-J}
\end{pmatrix}
\begin{pmatrix}
    m_1 \sum_{j=1}^{X_i} d_{ij} \\
    m_2 \sum_{j=1}^{X_i} d_{ij}^2 \\
    \vdots \\
    m_j \sum_{j=1}^{X_i} d_{ij}^j
\end{pmatrix} = \begin{pmatrix}
    w_1 \\
    w_2 \\
    \vdots \\
    w_{K'''}
\end{pmatrix}
\tag{3-28}
\]

(2) cost constraint.

\[
\sum_{i=1}^{J} \sum_{j=1}^{X_i} \left[ d_{ij} \ast (C_{ij} + R) \right] - R \leq T \tag{3-26}
\]

Where, 
\( a_{ij} = 0 \), if the \( i^{th} \) sub-function is not implemented by \( j^{th} \) module
\( a_{ij} = 1 \), if the \( i^{th} \) sub-function is implemented by \( j^{th} \) module
\( C_{ij} \) = cost of the \( i^{th} \) module's \( j^{th} \) instance
\( d_{ij} \) = 1, if the \( i^{th} \) module's \( j^{th} \) instance will be used in the product.
\( d_{ij} = 0 \), otherwise.
\( g_i \) = importance weight of the \( i^{th} \) performance characteristic
\( l \) = total number of the performance characteristics
\( J \) = the total number of modules in the product family
\( K''' \) = total number of sub-functions of the product family
\( m_i = 1 \), represents the \( i^{th} \) element of module vector of the product family

\( N_{ij} \) = the number of the \( i^{th} \) module's \( j^{th} \) instances used in the variety

\( R \) = cost for assembling one module instance into product variety

\( S \) = total number of module instances of the family

\( T \) = target cost of the product variety

\( w_i = 1 \), if the \( i^{th} \) sub-function is included in the requirement

\( w_i = 0 \), otherwise.

\( X_i \) = total number of instances of the \( i^{th} \) module.

\( \sigma_{jk}^i \) = instance-performance correlation weight, it represents how well the

\( i^{th} \) module's \( j^{th} \) instance contribute to the \( k^{th} \) product performance.
CHAPTER 4 CASE STUDY – Power Screwdriver Design

In this chapter, an simple example will be given to show how to construct generic modular product architecture of a power screwdriver based on three product concepts (see Figure 4-1), and configure a product variety based on the generic architecture using the methodology presented in the previous chapter.

4.1 Determine Independent Sub-function Ratings for Each Concept

(1) Identify customer needs and determine average importance of each need.

Customer needs of the power screwdriver can be identified by means of interviewing with customers. Questionnaires can be used to determine average importance of each customer need. Then, the values of customer need vector for the first screwdriver concept 1 are listed in Table 4-1. The scales of ranking are 1,2,3,4,5, with 5 being the most important.

<table>
<thead>
<tr>
<th>Customer need No.</th>
<th>Description</th>
<th>Average customer need weight for concept 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>Powerful</td>
<td>4</td>
</tr>
<tr>
<td>$r_2$</td>
<td>Fast</td>
<td>3</td>
</tr>
<tr>
<td>$r_3$</td>
<td>Reversible (screw and unscrew)</td>
<td>4</td>
</tr>
<tr>
<td>$r_4$</td>
<td>Long lasting battery</td>
<td>5</td>
</tr>
<tr>
<td>$r_5$</td>
<td>Manual use capability</td>
<td>5</td>
</tr>
<tr>
<td>$r_6$</td>
<td>Interchangeable tips</td>
<td>3</td>
</tr>
<tr>
<td>$r_7$</td>
<td>Comfortable handle</td>
<td>2</td>
</tr>
</tbody>
</table>
Therefore, customer need vector for each product concept can be created to indicate the weight of each customer need, see equation (3-1).

\[ CN_1 = (4,3,4,5,5,3,2) \]

**Figure 4-1: Three Power Screwdriver Concepts**

(2) Create customer need-function matrix for each product concept.

Usually, sub-functions involve the exchange of information, materials, and energy. Product sub-functions are defined by engineering judgment. Defining the function of a given object involves asking the question "What is the action" and then expressing the answer in terms of a verb and a noun. In this case “sub-function” is referred as secondary functions at the same hierarchical level. After sub-functions for each product concept have been defined, they can be listed in Table 4-2. Then, a customer need-function matrix (see Table 4-3) can be created to answer the question that how well each sub-function meets the customer
needs. The scales of correlation weight are 0, 1, 3, 9, to represent the no, weak, moderate, and strong relationship between customer needs and sub-functions.

Table 4-2: Function List of Screwdriver Concept 1

<table>
<thead>
<tr>
<th>Function No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>Actuate electricity</td>
</tr>
<tr>
<td>$f_2$</td>
<td>Change torque</td>
</tr>
<tr>
<td>$f_3$</td>
<td>Convert electricity to torque</td>
</tr>
<tr>
<td>$f_4$</td>
<td>Input hand torque</td>
</tr>
<tr>
<td>$f_5$</td>
<td>Import human hand</td>
</tr>
<tr>
<td>$f_6$</td>
<td>Secure rotation</td>
</tr>
<tr>
<td>$f_7$</td>
<td>Secure bit</td>
</tr>
<tr>
<td>$f_8$</td>
<td>Store electricity</td>
</tr>
<tr>
<td>$f_9$</td>
<td>Supply electricity</td>
</tr>
<tr>
<td>$f_{10}$</td>
<td>Transmit torque</td>
</tr>
</tbody>
</table>

Table 4-3: Customer Need-function Matrix of Screwdriver Concept 1

<table>
<thead>
<tr>
<th></th>
<th>$f_1$</th>
<th>$f_2$</th>
<th>$f_3$</th>
<th>$f_4$</th>
<th>$f_5$</th>
<th>$f_6$</th>
<th>$f_7$</th>
<th>$f_8$</th>
<th>$f_9$</th>
<th>$f_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>$r_2$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$r_3$</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$r_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$r_5$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$r_6$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$r_7$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(3) Determine sub-function ratings for each concept in terms of customer needs.

Function vector $F_1$ for concept 1 can be created by means of calculating the sum of the value of the customer need-function correlation weight multiplied by weight of customer need, see equation (3-3). That is:
\[ F^*_1 = (f_1, f_2, \ldots, f_k) = (4,3,4,5,5,3,2) \]

\[
\begin{pmatrix}
0 & 9 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 9 \\
0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 9 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 \\
0 & 0 & 0 & 9 & 3 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 \\
\end{pmatrix}
\]

\[ F^*_1 = (36,36,45,45,27,9,27,45,39,63) \]

Using the same method, we can get the function vectors for the other two product concepts (see Figure 4-2). Finally, the total potential sub-functions of the product family are summarized in Table 4-4:

**Table 4-4: Function List of the Screwdriver Family**

<table>
<thead>
<tr>
<th>Function No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_1' )</td>
<td>Actuate electricity</td>
</tr>
<tr>
<td>( f_2' )</td>
<td>Change torque</td>
</tr>
<tr>
<td>( f_3' )</td>
<td>Convert electricity to torque</td>
</tr>
<tr>
<td>( f_4' )</td>
<td>Input hand torque</td>
</tr>
<tr>
<td>( f_5' )</td>
<td>Import human hand</td>
</tr>
<tr>
<td>( f_6' )</td>
<td>Dissipate torque</td>
</tr>
<tr>
<td>( f_7' )</td>
<td>Regulate electricity</td>
</tr>
<tr>
<td>( f_8' )</td>
<td>Regulate rotation</td>
</tr>
<tr>
<td>( f_9' )</td>
<td>Regulate translation</td>
</tr>
<tr>
<td>( f_{10}' )</td>
<td>Secure rotation</td>
</tr>
<tr>
<td>( f_{11}' )</td>
<td>Secure bit</td>
</tr>
<tr>
<td>( f_{12}' )</td>
<td>Store electricity</td>
</tr>
<tr>
<td>( f_{13}' )</td>
<td>Supply electricity</td>
</tr>
<tr>
<td>( f_{14}' )</td>
<td>Transmit torque</td>
</tr>
<tr>
<td>( f_{15}' )</td>
<td>Swivel handle</td>
</tr>
</tbody>
</table>
Function vectors of the three concepts are listed in the following figure:

![Table of Function Vectors](image)

**Figure 4-2: Function Vectors of Three Screwdriver Concepts**

### 4.2 Compute Dependent Sub-function Ratings for Product Family

(1) Normalize sub-function ratings for each product concept.

A function-product matrix can be created by assembling the function vectors of the three concepts. Each product variety of the family has different sub-functions, so zeros are added to the original function vectors to indicate functions that are not presented in the product concept (see Table 4-5). The total customer need rating $S_j$ and sub-function number $N_j$ of the $j^{th}$ product concept can be calculated by equation (3-5) and (3-7):

$$
S_1 = 36 + 36 + 45 + 45 + 27 + 9 + 27 + 45 + 39 + 63 = 372, \quad N_1 = 10
$$
$$
S_2 = 36 + 45 + 45 + 18 + 36 + 45 + 12 + 12 + 36 + 12 + 39 + 36 + 63 = 435, \quad N_2 = 13
$$
$$
S_3 = 36 + 45 + 36 + 12 + 36 + 12 + 12 + 27 + 12 + 36 + 45 = 321, \quad N_3 = 12
$$
Table 4-5: Function-product Matrix of the Screwdriver Family

<table>
<thead>
<tr>
<th>Function No.</th>
<th>Concept 1</th>
<th>Concept 2</th>
<th>Concept 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1'$</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>$f_2'$</td>
<td>36</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>$f_3'$</td>
<td>45</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>$f_4'$</td>
<td>45</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>$f_5'$</td>
<td>27</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>$f_6'$</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>$f_7'$</td>
<td>0</td>
<td>45</td>
<td>36</td>
</tr>
<tr>
<td>$f_8'$</td>
<td>0</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>$f_9'$</td>
<td>0</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>$f_{10}'$</td>
<td>9</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>$f_{11}'$</td>
<td>27</td>
<td>36</td>
<td>27</td>
</tr>
<tr>
<td>$f_{12}'$</td>
<td>45</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>$f_{13}'$</td>
<td>39</td>
<td>39</td>
<td>36</td>
</tr>
<tr>
<td>$f_{14}'$</td>
<td>63</td>
<td>36</td>
<td>45</td>
</tr>
<tr>
<td>$f_{15}'$</td>
<td>0</td>
<td>63</td>
<td>0</td>
</tr>
</tbody>
</table>

The average sum of the customer need rating for all products and number of sub-functions per product are, according to equation (3-6) and (3-8):

$$\bar{S} = \left( \sum_{i=1}^{3} S_j \right) / 3 = 376, \quad \bar{N} = \left( \sum_{i=1}^{3} N_j \right) / 3 = 11.66$$

Let

$$\lambda_j = \left( \frac{\bar{S}}{S_j} \right) \cdot \left( \frac{N_j}{\bar{N}} \right)$$

Then

$$\lambda_1 = 0.87, \quad \lambda_2 = 0.96, \quad \lambda_3 = 1.21$$

Therefore, the function-product matrix of the screwdriver family can be normalized as, according to equation (3-9).
### Table 4-6: Normalized Function-product Matrix of the Screwdriver Family

<table>
<thead>
<tr>
<th>Function No.</th>
<th>Concept 1</th>
<th>Concept 2</th>
<th>Concept 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1'$</td>
<td>$36\times0.87=31.32$</td>
<td>$36\times0.96=34.56$</td>
<td>$36\times1.21=43.56$</td>
</tr>
<tr>
<td>$f_2'$</td>
<td>$36\times0.87=31.32$</td>
<td>$45\times0.96=43.20$</td>
<td>$0$</td>
</tr>
<tr>
<td>$f_3'$</td>
<td>$45\times0.87=39.15$</td>
<td>$45\times0.96=43.20$</td>
<td>$45\times1.21=54.45$</td>
</tr>
<tr>
<td>$f_4'$</td>
<td>$45\times0.87=39.15$</td>
<td>$18\times0.96=17.28$</td>
<td>$0$</td>
</tr>
<tr>
<td>$f_5'$</td>
<td>$27\times0.87=23.49$</td>
<td>$36\times0.96=34.56$</td>
<td>$36\times1.21=43.56$</td>
</tr>
<tr>
<td>$f_6'$</td>
<td>$0$</td>
<td>$0$</td>
<td>$12\times1.21=14.52$</td>
</tr>
<tr>
<td>$f_7'$</td>
<td>$0$</td>
<td>$45\times0.96=43.20$</td>
<td>$36\times1.21=43.56$</td>
</tr>
<tr>
<td>$f_8'$</td>
<td>$0$</td>
<td>$12\times0.96=11.52$</td>
<td>$12\times1.21=14.52$</td>
</tr>
<tr>
<td>$f_9'$</td>
<td>$0$</td>
<td>$12\times0.96=11.52$</td>
<td>$12\times1.21=14.52$</td>
</tr>
<tr>
<td>$f_{10}'$</td>
<td>$9\times0.87=7.83$</td>
<td>$0$</td>
<td>$12\times1.21=14.52$</td>
</tr>
<tr>
<td>$f_{11}'$</td>
<td>$27\times0.87=23.49$</td>
<td>$36\times0.96=34.56$</td>
<td>$27\times1.21=32.67$</td>
</tr>
<tr>
<td>$f_{12}'$</td>
<td>$45\times0.87=39.15$</td>
<td>$12\times0.96=11.52$</td>
<td>$12\times1.21=14.52$</td>
</tr>
<tr>
<td>$f_{13}'$</td>
<td>$39\times0.87=33.93$</td>
<td>$39\times0.96=37.44$</td>
<td>$36\times1.21=43.56$</td>
</tr>
<tr>
<td>$f_{14}'$</td>
<td>$63\times0.87=54.81$</td>
<td>$36\times0.96=34.56$</td>
<td>$45\times1.21=54.45$</td>
</tr>
<tr>
<td>$f_{15}'$</td>
<td>$0$</td>
<td>$63\times0.96=60.48$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

### Table 4-7: Correlation Weights among Sub-functions

<table>
<thead>
<tr>
<th></th>
<th>$f_1'$</th>
<th>$f_2'$</th>
<th>$f_3'$</th>
<th>$f_4'$</th>
<th>$f_5'$</th>
<th>$f_6'$</th>
<th>$f_7'$</th>
<th>$f_8'$</th>
<th>$f_9'$</th>
<th>$f_{10}'$</th>
<th>$f_{11}'$</th>
<th>$f_{12}'$</th>
<th>$f_{13}'$</th>
<th>$f_{14}'$</th>
<th>$f_{15}'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1'$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_2'$</td>
<td>0</td>
<td>1</td>
<td>0.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_3'$</td>
<td>0</td>
<td>0.6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_4'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_5'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_6'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_7'$</td>
<td>0.6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_8'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_9'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_{10}'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_{11}'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_{12}'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_{13}'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_{14}'$</td>
<td>0</td>
<td>0.6</td>
<td>0.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$f_{15}'$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
(2) Determine inter-relationships of each sub-function.

Correlation weight provides early recognition of correlated sub-functions, and can be given by analyzing the interactions or dependencies of each sub-function based on function structure. The scales of $\gamma$ can be 0, 0.3, 0.6, 1, to represent the no, weak, moderate, and strong relationship between two sub-functions (see Table 4-7). The sub-function correlation matrix, $FF$, can be presented as:

\[
    FF = \begin{pmatrix}
    1 & 0 & 0 & 0 & 0 & 0 & 0.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
    0 & 1 & 0.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.6 & 0 & 0 & 0 \\
    0 & 0.6 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
    0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.3 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0 & 1 & 0.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
    0.6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0.3 & 0 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.3 & 1 & 0 & 0 \\
    0.6 & 0 & 0.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
    0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
    \end{pmatrix}
\]

(3) Compute sub-function ratings for product family in terms of customer needs.

Finally, we can unite function-product matrix into one vector to indicate the independent importance weight of the sub-functions in the family, see equation (3-12):

\[
    V = (\sum_{j=1}^{3} u_{1,j}, \sum_{j=1}^{3} u_{2,j}, ..., \sum_{j=1}^{3} u_{15,j})
\]
Therefore,

\[ V = (109.44, 74.52, 136.80, 56.43, 101.61, 14.52, 86.76, 26.04, 26.04, 22.35, 90.72, 65.19, 114.93, 143.82, 60.48) \]

Then considering about the inter-relationship of each sub-function, sub-function ratings for product family in terms of customer need can be expressed as a family vector, \( P \), according to equation (3-13):

\[ P = V \cdot FF \]
\[ = (161.50, 242.89, 181.51, 99.58, 110.32, 75.49, 152.42, 26.04, 26.04, 22.35, 90.72, 99.67, 134.38, 205.46, 60.48) \]

4.3 Identify the Feasible Sub-functions

(1) Estimate the design cost committed to each candidate sub-function.

Design cost committed to each sub-function can be estimated by engineering judgment and experience. It is given in the following table.

<table>
<thead>
<tr>
<th>Sub-function</th>
<th>Cost (Unit: $1000)</th>
<th>Sub-function</th>
<th>Cost (Unit: $1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_1' )</td>
<td>10</td>
<td>( f_9' )</td>
<td>7</td>
</tr>
<tr>
<td>( f_2' )</td>
<td>30</td>
<td>( f_{10}' )</td>
<td>16</td>
</tr>
<tr>
<td>( f_3' )</td>
<td>20</td>
<td>( f_{11}' )</td>
<td>20</td>
</tr>
<tr>
<td>( f_4' )</td>
<td>10</td>
<td>( f_{12}' )</td>
<td>10</td>
</tr>
<tr>
<td>( f_5' )</td>
<td>6</td>
<td>( f_{13}' )</td>
<td>6</td>
</tr>
<tr>
<td>( f_6' )</td>
<td>15</td>
<td>( f_{14}' )</td>
<td>12</td>
</tr>
<tr>
<td>( f_7' )</td>
<td>8</td>
<td>( f_{15}' )</td>
<td>30</td>
</tr>
<tr>
<td>( f_8' )</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Then, the cost vector $Q$ can be presented as, see equation (3-14):

$$Q = (10, 30, 20, 10, 6, 15, 8, 10, 7, 16, 20, 10, 6, 12, 12)$$

Each element of $Q$ indicates the design cost committed to a sub-function.

(2) Identify the feasible sub-functions.

Suppose the design budget is given, $B = $ 160,000. It is not enough to develop all the sub-functions derived from customer needs. Therefore, tradeoffs between the customer's satisfaction and the design budget should be taken into account. The objective is to maximize total customer satisfaction, see formula (3-15):

$$161.50 \vartheta_1 + 242.89 \vartheta_2 + 181.51 \vartheta_3 + 99.58 \vartheta_4 + 110.32 \vartheta_5 + 75.49 \vartheta_6 + 152.42 \vartheta_7 + 26.04 \vartheta_8 + 26.04 \vartheta_9 + 22.35 \vartheta_{10} + 90.72 \vartheta_{11} + 99.67 \vartheta_{12} + 134.38 \vartheta_{13} + 205.46 \vartheta_{14} + 60.48 \vartheta_{15}$$

Subject to constraint of the design budget, see equation (3-16):

$$10 \vartheta_1 + 30 \vartheta_2 + 20 \vartheta_3 + 10 \vartheta_4 + 6 \vartheta_5 + 15 \vartheta_6 + 8 \vartheta_7 + 10 \vartheta_8 + 7 \vartheta_9 + 16 \vartheta_{10} + 20 \vartheta_{11} + 10 \vartheta_{12} + 6 \vartheta_{13} + 12 \vartheta_{14} + 30 \vartheta_{15} \leq 160$$

with decision variable $\vartheta_i = 0$ or 1, $i = 1, 2, 3, \ldots, 14$

This linear programming problem is solved by Lindo software. The feasible sub-functions are: $f'_1, f'_2, f'_3, f'_4, f'_5, f'_6, f'_7, f'_8, f'_9, f'_{11}, f'_{12}, f'_{13}, f'_{14}$.

Therefore, the feasible sub-function vector of product family can be presented as:

$$H = (h_1, h_2, \ldots, h_{12})$$

Where $h_i = 1$, represents the $i^{th}$ feasible sub-function of product family.
Table 4-9: Feasible Function List of the Screwdriver Family

<table>
<thead>
<tr>
<th>Function No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
<td>Actuate electricity</td>
</tr>
<tr>
<td>$h_2$</td>
<td>Change torque</td>
</tr>
<tr>
<td>$h_3$</td>
<td>Convert electricity to torque</td>
</tr>
<tr>
<td>$h_4$</td>
<td>Input hand torque</td>
</tr>
<tr>
<td>$h_5$</td>
<td>Import human hand</td>
</tr>
<tr>
<td>$h_6$</td>
<td>Dissipate torque</td>
</tr>
<tr>
<td>$h_7$</td>
<td>Regulate electricity</td>
</tr>
<tr>
<td>$h_8$</td>
<td>Regulate translation</td>
</tr>
<tr>
<td>$h_9$</td>
<td>Secure bit</td>
</tr>
<tr>
<td>$h_{10}$</td>
<td>Store electricity</td>
</tr>
<tr>
<td>$h_{11}$</td>
<td>Supply electricity</td>
</tr>
<tr>
<td>$h_{12}$</td>
<td>Transmit torque</td>
</tr>
</tbody>
</table>

The function structure of the product family can be presented as:

Figure 4-3: Function Structure for the Screwdriver Family
4.4 Construct Generic Product Architecture

(1) Map the sub-functions to modules to construct a function-module matrix.

A set of modules to implement the sub-functions derived from the previous step can be identified by means of the heuristic methods introduced in Chapter 2. Shared sub-functions of the product concepts can be grouped into a set of modules. Sub-function to module mappings are based on one-to-one or many-to-one mappings. Then, function-module mapping matrix can be formed by the following mappings:

<table>
<thead>
<tr>
<th>$h_1$: Actuate electricity</th>
<th>$m_1$: Actuate electricity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_2$: Change torque</td>
<td>$m_2$: Change torque</td>
</tr>
<tr>
<td>$h_3$: Convert electricity</td>
<td>$m_3$: Convert electricity</td>
</tr>
<tr>
<td>$h_4$: Input hand torque</td>
<td>$m_4$: Manual use</td>
</tr>
<tr>
<td>$h_5$: Import human hand</td>
<td></td>
</tr>
<tr>
<td>$h_6$: Dissipate torque</td>
<td>$m_5$: Dissipate torque</td>
</tr>
<tr>
<td>$h_7$: Regulate electricity</td>
<td>$m_6$: Regulate electricity</td>
</tr>
<tr>
<td>$h_8$: Regulate translation</td>
<td></td>
</tr>
<tr>
<td>$h_9$: Secure bit</td>
<td>$m_7$: Positioning</td>
</tr>
<tr>
<td>$h_{10}$: Store electricity</td>
<td>$m_8$: Supply electricity</td>
</tr>
<tr>
<td>$h_{11}$: Supply electricity</td>
<td></td>
</tr>
<tr>
<td>$h_{12}$: Transmit torque</td>
<td>$m_9$: Transmit torque</td>
</tr>
</tbody>
</table>

**Figure 4-4: Mapping Sub-functions to Modules**
The module vector of the product family can be presented as

\[ M = (m_1, m_2, \ldots, m_9) \]

Where \( m_i = 1 \), represents the \( i^{th} \) module of product family

Table 4-10 lists the description of each module:

<table>
<thead>
<tr>
<th>Module No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_1 )</td>
<td>Actuate electricity</td>
</tr>
<tr>
<td>( m_2 )</td>
<td>Change torque</td>
</tr>
<tr>
<td>( m_3 )</td>
<td>Convert electricity to torque</td>
</tr>
<tr>
<td>( m_4 )</td>
<td>Manual use</td>
</tr>
<tr>
<td>( m_5 )</td>
<td>Dissipate torque</td>
</tr>
<tr>
<td>( m_6 )</td>
<td>Regulate electricity</td>
</tr>
<tr>
<td>( m_7 )</td>
<td>Positioning</td>
</tr>
<tr>
<td>( m_8 )</td>
<td>Supply electricity</td>
</tr>
<tr>
<td>( m_9 )</td>
<td>Transmit torque</td>
</tr>
</tbody>
</table>

Therefore, sub-function to module mapping matrix, \( A \), is:

\[
A = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]
Figure 4-5 shows the sketch of the modules in the power screwdriver.

Figure 4-5: Modules of the Power Screwdriver

(2) Create module instances for each module.

Each module of the screwdriver's family may have many instances that are same in sub-functions, but different in performances, and costs. The cost record of the previously manufactured products can be used as a template in the cost estimation process of the new products. Suppose the assembly cost for each instance is constant, $R = \$1$. The cost information of each instance is listed as follows:
Table 4-11: Cost of Module Instance

<table>
<thead>
<tr>
<th>Instance</th>
<th>Cost ($)</th>
<th>Instance</th>
<th>Cost ($)</th>
<th>Instance</th>
<th>Cost ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_1^1$</td>
<td>4</td>
<td>$m_1^4$</td>
<td>3</td>
<td>$m_2^7$</td>
<td>3</td>
</tr>
<tr>
<td>$m_2^1$</td>
<td>6</td>
<td>$m_2^4$</td>
<td>4</td>
<td>$m_1^8$</td>
<td>4</td>
</tr>
<tr>
<td>$m_1^2$</td>
<td>4</td>
<td>$m_1^5$</td>
<td>5</td>
<td>$m_2^8$</td>
<td>5</td>
</tr>
<tr>
<td>$m_2^2$</td>
<td>5</td>
<td>$m_2^5$</td>
<td>6</td>
<td>$m_3^8$</td>
<td>6</td>
</tr>
<tr>
<td>$m_3^2$</td>
<td>6</td>
<td>$m_1^6$</td>
<td>4</td>
<td>$m_1^9$</td>
<td>2</td>
</tr>
<tr>
<td>$m_1^3$</td>
<td>5</td>
<td>$m_2^6$</td>
<td>5</td>
<td>$m_2^9$</td>
<td>3</td>
</tr>
<tr>
<td>$m_3^3$</td>
<td>6</td>
<td>$m_1^7$</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performances of the screwdriver family focus on total mass, vibration, and life. The correlation weight that presents how well each instance contributes to the product performance are listed in the following table. The scales are 1,3,9 representing bad, moderate, good instance contributions to product performance.

Table 4-12: Instance-performance Correlation Weight

<table>
<thead>
<tr>
<th>Instance</th>
<th>Mass</th>
<th>Vibration</th>
<th>Life</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_1^1$</td>
<td>3</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>$m_2^1$</td>
<td>9</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>$m_1^2$</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$m_2^2$</td>
<td>3</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>$m_3^2$</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$m_1^3$</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$m_2^3$</td>
<td>9</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>$m_1^4$</td>
<td>3</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>$m_2^4$</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$m_1^5$</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$m_2^5$</td>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$m_3^5$</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$m_4^5$</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$m_1^7$</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$m_2^7$</td>
<td>9</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>$m_1^8$</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$m_2^8$</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$m_3^8$</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$m_1^9$</td>
<td>3</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>$m_2^9$</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>
So far, generic modular product architecture has been developed. Then, a product variety can be configured based on the generic architecture.

4.5 Configure Product Variety

Product configuration should be based on a set of given requirements. Usually, Product Design Specification (PDS) given by the product planning department includes product target cost, performance weights, and function requirements. The PDS of a screwdriver variety is presented in the following figure:

<table>
<thead>
<tr>
<th>Product Design Specification of Power Screwdriver A</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1). Performance weight:</td>
</tr>
<tr>
<td>Scales of weight are 1,2,3,4,5, with 5 being the most important:</td>
</tr>
<tr>
<td>Performance</td>
</tr>
<tr>
<td>Weight</td>
</tr>
<tr>
<td>(2). Product target cost: $36</td>
</tr>
<tr>
<td>(3). Sub-function requirements:</td>
</tr>
<tr>
<td>Sub-function</td>
</tr>
<tr>
<td>Status value</td>
</tr>
</tbody>
</table>

**Figure 4-6: Product Design Specification**

Therefore, product variant configuration can be formulated as an optimization problem based on previous discussion. That is:
Maximize the performance rating of product variety, see formula (3-24):

\[ 66d_1^1 + 90d_1^1 + 22d_2^2 + 54d_2^2 + 108d_3^2 + 30d_3^3 + 90d_3^3 + 66d_4^4 + 108d_2^4 + 30d_3^5 + 60d_2^5 + 36d_2^5 + 108d_2^6 + 36d_3^6 + 90d_2^7 + 28d_3^8 + 36d_2^8 + 108d_3^9 + 54d_4^9 + 108d_2^9 \]

Subject to

(1) function constraint (exclude the constraints in which the value of sub-function requirement is zero), see equation (3-28):

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
\sum_{j=1}^{2} d_j^1 \\
\sum_{j=1}^{3} d_j^2 \\
\sum_{j=1}^{2} d_j^3 \\
\sum_{j=1}^{2} d_j^4 \\
\sum_{j=1}^{2} d_j^5 \\
\sum_{j=1}^{2} d_j^6 \\
\sum_{j=1}^{2} d_j^7 \\
\sum_{j=1}^{2} d_j^8 \\
\sum_{j=1}^{2} d_j^9 \\
\end{bmatrix}
= \begin{bmatrix} 1 \\
1 \\
1 \\
0 \\
0 \\
1 \\
0 \\
1 \\
1 \\
1 \\
\end{bmatrix}
\]

(2) cost constraint, see equation (3-26):

\[ 5d_1^1 + 7d_2^1 + 5d_1^2 + 6d_2^2 + 7d_3^2 + 6d_1^3 + 7d_2^3 + 4d_1^4 + 5d_2^4 + 6d_1^5 + 7d_2^5 + 5d_1^6 + 6d_2^6 + 3d_1^7 + 4d_2^7 + 5d_1^8 + 6d_2^8 + 7d_3^8 + 3d_1^9 + 4d_2^9 - 1 \leq 36 \]
This linear programming problem can also be solved by Lindo or Lingo software.

The optimization instances of the variety are:

\[ m_1^1, m_2^2, m_3^3, m_4^5, m_5^7, m_6^8, m_7^9 \]

Cost of the product variety, "screwdriver A", is $36; the total weight of product performance is 466.
CHAPTER 5 RELEVANT SOFTWARE TECHNOLOGIES

In this chapter, some relevant software technologies will be introduced. These technologies will be used for the software system structure design in this research.

5.1 Introduction of Distributed System

A distributed system is a collection of independent computers that appears to the users of the system as a single computer. The advantages of a distributed system over centralized systems are summarized in the following table.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Economics</td>
<td>Microprocessor offer a better price/performance than mainframe</td>
</tr>
<tr>
<td>Speed</td>
<td>A distributed system may have more total computing power than a mainframe</td>
</tr>
<tr>
<td>Inherent distribution</td>
<td>Some applications involve spatially separated machines</td>
</tr>
<tr>
<td>Reliability</td>
<td>If one machine crashes, the system as a whole can still survive</td>
</tr>
<tr>
<td>Incremental growth</td>
<td>Computing power can be added in small increments</td>
</tr>
</tbody>
</table>

A distributed system provides a way for systems to work together and exchange information at multiple points of continuous integration, from design through manufacturing and support. It can be used for sharing engineering data, maintaining complete product configurations and relationships among product
data, enforcing the rules that describe data flows and processes, and perform notification and messaging functions.

Typically, distributed system can be thought as a three-tiered application. The user-interface tier is the layer of user interaction. It can reside on the user’s desktop, on the organization’s intranet, or on the internet. Several user interface implementations may be deployed which access the same server. The server layer is composed of server-based code with which the client code interacts. This layer can be made up of business objects that perform logic business functions. These objects invoke methods on data store tier objects. The data store layer is made up of objects that encapsulate database routines and interact directly with the DBMS product. The diagram below illustrates a three-tiered distributed application:

![Diagram of a three-tiered distributed application]

Figure 5-1: A Three-tiered Distributed Application

Currently, some technologies like CORBA, DBMS, ODBC or JDBC are widely used in distributed system implementation.
(1) CORBA

Common Object Request Broker Architecture (CORBA) is the ideal programming environment for the three-tiered distributed application. CORBA is the standard distributed object architecture developed by the Object Management Group (OMG) consortium. This standard allows CORBA objects to invoke one another without knowing where the objects they access reside or in what language the requested objects are implemented. There are some benefits of using CORBA as follows:

- CORBA objects can be located anywhere on a network.
- CORBA objects can inter-operate with objects on other platforms.
- CORBA object can be written in any programming language for which there is a mapping from OMG IDL to that language. (e.g. Java, C++, C, COBOL, etc)

The OMG-specified Interface Definition Language (IDL) is used to define the interfaces to CORBA objects. An interface consists of a set of named operations and the parameters to those operations. IDL is the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems. A particular mapping of OMG IDL to a programming language should be the same for all ORB (Object Request Broker) implementations. Language mapping includes definition of the language-specific data types and procedure interfaces to access objects through the ORB.
(2) DBMS

DBMS means database management system. A DBMS is expected to:

- Control access to data from many users at once, without allowing the actions of one user to affect other users and without allowing simultaneous accesses to corrupt the data accidentally.
- Allow users to create new databases and specify their logical data structure.
- Support the storage of very large amounts of data over a long period of time, keeping it secure from accident or unauthorized use and allowing efficient access to the data for queries and database modifications.

(3) ODBC and JDBC

Open Database Connectivity (ODBC) provides a standard interface that allows one application to access many different data sources. A database driver is a dynamic-link library that an application can invoke on demand to gain access to a particular data source. Therefore, the application can access any data source for which a database driver exists.

Java Database Connectivity (JDBC) is also a driver, which consists of a variety of classes and interfaces written in Java that developers can use to write a pure Java program for accessing virtually any relation database. Using JDBC, we can establish a database connection with virtually any relational database, send SQL statement to the database, and process the query results.
5.2 System Overview

The following figure presents the flow chart to develop generic modular product architecture and configure product variety.

Figure 5-2: System Flow Chart
Architecture of the distributed system for generic modular product architecture
development and variant configuration is presented as follows:

![Diagram of distributed system architecture]

Figure 5-3: Distributed System Architecture

5.3 Object Oriented Methodology and UML

The Object Oriented methodology classification emerged in the middle to late
1980s as businesses began to seriously consider object-oriented programming
languages for developing systems. In Object Oriented methodology, an object is
defined as an abstraction of a thing within the problem domain of which the
information system must be aware. A class is set or collection of abstracted
objects that share common characteristics. The main characteristics of an Object-
Oriented methodology include common methods of organization, abstraction, encapsulation or information hiding, inheritance, message communication, associations, and reuse, etc.

UML is a notation language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. As a standard language for software blueprints, UML focus on the conceptual and physical representation of a system. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

The basic relationships used in UML are defined as follows:

(1) Association Relationship
Definition: An association represents a semantic connection between two classes, or between a class and an interface. Associations are bi-directional; they are the most general of all relationships and the most semantically weak.

Graphical Depiction: An association relationship is an oblique or orthogonal line.

![Figure 5-4: Association Relationship](image-url)
(2) Generalize/Inherits Relationship

Definition: A generalize relationship between classes shows that the subclass shares the structure or behavior defined in one or more super-classes. Use a generalize relationship to show a "is-a" relationship between classes.

Graphical Depiction: A generalize relationship is a solid line with an arrowhead pointing to the super class:

![Diagram of Generalize Relationship](image)

Figure 5-5: Generalize Relationship

(3) Dependency Relationship

Definition: A dependency relationship between two classes, or between a class and an interface, shows that the client class depends on the supplier class/interface to provide certain services, such as: the client class accesses a value (constant or variable) defined in the supplier class/interface. Dependencies in the component diagram represent compilation dependencies. The dependency relationship may also be used to show calling dependencies among components, using dependency arrows from components to interfaces on other components.
Graphical Depiction: A dependency relationship is a dashed line with an arrow at one end. The triangle end designates the supplier class. In this example, class A is dependent on class B.

Figure 5-6: Dependency Relationship

(4) Aggregate Relationship

Definition: The aggregate relationship shows a whole and part relationship between two classes. The class at the client end of the aggregate relationship is sometimes called the aggregate class. An instance of the aggregate class is an aggregate object. The class at the supplier end of the aggregate relationship is the part whose instances are contained or owned by the aggregate object. Use the aggregate relationship to show that the aggregate object is physically constructed from other objects or that it logically contains another object. The aggregate object has ownership of its parts.

Graphical Depiction: An aggregate relationship is a solid line with a diamond at one end. The diamond end designates the client class.

Figure 5-7: Aggregate Relationship
(5) Realize Relationship

Definition: A realize relationship between classes and interfaces and between components and interfaces shows that the class realizes the operations offered by the interface.

Graphical Depiction: A realize relationship is a dashed line with an arrowhead pointing to the interface.

![Figure 5-8: Realize Relationship](image)

Generally, it is very difficult to understand a complex system from only one perspective, the UML defines a number of diagrams so that we can focus on different aspects of the system independently. Typically, we can view static parts of a system using class diagram, object diagram, component diagram, and deployment diagram. The dynamic parts of a system can be viewed using use case diagram, sequence diagram, collaboration diagram, state-chart diagram, and activity diagram. In this research, use case diagram, class diagram, sequence diagram and collaboration diagram are used to represent the system. They are briefly introduced in the following paragraphs:

(1) Use case diagram is a diagram that shows a set of use cases and actors and their relationships. The common uses are to model the context of a system and to model the requirements of a system.
(2) Class diagram is a diagram that shows a set of classes, interfaces, and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs. The common uses are to model the vocabulary of a system, to model simple collaborations, and to model a logical database schema.

(3) Sequence diagram is one type of interaction diagram, emphasizing the time ordering of messages. It has two features that distinguish it from collaboration diagram. First, there is the object lifeline. An object lifeline is the vertical dashed line that represents the existence of an object over a period of time. Second, there is the focus of control. The focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure.

(4) Collaboration diagram is the other type of interaction diagram. It emphasizes the organization of the objects that participate in an interaction. Collaboration diagrams have two features that distinguish them from sequence diagrams. First, there is the path to indicate how one object is linked to another. Second, there is the sequence number to indicate the time order of a message.

Finally, these four diagrams consisting of a blueprint of the system are given in the Appendix 1-4, which are designed by the software "Rational Rose". It is the world's leading visual modeling tool for software design using UML.
CHAPTER 6 CONCLUSION

6.1 Achievements and Purpose

This research introduced the modular product design process from concept design phase to embodiment design phase. Product design is a problem of both engineering and marketing. Therefore, this process considered not only product functions and product performances, but also the customer needs, function cost, design budget, module instance cost and target cost. The tradeoffs of these factors derived from two linear programming models give the optimal solutions for modular product architecture development and variation.

Based on the process, a distributed system structure for implementing the automation and collaboration modular product design is analyzed and formulated using UML. Finally, class diagram, use case diagram, sequence diagram and collaboration diagram of the system are designed. This structure can be used for implementing the distributed design support system in practice.

With the distributed system in place, product data can be used as an asset across the enterprise. It can support the reuse of existing modular product configuration data and can have a dramatic impact on product cost, cycle time, and quality. It allows people who are geographically dispersed to work together on the design and release of a product, and also allow large numbers of users to concurrently develop the same product. Product data can be centrally controlled,
but be physically distributed for rapid access by users who require the data. The key benefits delivered by this distributed system will be:

- Reduced product development time.
- Quicker reaction to customer requirements.
- Lower product cost.

Furthermore, this design support system can be integrated into a Product Data Management (PDM) system. PDM technology can provides a way for systems to work together and exchange information at multiple points of continuous integration, from design through manufacturing and support. This system complements and enhances traditional PDM system, going far beyond data management and distribution capabilities by providing the optimization models to ensure that the intended products are the products actually built and delivered. It can also facilitate the dynamic configuration of modular product to meet specific customer requirements.

6.2 Sensitivity Analysis

Sensitivity analysis (also called post-optimality or parametric analysis) deals with the problem of obtaining an optimum feasible solution of modified problem starting with the optimum feasible solution of the old problem. The following paragraphs will briefly discuss some sensitivity issues of the two linear programming models used in this research.
(1) The first model is used to identify the feasible sub-functions. The objective is
to maximize total customer satisfaction:

$$\sum_{i=1}^{K} \theta_i p_i$$  \hspace{1cm} (3-15)

The design cost should less than or equal to the design budget:

$$\sum_{i=1}^{K} \theta_i q_i \leq B$$  \hspace{1cm} (3-16)

- Vary an objective coefficient:

In the objective, see formula (3-15), $p_i$ is the normalized importance weight for the
$i^{th}$ candidate sub-function, and $p_i > 0$. If the only change of the model is that the
value of $p_i$ is increased, then the $i^{th}$ candidate sub-function will become more
feasible. However, the value of $p_i$ is decided by the customer need vector, $CN$. As
discussed before, $CN$ should be determined by surveying a statistically significant
number of customers and taking the mean of the importance. Therefore, $CN$
should not vary much during the product design period.

- Vary the right-hand side value

In the constraint, see equation (3-16), $B$ is the design budget. If $B$ increase, there
is no problem. If $B$ is decreased, then some sub-functions with less importance
will not be developed in the product design. Therefore, $B$ should have its range.
At least, it should be enough for developing the basic sub-functions of the
product. How to determine the design budget and the basic sub-functions is
beyond this research.
(2) The second model is used to configure a product variety based on the generic product architecture. The objective is to maximize the total performance rating of product variety, see formula (3-24).

\[ \sum_{i=1}^{J} \sum_{j=1}^{X_i} \left( d_j^i \ast \left( \sum_{k=1}^{I} \sigma_{jk}^i g_k \right) \right) \]  \hspace{1cm} (3-24)

Subject to:

- function constraint (exclude the constraints in which the value of sub-function requirement is zero).

\[ \begin{pmatrix} a_{i1} & a_{i2} & \ldots & a_{iJ} \\ a_{21} & a_{22} & \ldots & a_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{K'1} & a_{K'2} & \ldots & a_{K'J} \end{pmatrix} \begin{pmatrix} m_1 \sum_{j=1}^{x_i} d_j^1 \\ m_2 \sum_{j=1}^{x_j} d_j^2 \\ \vdots \\ m_J \sum_{j=1}^{x_j} d_j^J \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_{K''} \end{pmatrix} \]  \hspace{1cm} (3-28)

- cost constraint

\[ \sum_{i=1}^{J} \sum_{j=1}^{X_i} [d_j^i \ast (C_j^i + R)] - R \leq T \]  \hspace{1cm} (3-26)

- Vary an objective coefficient:

In the objective, see formula (3-24), \( g_k \) is the importance weight of the \( k^{th} \) performance characteristic given by Product Design Specification (PDS). \( \sigma_{jk}^i \) is
the instance-performance correlation weight, it represents how well the $i^{th}$ module's $j^{th}$ instance contribute to the $k^{th}$ product performance. Once the generic product architecture is developed, the $\sigma_{jk}^i$ should be fixed. Therefore, if the only change of the model is that the value of $g_k$ is increased, then the candidate module instance with higher instance-performance correlation weight of the $k^{th}$ performance will become more probable to be selected.

- Vary the right-hand side value

In the function constraint, see equation (3-28), if the $i^{th}$ sub-function is included in PDS, then $w_i = 1$; otherwise, $w_i = 0$. After the generic product architecture is developed, the function-module mapping matrix $A$ and the instances of each module are fixed. Therefore, vary the sub-function requirement may lead to different modules to be selected for the product.

In the cost constraint, see equation (3-26), $T$ is target cost of the product variety. If $T$ increase, there is no problem. If $T$ is decreased, then the module instance with less cost will be selected. But, the $T$ should not be less than the limit that is able to satisfy the function requirements listed in the PDS. If $T$ is not enough to fulfill the function requirements, then the function requirements in PDS should be changed.

6.3 Future Work

One important further work that should be carried out is to implement the
distributed system, and integrated it into a PDM system. Achieving the implementation would give this research more practical significance.

In other aspects, as the literature review in Chapter 2 indicates, product modules can be identified by means of heuristic methods based on the product function structure. The problem of how to identify product module automatically is beyond this research. In addition, modular design may be not appropriate to be applied in all the levels of product function structure. Therefore, determining the most appropriate function structure level for the product to be developed and identifying product modules automatically can be a research area for future work.
APPENDIX 1 USE CASE DIAGRAM

1: Determine independent sub-function ratings for each product concept

- Give weights of CNs
- Calculate function ratings
- Assign CN-FR co-weight
- Input CNs
- Input FRs
- Designer

2: Compute dependent sub-function ratings for product family

- Calculate function ratings
- Normalize function ratings
- Combine FRs of concepts
- Create function vector of family
- Assign FR-FR correlation weight
- Designer
3: Identify the feasible sub-functions

- Create project and give budget
- Assign design cost of each function
- Compute feasible FRs

Designer

Planning people

4: Construct generic product architecture

- create instances
- createa Modules
- Compute feasible FRs

Designer

5: Configure product varieties

- Planning people
- create PDS
- configure varieties
- createa Modules

Designer
APPENDIX 2 CLASS DIAGRAM

Project <--- ProductConcept <--- Function

CustomerNeed

User

<<Interface>>
DataAccess

ClientGUI

<<Interface>>
ConfigureVariety <--- ProductVariety

<<Interface>>
IdentifyFeasibleFRs

Module

Instance

PDS

Architecture
APPENDIX 3 CLASS AND INTERFACE DEFINITION

CustomerNeed
- CustomerNeedNumber : String
- CustomerNeedDescription : String
- CustomerNeedWeight : short

- add() : boolean
- remove() : boolean
- modify() : boolean
- select() : Vector
- setCustomerNeedNumber() : void
- getCustomerNeedNumber() : String
- setCustomerNeedDescription() : void
- getCustomerNeedDescription() : String
- setCustomerNeedWeight() : void
- getCustomerNeedWeight() : short

The class "CustomerNeed" is used for representing the data structures and operations relate to customer need.

Function
- FunctionNumber : String
- FunctionDescription : String
- FunctionCost : float
- CN-FRWeight : Vector
- FR-FRWeight : Vector

- add() : boolean
- remove() : boolean
- modify() : boolean
- select() : Vector
- setFunctionNumber() : void
- getFunctionNumber() : String
- setFunctionDescription() : void
- getFunctionDescription() : String
- setFunctionCost() : void
- getFunctionCost() : short
- setCN-FR() : void
- getCN-FR() : Vector
- setFR-FR() : void
- getFR-FR() : Vector

The class "Function" is used for representing the data structures and operations relate to functions of concepts, product architecture, and product variety.
The class "Project" is used for representing the data structure and operations related to design project.

<table>
<thead>
<tr>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProjectNumber : String</td>
</tr>
<tr>
<td>ProjectDescription : String</td>
</tr>
<tr>
<td>Concepts : Vector</td>
</tr>
<tr>
<td>ProjectBudget : float</td>
</tr>
<tr>
<td>add() : boolean</td>
</tr>
<tr>
<td>remove() : boolean</td>
</tr>
<tr>
<td>modify() : boolean</td>
</tr>
<tr>
<td>select() : Vector</td>
</tr>
<tr>
<td>setProjectNumber() : void</td>
</tr>
<tr>
<td>getProjectNumber() : String</td>
</tr>
<tr>
<td>setProjectDescription() : void</td>
</tr>
<tr>
<td>getProjectDescription() : String</td>
</tr>
<tr>
<td>setConcepts() : void</td>
</tr>
<tr>
<td>getConcepts() : Vector</td>
</tr>
<tr>
<td>setProjectBudget() : void</td>
</tr>
<tr>
<td>getProjectBudget() : float</td>
</tr>
</tbody>
</table>

The class "ProductConcept" is used for representing the data structures and operations relate to concepts in product design.

<table>
<thead>
<tr>
<th>ProductConcept</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConceptNumber : String</td>
</tr>
<tr>
<td>ConceptDescription : String</td>
</tr>
<tr>
<td>ConceptFRs : Vector</td>
</tr>
<tr>
<td>add() : boolean</td>
</tr>
<tr>
<td>remove() : boolean</td>
</tr>
<tr>
<td>modify() : boolean</td>
</tr>
<tr>
<td>select() : Vector</td>
</tr>
<tr>
<td>setConceptNumber() : void</td>
</tr>
<tr>
<td>getConceptNumber() : String</td>
</tr>
<tr>
<td>setConceptDescription() : void</td>
</tr>
<tr>
<td>getConceptDescription() : String</td>
</tr>
<tr>
<td>setConceptFRs() : void</td>
</tr>
<tr>
<td>getConceptFRs() : Vector</td>
</tr>
<tr>
<td>computeCFRsR() : Vector</td>
</tr>
</tbody>
</table>
### Architecture

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add()</code> : boolean</td>
<td></td>
</tr>
<tr>
<td><code>remove()</code> : boolean</td>
<td></td>
</tr>
<tr>
<td><code>modify()</code> : boolean</td>
<td></td>
</tr>
<tr>
<td><code>select()</code> : Vector</td>
<td></td>
</tr>
<tr>
<td><code>setArchitectureNumber()</code> : void</td>
<td></td>
</tr>
<tr>
<td><code>getArchitectureNumber()</code> : String</td>
<td></td>
</tr>
<tr>
<td><code>setArchitectureDescription()</code> : void</td>
<td></td>
</tr>
<tr>
<td><code>getArchitectureDescription()</code> : String</td>
<td></td>
</tr>
<tr>
<td><code>setArchitectureModules()</code> : void</td>
<td></td>
</tr>
<tr>
<td><code>getArchitectureModules()</code> : Vector</td>
<td></td>
</tr>
<tr>
<td><code>computeFRsR()</code> : Vector</td>
<td></td>
</tr>
<tr>
<td><code>organizeFRsData()</code> : void</td>
<td></td>
</tr>
<tr>
<td><code>identifyFeasibleFRs()</code> : Vector</td>
<td></td>
</tr>
</tbody>
</table>

The class "Architecture" is used for representing data structure and operations related to generic modular product architecture.

### Module

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add()</code> : boolean</td>
<td></td>
</tr>
<tr>
<td><code>remove()</code> : boolean</td>
<td></td>
</tr>
<tr>
<td><code>modify()</code> : boolean</td>
<td></td>
</tr>
<tr>
<td><code>select()</code> : Vector</td>
<td></td>
</tr>
<tr>
<td><code>setModuleNumber()</code> : void</td>
<td></td>
</tr>
<tr>
<td><code>getModuleNumber()</code> : String</td>
<td></td>
</tr>
<tr>
<td><code>setModuleDescription()</code> : void</td>
<td></td>
</tr>
<tr>
<td><code>getModuleDescription()</code> : String</td>
<td></td>
</tr>
<tr>
<td><code>setModuleFRs()</code> : void</td>
<td></td>
</tr>
<tr>
<td><code>getModuleFRs()</code> : Vector</td>
<td></td>
</tr>
<tr>
<td><code>setModuleInstances()</code> : void</td>
<td></td>
</tr>
<tr>
<td><code>getModuleInstances()</code> : Vector</td>
<td></td>
</tr>
</tbody>
</table>

The class "Module" is used for representing the data structure and operations related to product modules.
Instance
- InstanceNumber : String
- InstanceDescription : String
- InstanceCost : float
- PF-INSWeight : Vector

- add() : boolean
- remove() : boolean
- modify() : boolean
- select() : Vector
- setInstanceNumber() : void
- getInstanceNumber() : String
- setInstanceDescription() : void
- getInstanceDescription() : String
- setInstanceCost() : void
- getInstanceCost() : float
- setPF-INS() : void
- getPF-INS() : Vector

The class "Instance" is used for representing the data structure and operations related to module instances.

PDS
- PDSNumber : String
- PDSDescription : String
- PDSFRs : Vector
- PFWeight : Vector
- TargetCost : float

- add() : boolean
- remove() : boolean
- modify() : boolean
- select() : Vector
- setPDSNumber() : void
- getPDSNumber() : String
- setPDSDescription() : void
- getPDSDescription() : String
- setPDSFRs() : void
- getPDSFRs() : Vector
- setPFWeight() : void
- getPFWeight() : Vector
- setTargetCost() : void
- getTargetCost() : float

The class "PDS" is used for representing data structure and operations related to Product Design Specification (PDS).
ProductVariety

@ VarietyNumber : String
@ VarietyDescription : String
@ VarietyInstances : Vector
@ VarietyCost : float

@add() : boolean
@remove() : boolean
@modify() : boolean
@select() : Vector
@setVarietyNumber() : void
@getVarietyNumber() : String
@setVarietyDescription() : void
@getVarietyDescription() : String
@setInstances() : void
@getInstances() : Vector
@setVarietyCost() : void
@getVarietyCost() : float
@organizeCgdData() : void
@configureVariety() : boolean

The class "ProductVariety" is used for representing data structure and operations related to Product Variety

User

@ UserID : String
@ Name : String
@ Password : String
@ DeptNo : String
@ Projects : Vector

@add() : boolean
@remove() : boolean
@modify() : boolean
@select() : boolean
@validateUser() : boolean
@setUserID() : void
@getUserID() : String
@setUserName() : void
@getUserName() : String
@setPassword() : void
@getPassword() : String
@setDeptNo() : void
@getDeptNo() : String
@setProjects() : void
@getProjects() : Vector

The class "User" is used for representing data structure and operations related to client side final users.
The class "ClientGUI" is used for representing data structure and operations related to client side user interface.

```
<<Interface>>
DataAccess
<<
add() : boolean
remove() : boolean
modify() : boolean
select() : Vector
```

This interface is used for accessing the product data via a set of functions.

```
<<Interface>>
IdentifyFeasibleFRs
<<
computeFRSRs() : Vector
organizeFRsData() : void
identifyFeasibleFRs() : Vector
```

This interface is used for identifying the feasible sub-functions to be developed.

```
<<Interface>>
ConfigureVariety
<<
organizeCfgData() : void
configureVariety() : boolean
```

This interface is used for configuring product variety.
APPENDIX 4 COLLABORATION DIAGRAM

1: Determine feasible sub-functions to be developed

------

\[ c : \text{ClientGUI} \]

\begin{align*}
1: \ & \text{computeFFRsR()} \\
\lor \\
5: \ & \text{organizeFRsData()} \\
\lor \\
8: \ & \text{identifyFeasibleFRs()} \\
\lor \\
\end{align*}

<<Server Side>>

\begin{align*}
6: \ & \text{getFunctionCost()} \\
4: \ & \text{getFR-FFR()} \\
3: \ & \text{getCN-FFR()} \\
\end{align*}

\[ a : \text{Architecture} \]

\[ p : \text{Project} \]

\[ pc : \text{ProductConcept} \]
2: Configure product variety

c : ClientGUI

1: organizeCfgData() ->

V

8: configureVariety() ->

<<Server side>>

7: getPF-INS() -> 6: getInstanceCost() ->

v : ProductVariety

ins : Instance

4: getTargetCost() -> 5: getModuleFRs() ->

3: getPFWeight() ->

V

2: getPDSFRs() ->

pds : PDS

m : Module
1: Determine feasible sub-functions to be developed
2: Configure product variety

c : ClientGUI  v : ProductVariety  pds : PDS  m : Module  ins : Instance

1: organizeCfgData()

2: getPDSFRs()

3: return PDS FRs

4: getPFWeight()

5: return PFWeight

6: getTargetCost()

7: return target cost

8: getModuleFRs()

9: return module FRs

10: getInstanceCost()

11: return instance cost

12: getPF-INS()

13: return PF-INS

14: configureVariety()

15: return variety
REFERENCES


VITA AUCTORIS

NAME: Shan Bai

PLACE OF BIRTH: Beijing, China

YEAR OF BIRTH: 1968

EDUCATION: the 80th High School of Beijing, China
1983-1986

Tsinghua University, Beijing, China
Automotive Engineering

University of Windsor, Windsor, Ontario
1999-2001 M. A. Sc.
Industrial & Manufacturing Systems Engineering